TOKENLIVELIHOOD: STUDYING THE ECONOMICS OF SELF-SUSTAINING LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

In the token economy, agents are increasingly completing tasks autonomously, and the token consumption along this way is critical to keeping agents running sustainably, with agents constantly weighing cost against reward. However, existing work remains fragmented: budget-aware inference research treats the computational budget as a static constraint within a single task episode, while agent-based simulations typically assume fixed rewards without modeling the economic cycle of resource consumption and capital accumulation. To address these challenges, we introduce **TokenLivelihood**, a verifiable economic sandbox where computation serves as both production cost and economic capital. Agents choose and spend tokens to perform tasks dynamically, earn rewards from the market, and must sustain their operation solely through their accumulated earnings. The sandbox supports market regimes ranging from monopoly to competition, with prices tied to production costs and agents freely competing for tasks under resource constraints. Extensive experiments show that agents adapt their behavior to different economic environments. Overall, abundant capital encourages more aggressive investment decisions that may hinder long-term wealth accumulation, while intensified competition increasingly favors stronger models. Improvements in productive capability, in contrast, generate broad wealth gains by enabling agents to complete more valuable tasks across different market conditions.[1].

1 INTRODUCTION

As LLM-based agents evolve from single-task assistants into long-lived autonomous systems, they are increasingly capable of performing productive tasks and creating value through interaction with their environments (Wang et al., 2024; Xi et al., 2025). Yet, production requires resources: agents must spend computational resources to pursue tasks and earn returns while preserving sufficient capital for future opportunities. Their continued operation therefore forms an economic cycle of resource consumption, value creation, and capital accumulation, in which agents must make decisions under resource constraints. This raises a fundamental question: how do economic conditions shape the ability of LLM agents to sustain themselves and accumulate wealth?

Existing research largely explores these aspects in isolation. Budget-aware inference (Liu et al., 2025; Wen et al., 2025; Li et al., 2026; Wang et al., 2025; Zhai et al., 2026; Lin et al., 2026) focuses on resource allocation within individual tasks, while generative-agent (Park et al., 2023; Wang et al., 2026) and emerging agent-economy studies (Li et al., 2023; Qi et al., 2026; Zhang & Shang, 2026; Hansen et al., 2026) examine agent interactions and economic behaviors. However, these settings do not systematically capture the economic cycle in which agents consume resources to produce value, earn capital, and reinvest it in subsequent production. It therefore remains unclear how fundamental economic conditions shape agent behavior and long-term economic outcomes.

To study these questions, we introduce **TokenLivelihood**, a verifiable economic sandbox where computation serves as both production cost and economic capital. Agents act as producers in a task market: they consume computational resources to complete tasks, earn rewards upon success, and must sustain future operation through accumulated capital. By integrating cost-based pricing, limited task opportunities, competitive markets, and heterogeneous LLM capabilities, TokenLivelihood

[1] Our code and data will be open-sourced soon.

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

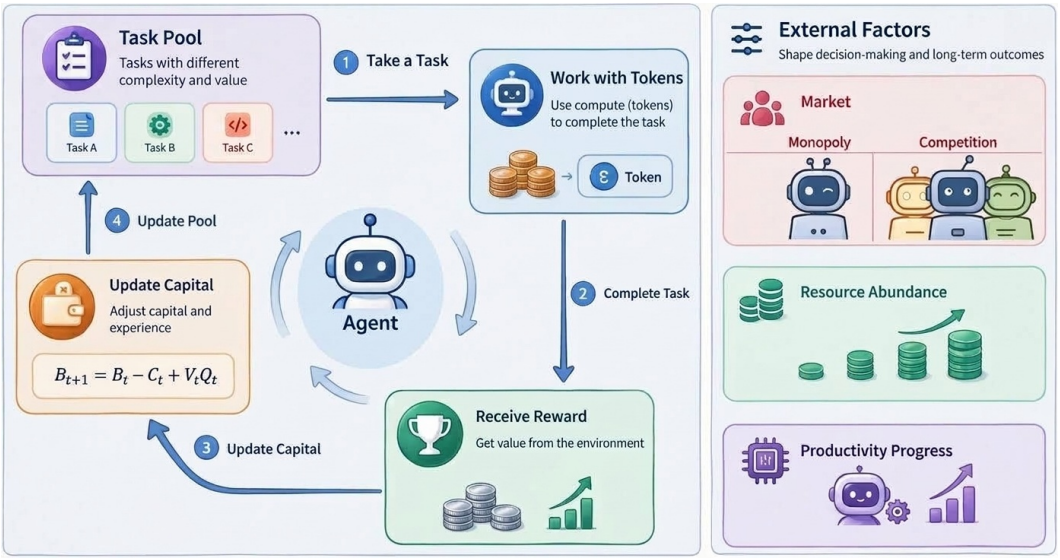


Figure 1: Overview of TokenLivelihood. Agents use their individual capital to select and pay for tasks from a finite task market, earn rewards upon success, and must sustain future operation entirely from accumulated capital, letting us study how resource availability, market competition, and productive capability shape agent economies.

provides a controlled environment for studying how economic conditions shape agent behavior and economic dynamics.

We instantiate TokenLivelihood as a simulated task market and evaluate agents powered by LLMs with diverse capability and scale across different economic regimes. Our experiments reveal how fundamental economic forces shape agent behavior and system-level outcomes. We find that resource availability influences agents' risk preferences: abundant capital encourages more aggressive investment decisions, which can increase short-term activity but may ultimately hinder long-term wealth accumulation. Market competition further reshapes the balance between efficiency and capability, making stronger models increasingly advantageous under intense competition while reducing the overall profitability of the economy. In contrast, improvements in productive capability generate broad economic gains by enabling agents to complete more valuable tasks and increase wealth accumulation across different market conditions.

Our contributions can be summarized as follows:

- We introduce **TokenLivelihood**, a verifiable economic sandbox that models LLM agents as heterogeneous producers in a task market, enabling controlled studies of how resource availability, market competition, and productive capability shape agent economies.
- Through extensive experiments, we show that abundant capital encourages riskier decisions that may reduce long-term wealth accumulation, while intensified competition reshapes model advantages and lowers overall economic surplus.
- We demonstrate the importance of productivity growth, showing that more capable agents can drive broader wealth creation across diverse market environments.

2 Related Work

2.1 Budget-Aware Inference

Recent work has studied how to improve LLM reasoning efficiency under explicit computational budget constraints. One line of research makes models aware of their remaining budget during inference. Liu et al. (2025) introduce a budget tracker that provides agents with budget information during generation, while Wen et al. (2025) train models to follow budget constraints through rein-

forcement learning. Another line focuses on adaptive budget allocation across reasoning processes. Li et al. (2026) allocate computation based on the estimated value of further exploration, while Wang et al. (2025) and Zhai et al. (2026) study adaptive allocation across subtasks or decision steps. Beyond budget optimization, Lin et al. (2026) benchmark agents' ability to predict and manage their own token consumption during execution.

Despite these advances, existing methods generally view computation budget as an externally provided resource within a single task episode. Their objective is to maximize task performance under a fixed budget, without modeling how agents generate computational resources, accumulate capital, and allocate resources across future opportunities. In contrast, TokenLivelihood treats computation as an economic asset that agents must earn, spend, and manage over their lifetime, extending budget-aware inference from static resource allocation to dynamic economic decision-making.

2.2 Agent-based Social Simulation

Recent studies have explored simulated societies and economic environments with LLM-based agents. Generative Agents (Park et al., 2023) and Agentopia (Wang et al., 2026) study long-term social interactions and emergent behaviors, while EconAgent (Li et al., 2023), AI Agent Economics (Zhang & Shang, 2026), and Energy Society (Hansen et al., 2026) investigate economic interactions, resource management, and survival behaviors among autonomous agents. These works demonstrate the potential of LLM agents to participate in complex social and economic systems.

However, existing environments primarily focus on whether such behaviors can emerge, rather than systematically studying how fundamental economic conditions shape agent decisions and outcomes. TokenLivelihood introduces an economic sandbox centered on three key factors in economic systems: resource scarcity, competition and development of productivity. By treating computation as both production cost and economic capital, TokenLivelihood enables controlled studies of how economic conditions shape agent strategies and long-term economic dynamics.

3 Preliminaries

We view each agent as an economic decision-maker that allocates limited resources across a sequence of production opportunities. At each step, the agent makes decisions that affect its resource consumption and future ability to operate. We introduce the basic economic concepts underlying our formulation below.

Production and Cost In economics, a production function characterizes how productive inputs are transformed into output (Cobb & Douglas, 1928). We model the outcome of completing task i by

$$Q_{ij} = f(P_j, T_{ij}), \tag{1}$$

where P_j denotes agent j's productive capacity and T_{ij} the computational effort allocated to task i. Producing an outcome consumes computational resources and incurs a cost C_{ij}, which captures the resources expended by agent j to complete task i.

Pricing and Profit A task or service generates revenue according to its market price V_i. Under cost-based pricing, prices may be anchored to the cost of producing the corresponding output and set with a markup over that cost (Hall & Hitch, 1939). The resulting profit of agent j on task i is

$$\Pi_{ij} = V_i Q_{ij} - C_{ij}, \tag{2}$$

where $V_i Q_{ij}$ denotes the realized revenue. Thus, an agent's economic performance depends jointly on the value it generates and the resources it consumes.

Capital Dynamics and Objective Let $B_{j,t}$ denote the capital available to agent j at step t. Capital increases through realized revenue and decreases through production expenditure, giving the general capital dynamics

$$B_{j,t+1} = B_{j,t} - C_{ij} + V_i Q_{ij}. \tag{3}$$

An economic agent therefore faces an intertemporal resource-allocation problem: spending resources on one task increases the opportunity to obtain its associated revenue, but reduces the resources available for subsequent opportunities. We formulate the agent's objective as maximizing terminal capital while remaining solvent throughout its lifetime:

$$\max_{\pi_j} \mathbb{E}[B_{j,H}] \quad \text{s.t.} \quad B_{j,t} \geq 0, \ \forall t, \tag{4}$$

where π_j denotes the agent's decision policy.

Scarcity and Opportunity Cost Scarcity is a fundamental premise of economic decision-making: productive resources are limited and have alternative uses (Robbins, 1984). Allocating resources to one opportunity therefore incurs an opportunity cost, corresponding to the value of the alternatives that cannot be pursued with the same resources. In a task market, this means that an agent must balance the expected return from the current opportunity against the future opportunities foregone by committing its limited resources.

4 TOKENLIVELIHOOD

Building on Section 3, TokenLivelihood instantiates a simulated task market around a single principle: economic capital and computational budget are the same quantity, measured in the same unit, coupling the agent, task, and market design.

4.1 AGENT DESIGN

Goal The agent is assigned an explicit economic objective rather than a purely task-oriented one. At each step, it observes its current capital $B_{j,t}$ and the cumulative history of income and expenditure, and aims to maximize its terminal capital while satisfying the solvency constraint $B_{j,t} \geq 0$ throughout the episode (Section 3). This formulation makes the agent responsible for generating the resources needed to sustain its own operation, distinguishing our setting from budget-aware inference methods (Liu et al., 2025; Lin et al., 2026).

Memory Each agent maintains a short-term memory that summarizes its recent interaction history. The memory records which tasks the agent selected or declined, the resulting outcome and reward, and the corresponding environment feedback. Following MemGPT (Packer et al., 2023), we maintain this memory within a fixed FIFO context window: once the context exceeds 20k tokens, earlier entries are progressively removed using a sliding-window mechanism.

Action Space At each step, an agent can select an available task, increase the compute budget of an ongoing task, or abandon an ongoing task.

- **Select Task:** accept an available task and allocate an initial compute budget to it; alternatively, the agent may skip the current step without selecting any task.
- **Increase Budget:** allocate additional compute budget to an ongoing task.
- **Abandon Task:** terminate an ongoing task before completion, forfeiting the capital already spent on it.

For a selected task, the allocated compute budget determines the number of tokens τ available to the underlying model. These actions jointly determine the agent's *acceptance policy*, governing which tasks to pursue, how much to invest, and when to stop.

4.2 ECONOMIC MECHANISM

TokenLivelihood instantiates the economic concepts in Section 3 by using computation as the common economic unit. Agent capital and computational expenditure are both denominated in FLOPs, making computation a direct economic cost: spending on one task reduces resources for subsequent decisions, while successful completion replenishes capital through rewards.

Production Cost For agent j performing task i, we define the production cost as

$$C_{ij} = P_j T_{ij}, \tag{5}$$

where P_j is the number of activated model parameters and T_{ij} is the number of generated tokens. Thus, production cost depends on both model size and computation, making production efficiency an endogenous source of economic advantage.

Pricing and Reward Rather than assigning fixed rewards, TokenLivelihood prices each task based on its reference production cost:

$$V_i = \kappa C_i^{\text{ref}}(1 + \epsilon_i), \tag{6}$$

where C_i^{ref} is the reference completion cost, $\kappa > 1$ is a shared markup, and ϵ_i captures market fluctuations. The realized price is public and identical across agents, while actual cost C_{ij} varies with each agent's model and policy.

Given a binary task outcome $Q_{ij} \in \{0, 1\}$, the realized profit is

$$\Pi_{ij} = V_i Q_{ij} - C_{ij}, \tag{7}$$

and capital evolves as

$$B_{j,t+1} = B_{j,t} + \Pi_{ij} = B_{j,t} - C_{ij} + V_i Q_{ij}. \tag{8}$$

Thus, task outcomes and computational costs directly determine the resources available for subsequent decisions.

4.3 Market

4.3.1 Task Pool

Scarcity is a fundamental premise of economic decision-making: resources are limited and have alternative uses, requiring agents to choose among competing opportunities (Robbins, 1984). Following this principle, TokenLivelihood provides a finite task pool rather than an unlimited stream. Tasks arrive in batches and are maintained in a circular queue, from which agents select tasks at each step. Once a task is successfully completed, it's permanently removed and becomes unavailable to all agents. Failed or abandoned tasks are returned to the back of the queue for future attempts.

The current batch is replaced by a fresh one when either 5 consecutive rounds pass without a completion or at least 40% of its tasks have been completed. This prevents the task pool from stalling while maintaining a bounded supply of available tasks. Each batch contains 20 tasks in our experiments.

4.3.2 Market Structure: From Monopoly to Competition

Classical market theory distinguishes market structures by the number of producers competing for a given demand. In **monopoly**, a single producer has exclusive access to the market, whereas under **competition**, multiple producers compete for the same demand (Hall & Hitch, 1939). TokenLivelihood instantiates this distinction: a monopoly gives one agent exclusive access, while competition allows multiple agents to compete for the same task opportunities. We consider three settings:

1. **Monopoly.** A single agent has exclusive access to the entire task pool and does not compete with other agents. Its performance is determined solely by its own production decisions.
2. **Competition – Synchronous.** Multi-agents share the task pool under a round-based protocol. In each round, agents simultaneously select tasks and commit compute budgets. If the budget is insufficient, an agent may increase it or abandon the task. The next round begins only after all agents finish their actions, synchronizing competition for task opportunities.
3. **Competition – Asynchronous.** Multi-agents share the task pool but proceed independently. Agents can accept new tasks immediately after completing current ones, so faster agents access new opportunities more frequently and competition occurs continuously.

4.3.3 CONFLICT RESOLUTION

In competitive settings, multiple agents may select the same task, creating allocation conflicts. TokenLivelihood resolves these conflicts by having each agent submit its top-m preferred tasks. If multiple agents request the same task, it is assigned to the agent with the higher historical completion rate, while others fall back to their next-ranked choices. This makes allocation depend on both agent track records and preferences rather than decision order.

4.4 EVALUATION METRICS

We evaluate economic outcomes of TokenLivelihood using three metrics: *revenue*, *expenditure*, and *net profit*. We report each metric for both individual agents and the economy as a whole.

Revenue Revenue measures the total economic return an agent earns from successfully completed tasks. For agent j, its cumulative revenue is

$$R_j = \sum_{i \in \mathcal{I}j} V_i Q_{ij}, \tag{9}$$

where $\mathcal{I}_j$ denotes the tasks on which agent j has spent compute, and $Q_{ij} = 1$ only when task i is successfully completed by agent j.

Expenditure Expenditure measures the total compute cost an agent incurs in carrying out tasks, regardless of their success or failure. For agent j,

$$E_j = \sum_{i \in \mathcal{I}j} Cij. \tag{10}$$

Profit Profit measures the change in an agent's capital resulting from its economic activity. Over the simulation, it is equivalently given by cumulative revenue minus cumulative expenditure:

$$\Pi_j = B_{j,t} - B_{j,0} = R_j - E_j. \tag{11}$$

At the economic-system level, the aggregate performance is obtained by summing the corresponding quantities across all agents:

$$R_{\text{sys}} = \sum_j R_j, \qquad E_{\text{sys}} = \sum_j E_j, \qquad \Pi_{\text{sys}} = \sum_j \Pi_j = R_{\text{sys}} - E_{\text{sys}}. \tag{12}$$

5 EXPERIMENT

5.1 INITIAL CAPITAL AND PRICE SETTING

Initial capital. Each agent begins with initial capital $B_{j,0}$ drawn from three levels. We use 30k, 90k, and 300k in the monopoly setting, and 90k, 300k, and 900k under competition, representing scarce, moderate, and abundant capital regimes. This allows us to examine how capital availability affects agent behavior while holding other market properties fixed.

Price setting We apply the pricing mechanism from Section 4.2 across all market regimes, adjusting the markup κ with the degree of competition. Following classical price theory, stronger competition limits markups above production costs (Hall & Hitch, 1939). We set $\kappa \in \{1.5, 2.0, 3.0\}$ for monopoly and competition–synchronous, and $\kappa \in \{1.1, 1.5, 2.0\}$ for competition–asynchronous. For all regimes, each task receives an independent fluctuation $\epsilon_i \sim \text{Uniform}(-0.1, 0.1)$. We discuss the impact of different reward settings on the experimental results in Appendix B.

5.2 DATASETS AND MODELS

Our task pool contains 258 tasks from three benchmarks: 58 tasks from SWE-bench (Jimenez et al., 2023), and 100 randomly sampled tasks each from WritingBench (Wu et al., 2025) and AutomationBench (Shepard & Salimans, 2026). Together, these benchmarks provide tasks with varying levels

of complexity, allowing us to evaluate agent economic behavior under heterogeneous task demands. We retain each task's reference cost C_i^{ref} and convert all task outcomes to binary success, using the original criteria for SWE-bench and AutomationBench and a score threshold of 0.6 for WritingBench. Successful tasks yield the full reward, while failed tasks yield no reward. We discuss the impact of task sequence in Appendix C.

We instantiate agents with seven model configurations, listed in Table 1. Their activated-parameter counts range from 2B to 49B. The table describes the recorded task-solving performance used by the simulator; economic outcomes additionally depend on task selection, expenditure, and access.

Table 1: Model configurations, activated parameters used for cost accounting, and reported per-domain success rates (%). WritingBench uses the 100-task operating subset, with one frozen outcome per task; it is not an average over 200 calibration tasks.

Model	Active parameters (B)	SWE-bench	WritingBench	AutomationBench
GLM-5.2 (Zeng et al., 2026)	40	77.6	98.0	23.0
GLM-5.1 (Zeng et al., 2026)	40	75.9	97.0	15.0
DeepSeek-v4-pro (Xu et al., 2026)	49	74.1	99.0	25.0
DeepSeek-v4-flash (Xu et al., 2026)	13	74.1	99.0	20.0
Qwen3.5-397b-a17b (Qwen Team, 2026)	17	69.0	99.0	4.0
Qwen3.5-35b-a3b (Qwen Team, 2026)	3	72.4	99.0	6.0
Qwen3.5-2b (Qwen Team, 2026)	2	3.4	90.0	0.0

5.3 Implementation Details

Executing tasks online would entangle economic decisions with task-solving ability and make repeated simulations costly. Following Lin et al. (2026), we therefore use a two-stage offline process. First, each model attempts every task without budget constraints, and we record its outcome and token consumption. Second, the agent makes only the economic decisions defined in Section 4, including task selection, budget allocation, and task abandonment, while the environment retrieves the corresponding outcome and cost from the first stage instead of re-executing the task. This allows us to vary market conditions while keeping the underlying task-solving behavior fixed. We repeat experiments in competition settings three times to ensure reproducibility.

5.4 Main Results

As shown in Table 2, we find that resource abundance affects both agents' decision-making and the wealth of the overall economy. As initial capital increases, both revenue and expenditure generally rise, but expenditure grows faster, resulting in lower profit. Greater capital enables agents to attempt more tasks, increasing revenue, but also encourages them to pursue more difficult tasks whose higher costs outweigh the additional returns. In contrast, increasing task rewards through a higher markup leads to a slight increase in profit, as it raises returns without substantially altering task-selection behavior. We further break down this shift in task-selection behavior by model in Appendix H, examining whether agents preferentially pursue high-reward tasks or favor safer, lower-risk ones. See more results and visualization in Appendix F and G.

This effect is further moderated by market structure. Under monopoly with a high reward markup, the negative relationship between resource abundance and profit becomes much weaker. Without competitive pressure to capture task opportunities, a monopolistic agent can afford to decline highly challenging tasks and complete fewer tasks overall, allowing higher resource availability to translate more directly into profit.

6 Analysis

In this section, we further investigate how fundamental economic conditions shape agent decisions and system-level dynamics from the following research questions: **RQ1**: *How does resource availability affect agents' final decisions and economic outcomes?* **RQ2**: *How does market competition reshape the behavior of the overall economy?* **RQ3**: *How does increased productive capability affect economic outcomes and wealth creation?*

Table 2: Aggregate revenue, expenditure, and net profit (in 10^4 credits) under each market regime, across reward markups κ and initial capital levels. Monopoly results sum seven independent single-agent runs, one per model.

Monopoly									
	$\kappa = 1.5$			$\kappa = 2.0$			$\kappa = 3.0$		
Initial capital	30k	90k	300k	30k	90k	300k	30k	90k	300k
Revenue	11.8	235.9	428.6	99.4	373.5	568.7	57.0	707.4	1272.3
Expenditure	31.3	289.5	559.4	93.3	369.4	560.5	76.8	421.2	790.1
Profit	−19.5	−53.6	−130.8	+6.1	+4.0	+8.2	−19.9	+286.2	+482.3
Competition (synchronous)									
	$\kappa = 1.5$			$\kappa = 2.0$			$\kappa = 3.0$		
Initial capital	90k	300k	900k	90k	300k	900k	90k	300k	900k
Revenue	136.3	281.1	306.1	167.2	334.2	398.2	339.0	630.8	632.7
Expenditure	153.2	420.6	671.6	199.8	404.5	631.4	228.8	701.8	843.7
Profit	−16.9	−139.4	−365.5	−32.6	−70.3	−233.2	+110.3	−71.0	−211.0
Competition (asynchronous)									
	$\kappa = 1.1$			$\kappa = 1.5$			$\kappa = 2.0$		
Initial capital	90k	300k	900k	90k	300k	900k	90k	300k	900k
Revenue	73.8	171.9	123.7	156.0	255.0	257.6	303.7	374.5	371.3
Expenditure	128.0	278.9	268.1	179.1	350.3	469.4	306.5	438.1	458.7
Profit	−54.1	−107.0	−144.4	−23.2	−95.2	−211.8	−2.8	−63.6	−87.4

6.1 RQ1: Resource Availability and Agent Decisions

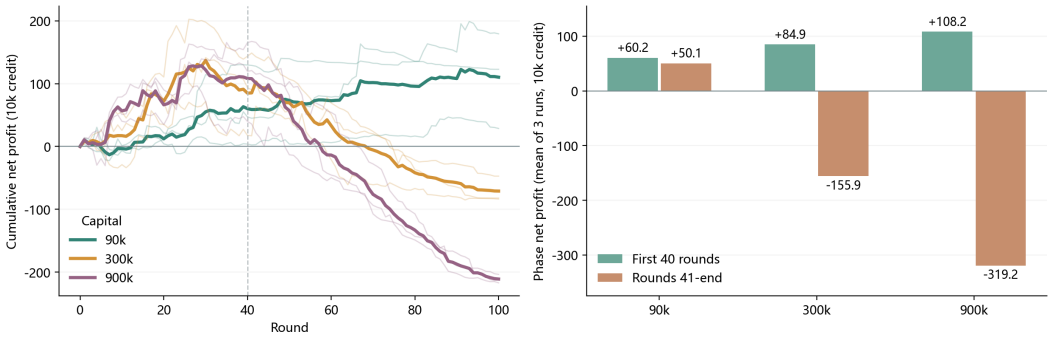


Figure 2: Left: aggregate profit of the economy running process. Right: profit accumulated during different stages across different initial capital levels (synchronous-competition setting).

Figure 2 illustrates how task availability shapes the economy's profitability over time (more in Appendix E). In the early rounds, abundant easy tasks allow agents to consistently find profitable opportunities, leading to steady profit accumulation. After round 40, however, the supply of easy tasks is exhausted, leaving harder and less profitable tasks and causing aggregate profit to decline. This transition becomes more pronounced with higher initial capital, as larger budgets encourage agents to pursue more tasks and deplete profitable opportunities faster. We further examine whether repeated attempts at previously failed tasks contribute to this loss in Appendix D.

6.2 RQ2: Competition and Economic Pressure

As shown in Figure 3, moving from monopoly to competition modestly affects economy-wide revenue but causes a much sharper drop in profits. This reflects heightened profit pressure on agents, who incur more attempt costs as rivals pre-empt available tasks.

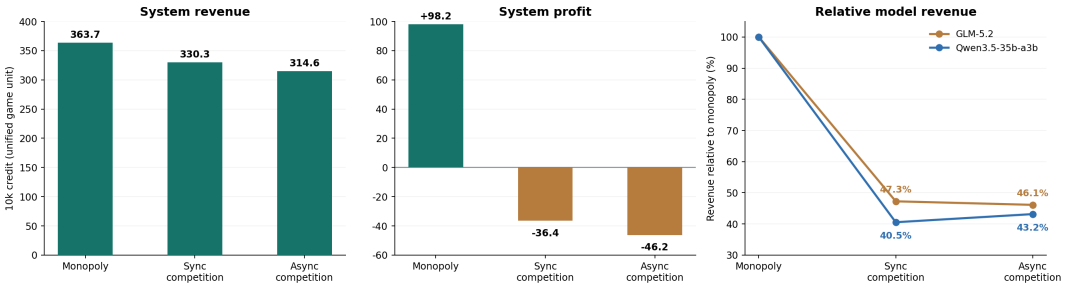


Figure 3: Effect of market structure, from monopoly to competition, on the economy's aggregate revenue and profit, and on individual models' revenue. Competition leaves revenue largely unchanged but sharply reduces profit, and smaller models such as Qwen3.5-35B-A3B lose revenue more severely than larger models as competition intensifies.

Disaggregating by model size, smaller models like Qwen3.5-35B-A3B suffer a significantly larger revenue drop under competition than larger ones. This gap stems from two compounding factors: first, more capable competitors pre-empt tasks across all competitive regimes; second, faster completion rates by larger models in asynchronous regimes allow them to capture strictly more tasks within the same timeframe. Consequently, competition disproportionately harms small models. Monopoly, by excluding rivals, thus acts as a strongly protectionist policy for low-capability models, whereas high-capability models remain resilient under competition.

6.3 RQ3: Productivity and Economic Growth

Table 3: Final wealth of each agent under the *Vanilla* setting and the *Productivity* setting (initial capital of 90k and reward coefficient of 3.0). The last row reports the wealth change induced by the entry of stronger agents.

	Qwen3.5-2b	Qwen3.5-35b-a3b	GLM-5.1	Deepseek-v4-flash	Qwen3.5-397b-a17b	GLM-5.2	Deepseek-v4-pro	**Sum.**
Vanilla	58.1k	846.7k	80.8k	2.6k	0.0k	—	—	988.3k
Productivity	69.9k	896.7k	236.5k	0.0k	0.0k	642.7k	0	1845.7k
	+11.8k	+49.9k	+155.7k	−2.6k	0.0k	—	—	+857.4k

To evaluate the economic impact of productivity gains, we compare two settings over 40 rounds: a *Vanilla* setting with five weaker agents, and a *Productivity* setting where two high-capability models (GLM-5.2 and Deepseek-v4-pro) join the original five agents.

As shown in Table 3, high-productivity entrants expand total economic wealth by 86.8%[2]. Crucially, most of the individual agents gain wealth, demonstrating broad-based benefits. High-capability entrants absorb complex tasks previously intractable for weaker agents, freeing the latter to focus on easier tasks without wasting capital. This dynamic encourages market specialization based on core capabilities, fostering shared growth rather than a zero-sum redistribution of wealth.

7 Conclusion

We present **TokenLivelihood**, a verifiable economic sandbox where computation serves as both production cost and economic capital, allowing LLM agents to act as self-sustaining producers that consume resources, earn rewards, and fund continued operation through accumulated capital. Our experiments show that abundant capital encourages riskier decisions that may hinder long-term wealth accumulation, while intensified competition favors stronger models but reduces overall economic profitability. In contrast, improvements in productive capability generate broad wealth gains by enabling agents to complete more valuable tasks across different market conditions.

[2]Excluding the 180k initial capital from the two new entrants, net wealth grows by 68.5%.

AI USE STATEMENT

In this work, we used generative AI tools to edit the paper for improved language and readability, and to assist with the creation and modification of scientific figures. We have not used generative AI tools for any of the required-disclosure research tasks, including developing theoretical models or conceptual frameworks, formulating mathematical claims, designing methodologies or experiments, implementing methods, interpreting results, or other research tasks listed in the ICLR 2027 AI Policy that require disclosure. We have reviewed all AI-assisted work and verified its correctness and consistency with the authors' intended content. We take full responsibility for the final content of this work, including all text, claims, and figures produced with the aid of generative AI.

REFERENCES

Charles W. Cobb and Paul H. Douglas. A theory of production. *American Economic Review*, 18(1): 139–165, 1928.

Robert L. Hall and Charles J. Hitch. Price theory and business behaviour. *Oxford Economic Papers*, (2):12–45, 1939.

Lucas Bergholdt Hansen, Federico Torrielli, Filippo Tonini, and Lukas Galke Poech. The energy society: A simulation environment for studying agent cooperation under survival pressure. *arXiv preprint arXiv:2607.14865*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Nian Li, Chen Gao, Mingyu Li, Yong Li, and Qingmin Liao. Econagent: Large language model-empowered agents for simulating macroeconomic activities. *arXiv preprint arXiv:2310.10436*, 2023.

Yushu Li, Wenlong Deng, Jiajin Li, and Xiaoxiao Li. Spend less, reason better: Budget-aware value tree search for llm agents. *arXiv preprint arXiv:2603.12634*, 2026.

Yuxiang Lin, Zihan Wang, Mengyang Liu, Yuxuan Shan, Longju Bai, Junyao Zhang, Xing Jin, Boshan Chen, Jinyan Su, Xingyao Wang, Jiaxin Pei, and Manling Li. Bagen: Are llm agents budget-aware? *arXiv preprint arXiv:2606.00198*, 2026.

Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool use enables effective agent scaling. *arXiv preprint arXiv:2511.17006*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Zhenting Qi, Huangyuan Su, Ao Qu, Chenyu Wang, Yu Yao, Han Zheng, Kushal Chattopadhyay, Guowei Xu, Zihan Wang, Weirui Ye, Vijay Janapa Reddi, Ju Li, Paul Pu Liang, Himabindu Lakkaraju, Sham Kakade, and Yilun Du. Economy of minds: Emerging multi-agent intelligence with economic interactions. *arXiv preprint arXiv:2606.02859*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

L Robbins. An essay on the nature and significance of economic science. *New York University Press,*, 1984.

Daniel Shepard and Robin Salimans. Automationbench, 2026. URL https://arxiv.org/abs/2604.18934.

Fali Wang, Hui Liu, Zhenwei Dai, Jingying Zeng, Zhiwei Zhang, Zongyu Wu, Chen Luo, Zhen Li, Xianfeng Tang, Qi He, and Suhang Wang. Agenttts: Large language model agent for test-time compute-optimal scaling strategy in complex tasks. *arXiv preprint arXiv:2508.00890*, 2025.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of computer science*, 18(6):186345, 2024.

Xintao Wang, Sirui Zheng, Hongqiu Wu, Weiyuan Li, Jen-tse Huang, Minghao Zhu, Can Zu, Qi Deng, Jiawei Wang, Qianyu He, Heng Wang, Xiaojian Wu, and Yunzhe Tao. Agentopia: Long-term life simulation and learning in agent societies. *arXiv preprint arXiv:2606.07513*, 2026.

Hao Wen, Xinrui Wu, Yi Sun, Feifei Zhang, Liye Chen, Jie Wang, Yunxin Liu, Yunhao Liu, Ya-Qin Zhang, and Yuanchun Li. Budgetthinker: Empowering budget-aware llm reasoning with control tokens. *arXiv preprint arXiv:2508.17196*, 2025.

Yuning Wu, Jiahao Mei, Ming Yan, Chenliang Li, Shaopeng Lai, Yuran Ren, Zijia Wang, Ji Zhang, Mengyue Wu, Qin Jin, and Fei Huang. Writingbench: A comprehensive benchmark for generative writing. *arXiv preprint arXiv:2503.05244*, 2025.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China information sciences*, 68(2):121101, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Zhiyuan Zhai, Bingcong Li, Bingnan Xiao, Ming Li, and Xin Wang. Adaptive test-time compute allocation for reasoning llms via constrained policy optimization. *arXiv preprint arXiv:2604.14853*, 2026.

Lingyun Zhang and Shang Shang. Ai agent economics: Can autonomous economic behavior emerge among ai agents under minimal external conditions? *arXiv preprint arXiv:2608.03076*, 2026.

A Datasets

SWE-bench (Jimenez et al., 2023) presents an agent with a GitHub issue and the corresponding Python repository. To resolve the issue, the agent must locate the relevant implementation, modify the code, and produce a patch. Evaluation applies the patch and runs repository tests to check the repair. These tasks require understanding an existing project and coordinating code inspection, editing, and testing. Following Lin et al. (2026), we select the same subset.

AutomationBench (Shepard & Salimans, 2026) gives an agent a business request and an initialized environment containing records in simulated applications such as customer databases, calendars, inboxes, and support platforms. The agent uses tools to retrieve information and update application records while following the task's business rules. Final application states are checked programmatically. A task succeeds only when all required checks pass. The benchmark covers sales, marketing, operations, support, finance, and human resources.

WritingBench (Wu et al., 2025) asks a model to produce text that satisfies content requirements and constraints on style, format, or length, sometimes using supplied reference material. Its tasks span six broad domains and 100 subdomains. Responses are assessed against five task-specific criteria reflecting the requirements of each prompt.

B Effect of Pricing Mechanism

Table 4: Reward markup κ assigned to each task difficulty tier, measured by the number of agents n that complete a task, under the Difficulty Incentive setting and the Vanilla setting.

n	> 4	$3 \sim 4$	$1 \sim 2$	0
Difficulty Incentive	1.5	2.0	3.0	4.0
Vanilla	1.5	1.5	1.5	1.5

Beyond uniform pricing, we experiment with an alternative pricing strategy: easy tasks, which most agents can complete, receive a lower reward markup, while hard tasks, which only a small fraction of agents can complete, receive a higher reward markup. This *difficulty incentive* setting rewards task difficulty directly through the price mechanism, encouraging agents to attempt more challenging tasks. Table 4 compares the reward markup κ assigned to each difficulty tier, measured by the number of agents n that complete a task, under the Difficulty Incentive setting and the Vanilla setting.

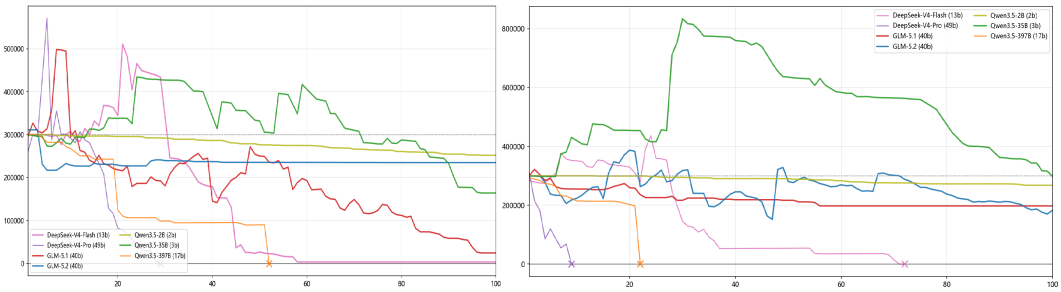


Figure 4: Wealth trajectories under the Difficulty Incentive setting (left panel) and the Vanilla setting (right panel). Rewarding difficult tasks more generously induces agents to attempt harder tasks, but the resulting increase in failed attempts outweighs the additional reward, leading to a net loss.

Figure 4 shows the resulting wealth trajectories. Rewarding harder tasks more generously changes some agents' decisions, leading them to attempt more difficult tasks that they would otherwise avoid under uniform pricing. However, because these harder tasks also carry a higher failure rate, this shift increases agents' expenditure on unsuccessful attempts, ultimately resulting in a net loss rather than the intended gain from pursuing higher-value tasks.

C Effect of Task Sequence

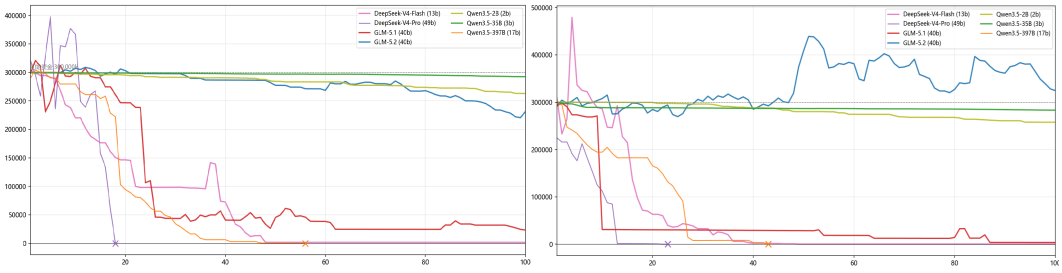


Figure 5: Wealth trajectories under two independently randomized task orderings, under the Competition – Synchronous regime with an initial capital of 300k and a tiered reward markup. The two orderings produce different intermediate trajectories but converge to similar final outcomes.

To examine how the order in which tasks arrive affects our results, we run the Competition – Synchronous regime with an initial capital of 300k and a tiered reward markup: tasks completed by

more than four agents receive a markup of 1.0, tasks completed by three to four agents receive 1.5, tasks completed by one to two agents receive 2.0, and tasks that no agent completes receive 3.0. Under this setting, we compare two independently randomized task orderings.

Figure 5 shows the resulting wealth trajectories. The two task orderings can shift the intermediate trajectory of wealth accumulation, since agents encounter easy and hard tasks in a different sequence. However, because the task pool cycles until every task is eventually either completed or abandoned by some agent, this reordering has only a limited effect on the final outcome.

D Effect of Retry Restriction

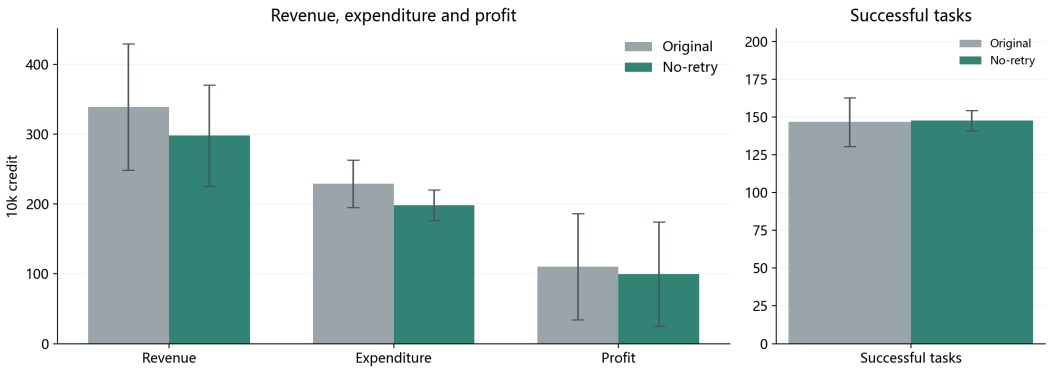


Figure 6: Comparison of the historical baseline and the no-retry setting under synchronous competition with an initial capital of 90k per agent and a reward markup of 3.0. Bars show means over three runs per condition, and error bars indicate sample standard deviations. Successful-task counts include only distinct tasks successfully completed within each run.

To examine the effect of repeated failed attempts on agent strategies, we prevent a model from retrying a task after it has completed an execution attempt and received a failed outcome. Tasks abandoned because of insufficient budget remain eligible for later attempts, and other models may still attempt the same task. We compare this setting with the historical baseline under synchronous competition with an initial capital of 90k per agent and a reward markup of 3.0, using three runs per condition.

Figure 6 compares the aggregate outcomes. The no-retry setting shows lower mean revenue, expenditure, and net profit, while the mean number of successful tasks is similar. With only three runs per condition and substantial variation, this comparison does not establish a consistent net-profit advantage for either setting. Since execution outcomes are frozen for each model–task pair, repeating a fully failed execution does not make the same model succeed on that task.

E Task Difficulty Selection Over Time

To further examine how agents' task selection evolves, we analyze the difficulty distribution of the tasks agents choose to attempt at different stages of the simulation, under the competitive regime with a reward markup of 3.0.

Figure 7 shows that the share of difficult tasks attempted rises markedly after round 40. On one hand, this indicates that current agents can identify task difficulty to some extent: in the early stage, agents preferentially select easier tasks to accumulate resources, and only turn to harder tasks once the pool of easy tasks has been depleted. On the other hand, this same shift toward harder tasks explains the decline in capital observed after round 40, since these later attempts are more likely to fail and consume budget without a corresponding reward.

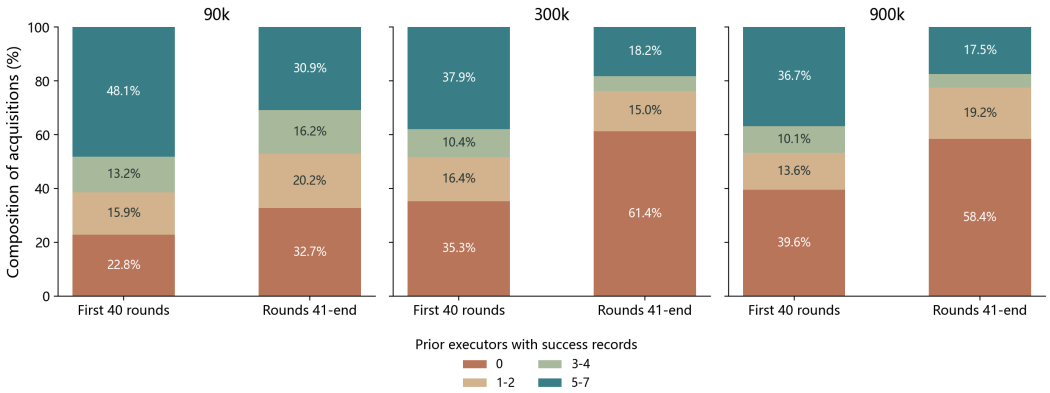


Figure 7: Distribution of task difficulty among the tasks agents choose to attempt, at different stages of the simulation, under synchronous competition with a reward markup of 3.0. The share of difficult tasks attempted rises markedly after round 40.



Figure 8: Share of attempted tasks drawn from WritingBench and AutomationBench as initial capital increases. As initial capital grows, the share of WritingBench tasks decreases while the share of AutomationBench tasks increases.

F Resource Abundance and Task Selection

We further examine how the abundance of initial resources relates to the difficulty of the tasks agents choose to attempt, by tracking the share of attempted tasks drawn from WritingBench and AutomationBench.

Figure 8 shows that as initial capital increases, the share of attempted tasks drawn from WritingBench steadily decreases, while the share drawn from AutomationBench increases. Since WritingBench tasks are considerably easier and cheaper to complete than AutomationBench tasks, this shift indicates that greater initial capital leads agents to attempt more difficult tasks, consistent with the more aggressive, risk-tolerant behavior induced by resource abundance observed elsewhere in our experiments.

G VISUALIZATION AND ERROR ANALYSIS OF MAIN RESULTS

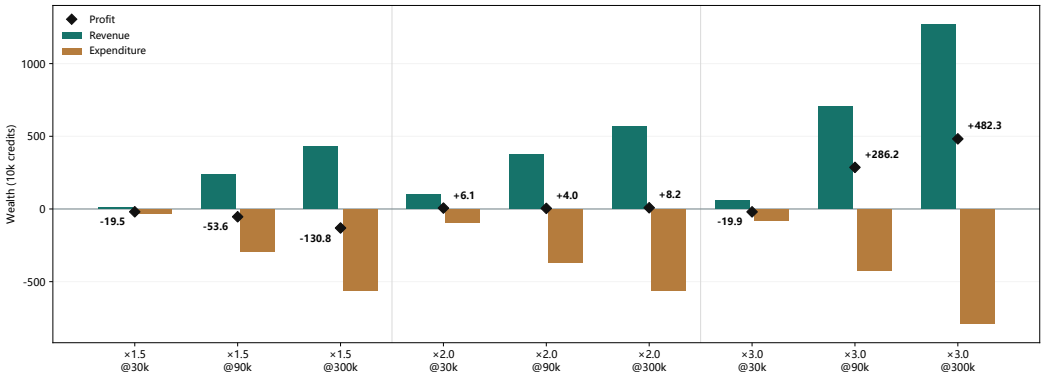


Figure 9: System revenue, expenditure, and net profit under monopoly. Each configuration sums seven independent single-agent runs, one per model, using the same selected runs as Table 2. Revenue is plotted upward, expenditure downward, and black diamonds indicate net profit. Values are in 10^4 credits. No error bars are shown because each model has one included run per configuration.

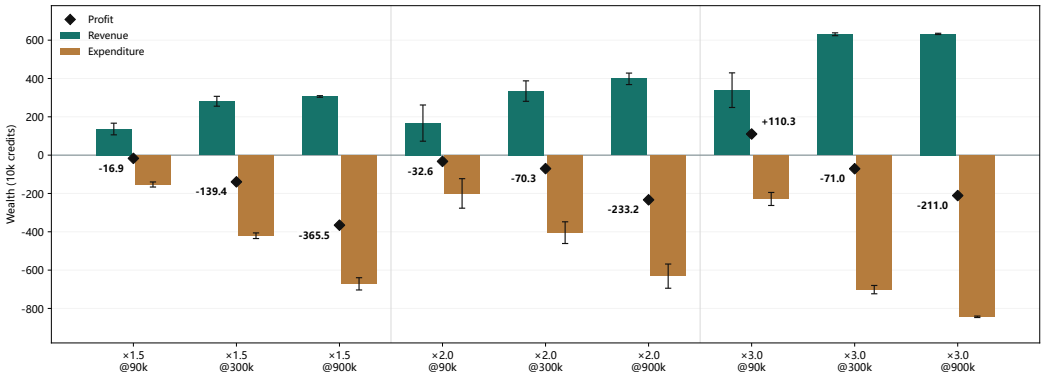


Figure 10: System revenue, expenditure, and net profit under synchronous competition. Bars show mean revenue upward and mean expenditure downward, and black diamonds show mean net profit. Error bars indicate sample standard deviations of revenue and expenditure across eligible runs. Each configuration has three runs, except $\kappa = 1.5$ and initial capital of 300k, which has two eligible runs. Values are in 10^4 credits.

H REWARD PREFERENCE AND TASK PROFITABILITY ACROSS MODELS

We analyze validated first-choice requests under synchronous competition with initial capital $B_0 = 300\text{k}$ credits per agent and reward markup $\kappa = 3.0$ (Table 5). *Reward percentile* is the requested task's percentile rank by posted reward among tasks visible in the same round. *Profitable choices* is the fraction of first choices for which payment under the model's frozen execution record exceeds its full execution cost, without imposing a cash-availability constraint. Both metrics are computed within each run and then averaged over three runs.

DeepSeek-V4-Pro exhibits the strongest preference for high rewards, followed by Qwen3.5-35B-A3B, Qwen3.5-2B, and DeepSeek-V4-Flash, yet most of their first choices are unprofitable for themselves. GLM-5.1 selects lower-reward tasks with the highest profitable-choice rate, while GLM-5.2 and Qwen3.5-397B-A17B show more moderate reward preferences and intermediate profitable-choice rates. High posted rewards therefore do not necessarily imply profitability for the selecting model. These descriptive patterns also reflect execution capability and available opportunities, so they do not establish superior risk assessment or guarantee higher realized profit.

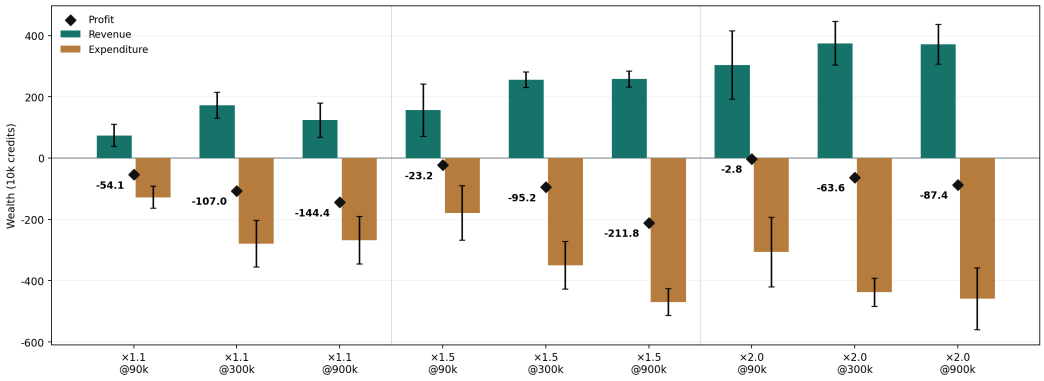


Figure 11: System revenue, expenditure, and net profit under asynchronous competition. Bars show mean revenue upward and mean expenditure downward, and black diamonds show mean net profit across three runs per configuration. Error bars indicate standard deviations of revenue and expenditure across the three runs, computed with divisor $n = 3$. Values are in 10^4 credits.

Table 5: Reward preference and profitability of first-choice tasks under synchronous competition at $B_0 = 300$k and $\kappa = 3.0$. Entries are means over three runs.

Model	Reward percentile	Profitable choices (%)
DeepSeek-V4-Flash	75.0	23.7
DeepSeek-V4-Pro	88.5	33.4
GLM-5.1	34.5	89.5
GLM-5.2	56.2	45.4
Qwen3.5-2B	77.2	8.4
Qwen3.5-35B-A3B	81.5	17.8
Qwen3.5-397B-A17B	58.1	43.4

I CAPITAL TRAJECTORIES IN THE MIXED TASK POOL

Figures 12–14 show capital trajectories for the 258-task mixed pool under monopoly, synchronous competition, and asynchronous competition. Each figure contains nine panels, with pricing multipliers κ increasing from top to bottom and initial capital B_0 increasing from left to right. Colors and line styles identify the same seven models throughout. For each competitive configuration, we use the first recorded run. The curves are individual runs with no averaging across repetitions. Under monopoly, each model instead operates in its own independent copy of the task pool.

Balances are converted to the common credit scale by multiplying each logged token-k balance by its model's recorded parameter multiplier, and the vertical axes display units of 10^4 credits. A gray dashed line marks initial capital, and colored dots mark the last recorded observation. All recorded events are retained without smoothing or extending trajectories beyond their endpoints. The horizontal axis is the recorded round for monopoly and synchronous competition, and the logged simulation clock for asynchronous competition.

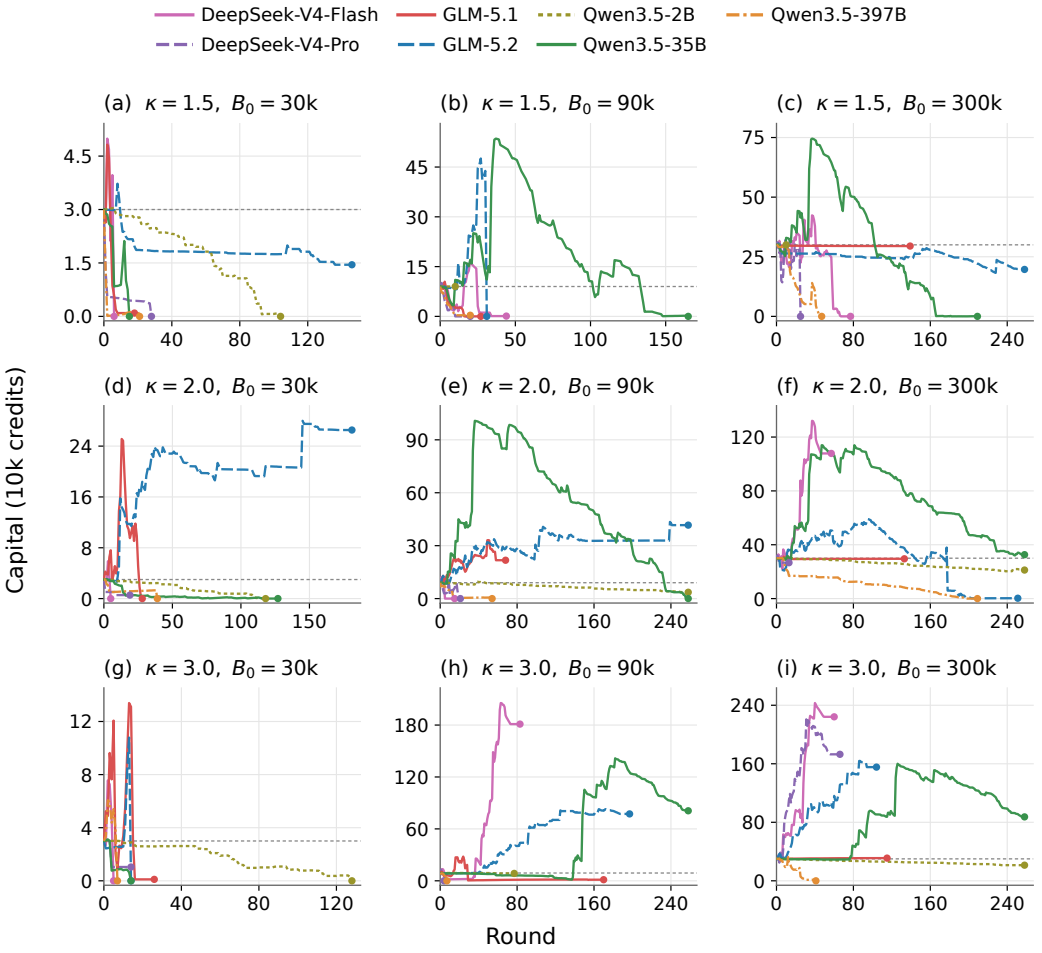


Figure 12: **Mixed-pool capital trajectories under monopoly.** Rows show $\kappa = 1.5$, 2.0, and 3.0. Columns show $B_0 = 30$k, 90k, and 300k credits. Each panel overlays seven independent single-agent runs, one per model, using the same selected runs as the monopoly results in Table 2. The figure covers 63 runs, with no averaging across repetitions. Curves retain all recorded events and end at their last recorded round, up to 258 rounds. Axis limits follow the observed ranges in each panel.

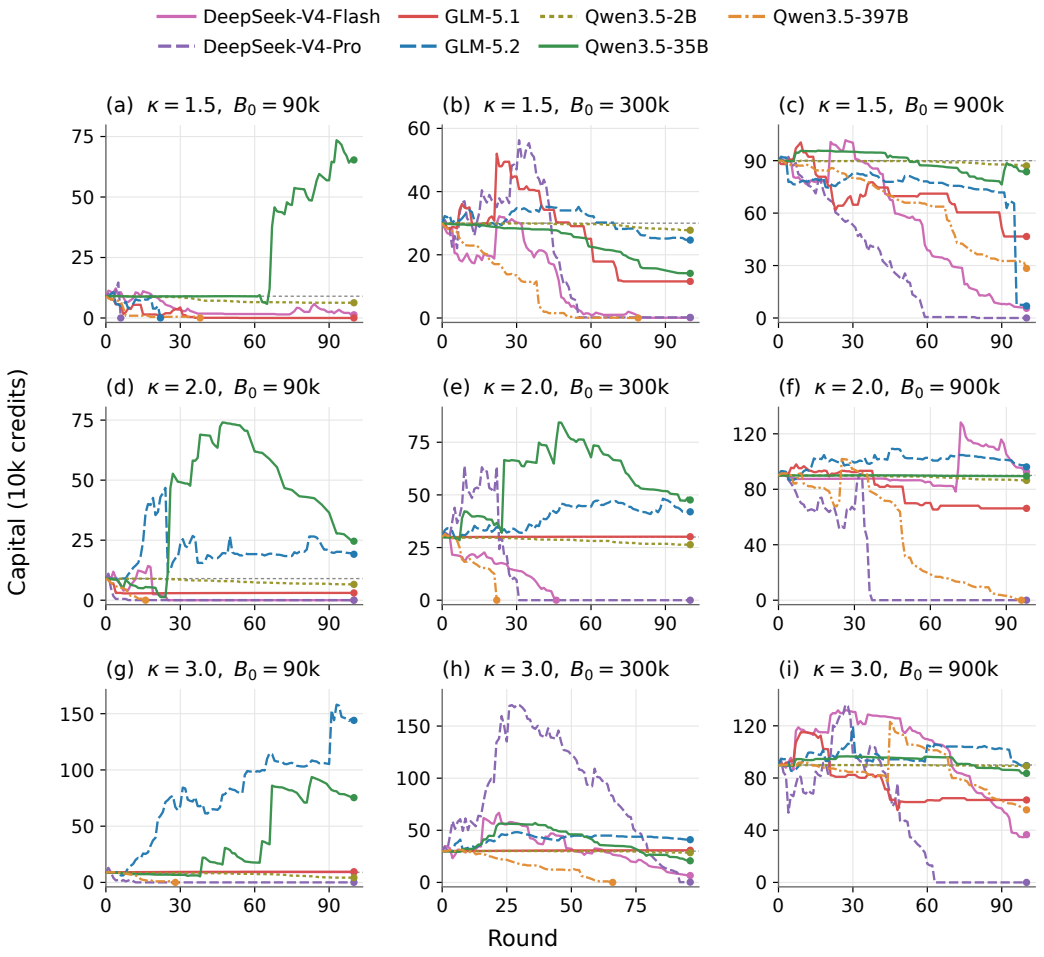


Figure 13: **Mixed-pool capital trajectories under synchronous competition.** Rows show $\kappa = 1.5$, 2.0, and 3.0. Columns show $B_0 = 90$k, 300k, and 900k credits. Each panel shows the first recorded run, in which all seven models compete for one shared task pool under a round-based protocol with a 100-round limit. Curves retain their recorded endpoints. Axis limits follow each panel's observed ranges.

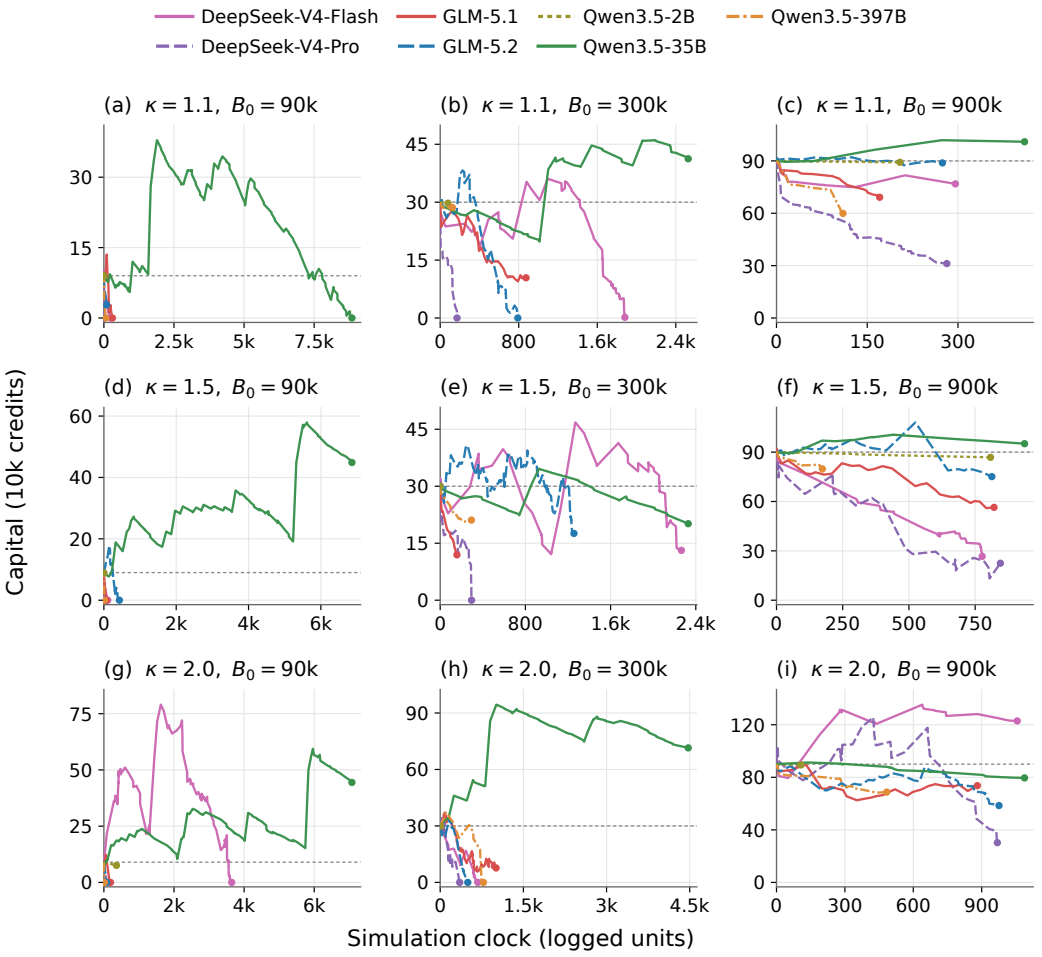


Figure 14: **Mixed-pool capital trajectories under asynchronous competition.** Rows show $\kappa = 1.1$, 1.5, and 2.0. Columns show $B_0 = 90$k, 300k, and 900k credits. Each panel shows the first recorded run with seven models sharing the task pool. The horizontal axis preserves the logged simulation clock in its original units, not measured wall-clock time. Clock ranges differ across panels. No common time horizon is imposed.

J PROMPTS FOR TASK EXECUTION AND MARKET DECISIONS

We present the task-execution prompts for SWE-bench, AutomationBench, and WritingBench. We then show the selection and budget-adjustment prompts used in the three market regimes. Chinese instructions are translated into English.

The headings identifying system and user messages are editorial. Runtime fields, such as <QUERY>, stand for variable inputs. JSON field names, action names, and the prompts' original k-token units are retained. Long lines in the prompt blocks are wrapped for page layout without changing their wording.

J.1 TASK EXECUTION

SWE-bench. Tables 6 and 7 reproduce the initial system and user templates stored in the Mini-SWE-Agent execution records. The {{task}} field is replaced by the issue description. Six models use the tool-call template, while Qwen3.5-2B uses a separate template requiring exactly one command block per response. Both versions are included because their action formats differ.

Table 6: Prompt for SWE-bench: tool-call interface

Initial SWE-bench prompt for DeepSeek-V4-Flash, DeepSeek-V4-Pro, GLM-5.1, GLM-5.2, Qwen3.5-35B, and Qwen3.5-397B. The task description is inserted into the recorded user template.

```
System message
You are a helpful assistant that can interact with a computer
  shell to solve programming tasks.

User message template
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

<instructions>
# Task Instructions

## Overview

You're a software engineer interacting continuously with a
  computer by submitting commands.
You'll be helping implement necessary changes to meet
  requirements in the PR description.
Your task is specifically to make changes to non-test files in
  the current directory in order to fix the issue described in
  the PR description in a way that is general and consistent
  with the codebase.
<IMPORTANT>This is an interactive process where you will think
  and issue AT LEAST ONE command, see the result, then think
  and issue your next command(s).</important>

For each response:

1. Include a THOUGHT section explaining your reasoning and
  what you're trying to accomplish
2. Provide one or more bash tool calls to execute

## Important Boundaries

- MODIFY: Regular source code files in /testbed (this is the
  working directory for all your subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml,
  setup.cfg, etc.)
```

```
## Recommended Workflow

1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

## Command Execution Rules

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include reasoning text explaining what
  you're doing
- Your response MUST include AT LEAST ONE bash tool call. You
  can make MULTIPLE tool calls in a single response when the
  commands are independent (e.g., searching multiple files,
  reading different parts of the codebase).
- Directory or environment variable changes are not
  persistent. Every action is executed in a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE
  cd /path/to/working/dir && ...` or write/load environment
  variables from files

Example of a CORRECT response:
<example_response>
I need to understand the Builder-related code. Let me find
  relevant files and check the project structure.

[Makes multiple bash tool calls: {"command": "ls -la"},
  {"command": "find src -name '*.java' | grep -i builder"},
  {"command": "cat README.md | head -50"}]
</example_response>

## Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require
  user input
- You can use bash commands or invoke any tool that is
  available in the environment
- You can also create new tools or scripts to help you with
  the task
- If a tool isn't available, you can also install it

## Submission

When you've completed your work, you MUST submit your changes
  as a git patch.
Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
```

````
Run 'git diff -- path/to/file1 path/to/file2 > patch.txt'
  listing only the source files you modified.
Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source
  files you modified to fix the issue.
Do not submit file creations or changes to any of the
  following files:

- test and reproduction files
- helper scripts, tests, or tools that you created
- installation, build, packaging, configuration, or setup
  scripts unless they are directly part of the issue you were
  fixing (you can assume that the environment is already set
  up for your client)
- binary or compiled files
</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended
  changes and headers show '--- a/' and '+++ b/' paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not
  submit.

<CRITICAL>
- Creating/viewing the patch and submitting it MUST be
  separate commands (not combined with &&).
- If you modify patch.txt after verifying, you SHOULD verify
  again before submitting.
- You CANNOT continue working (reading, editing, testing) in
  any way on this task after submitting.
</CRITICAL>
</instructions>
````

Table 7: Prompt for SWE-bench: Qwen3.5-2B command-block interface

Initial SWE-bench prompt for Qwen3.5-2B. The recorded template requires one `mswea_bash_command` block in each response.

````
System message
You are a helpful assistant that can interact multiple times
  with a computer shell to solve programming tasks.
Your response must contain exactly ONE bash code block with
  ONE command (or commands connected with && or ||).

Include a THOUGHT section before your command where you
  explain your reasoning process.
Format your response as shown in <format_example>.

<format_example>
THOUGHT: Your reasoning and analysis here

```mswea_bash_command
your_command_here
```
````

```
</format_example>

Failure to follow these rules will cause your response to be
  rejected.

User message template
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

<instructions>
# Task Instructions

## Overview

You're a software engineer interacting continuously with a
  computer by submitting commands.
You'll be helping implement necessary changes to meet
  requirements in the PR description.
Your task is specifically to make changes to non-test files in
  the current directory in order to fix the issue described in
  the PR description in a way that is general and consistent
  with the codebase.

<IMPORTANT>This is an interactive process where you will think
  and issue ONE command, see its result, then think and issue
  your next command.</IMPORTANT>

For each response:

1. Include a THOUGHT section explaining your reasoning and
  what you're trying to accomplish
2. Provide exactly ONE bash command to execute

## Important Boundaries

- MODIFY: Regular source code files in /testbed (this is the
  working directory for all your subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml,
  setup.cfg, etc.)

## Recommended Workflow

1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

## Command Execution Rules

You are operating in an environment where

1. You write a single command
2. The system executes that command in a subshell
3. You see the result
4. You write your next command

Each response should include:

1. A **THOUGHT** section where you explain your reasoning and
  plan
2. A single bash code block with your command
```

````
Format your responses like demonstrated within the
  <format_example> block:

<format_example>
THOUGHT: Here I explain my reasoning process, analysis of the
  current situation,
and what I'm trying to accomplish with the command below.

```mswea_bash_command
your_command_here
```
</format_example>

Commands must be specified in a single bash code block:

```mswea_bash_command
your_command_here
```

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include a THOUGHT section explaining
  your reasoning
- Your response MUST include EXACTLY ONE bash code block
- This bash block MUST contain EXACTLY ONE command (or a set
  of commands connected with && or ||)
- If you include zero or multiple bash blocks, or no command
  at all, YOUR RESPONSE WILL FAIL
- Do NOT try to run multiple independent commands in separate
  blocks in one response
- Directory or environment variable changes are not
  persistent. Every action is executed in a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE
  cd /path/to/working/dir && ...` or write/load environment
  variables from files

Example of a CORRECT response:
<example_response>
THOUGHT: I need to understand the structure of the repository
  first. Let me check what files are in the current directory
  to get a better understanding of the codebase.

```mswea_bash_command
ls -la
```
</example_response>

Example of an INCORRECT response:

<example_response>
THOUGHT: I need to examine the codebase and then look at a
  specific file. I'll run multiple commands to do this.

```mswea_bash_command
ls -la
```

Now I'll read the file:

```mswea_bash_command
cat file.txt
```
</example_response>

If you need to run multiple commands, either:
````

````
1. Combine them in one block using && or ||
```mswea_bash_command
command1 && command2 || echo "Error occurred"
```

2. Wait for the first command to complete, see its output,
  then issue the next command in your following response.

## Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require
  user input
- You can use bash commands or invoke any tool that is
  available in the environment
- You can also create new tools or scripts to help you with
  the task
- If a tool isn't available, you can also install it

## Submission

When you've completed your work, you MUST submit your changes
  as a git patch.
Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
Run `git diff -- path/to/file1 path/to/file2 > patch.txt`
  listing only the source files you modified.
Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source
  files you modified to fix the issue.
Do not submit file creations or changes to any of the
  following files:

- test and reproduction files
- helper scripts, tests, or tools that you created
- installation, build, packaging, configuration, or setup
  scripts unless they are directly part of the issue you were
  fixing (you can assume that the environment is already set
  up for your client)
- binary or compiled files
</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended
  changes and headers show `--- a/` and `+++ b/` paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```mswea_bash_command
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not
  submit.

<CRITICAL>
- Creating/viewing the patch and submitting it MUST be
  separate commands (not combined with &&).
- If you modify patch.txt after verifying, you SHOULD verify
````

```
  again before submitting.
- You CANNOT continue working (reading, editing, testing) in
  any way on this task after submitting.
</CRITICAL>
</instructions>
```

AutomationBench. Table 8 shows the system prompt and the task-specific user-message format. The benchmark supplies the API tool interface separately. In the recorded executions, TURN_BUDGET is 50 for GLM-5.1 and 35 for the other six models. This difference in prompt wording is retained. The example user message is taken from the recorded conversion-tracking task.

Table 8: Prompt for AutomationBench

AutomationBench system and user prompts. Replace <TURN_BUDGET> with 50 for GLM-5.1 or 35 for the other models. The example is the recorded conversion-tracking request.

```
System message template
You are a workflow automation agent. Execute the requested
  tasks using the available tools. Do not ask clarifying
  questions - use the information provided and make reasonable
  assumptions when needed. You have a budget of ~<TURN_BUDGET>
  tool-using turns - favor parallel tool calls and avoid
  duplicate searches. When summarizing your work in messages
  or records, list only items you acted on. Do not name,
  enumerate, or explain items you skipped, excluded, or
  rejected - handle exclusions silently in the action, not
  narratively in the output.

User message template
<TASK_REQUEST>

Example user message
We closed some deals this week - send the conversion data to
  Google Ads (account ID: acct_1) so we can track ROI. Follow
  the standard conversion tracking process.

Today is January 27, 2026.
```

WritingBench. The local generation runners send the benchmark query directly as a user message, without adding a system message. Table 9 shows this format and an English translation of one query. The separate evaluation prompt in Table 10 is sent to the frozen critic once per task-specific criterion, with the query, generated response, and criterion inserted into the template.

Table 9: Prompt for WritingBench generation

WritingBench generation uses a single user message containing the query. The acknowledgments example is translated from the Chinese query at benchmark index 892. The template and example are alternative presentations of the same message slot.

```
User message template
<QUERY>

Example user message, translated from Chinese
Please write an acknowledgments section, giving particular
  attention to the emotional support and companionship my
  family provided while I completed my academic work. Make the
  content specific and detailed. For example, describe the
  help and encouragement my family provided at important
  moments, such as when I encountered difficulties writing my
  thesis, faced intense academic pressure, or had to make
  major decisions. You may follow the standard format for
```

```
  acknowledgments in an academic paper, using this outline:

1. Express gratitude to my family.
2. Thank my supervisor and colleagues, if applicable.
3. Acknowledge the institutions and resources that supported
  the research or work.

Conclude by thanking everyone who helped me. The writing
  should be heartfelt and reflect my personal growth.
```

Table 10: Prompt for WritingBench evaluation

Frozen-critic evaluation prompt for WritingBench. Each call evaluates the response against one criterion and returns a score with its justification.

```
System message
You are an expert evaluator with extensive experience in
  evaluating response of given query.

User message template
Evaluate the Response based on the Query and Criteria provided
  following the Scoring Rules.

** Scoring Rules **

"1-2": "Low score description: Critical deficiencies and major
  issues that prevent adequate functionality.",
"3-4": "Below average score description: Lacking with
  noticeable shortcomings that impact overall effectiveness
  and require improvement.",
"5-6": "Average score description: Adequate but not exemplary,
  Baseline performance that meets essential requirements. Most
  models may achieve this score.",
"7-8": "Above average score description: Strong performance
  characterized by competent execution, though minor
  refinements are needed to achieve excellence.",
"9-10": "High score description: Exceptional performance with
  all aspects optimally addressed, demonstrating superior
  effectiveness and quality without any flaws."

-Provide reasons for each score by indicating specific
  strengths or deficiencies within the Response. Reference
  exact text passages to justify the score, ensuring that each
  reason is concrete and aligns with the criteria requirements
  while highlighting key gaps from the ideal answer.

-Be very STRICT and do not be misled by format or length.
  Ensure that the Response is thoroughly evaluated beyond
  superficial appearances.

-Carefully discern whether the content of the Response is an
  illusion, appearing substantial but actually entirely
  fabricated.

-Sometimes the model may only provide an introduction or an
  overview without truly completing the query, which should be
  considered a failed response. Carefully discern this.

-Scoring Range: Assign an integer score between 1 to 10

** Output format **
(Remove symbols that interfere with JSON parsing, don't use "
  inside reason)
Return the results in the following JSON format, Only output
```

```
  the following JSON format and nothing else:
'''json
{
    "score": an integer score between 1 to 10,
    "reason": "Specific and detailed justification for the
      score using text elements."
}

** Criteria **
'''<CRITERION>'''

** Query **
'''<QUERY>'''

** Response **
'''<RESPONSE>'''

Provide your evaluation based on the criteria restated below:

'''<CRITERION>'''

** Output format **
(Remove symbols that interfere with JSON parsing, don't use "
  inside reason)
Return the results in the following JSON format, Only output
  the following JSON format and nothing else:
'''json
{
    "score": an integer score between 1 to 10,
    "reason": "Specific and detailed justification for the
      score using text elements."
}
'''
```

J.2 Market Decisions

The market prompts guide task selection and budget allocation using the current task list, remaining funds, and recorded completion rate. Task descriptions in selection messages are limited to their first 800 characters, with an ellipsis appended when truncated. The templates below show one task-entry slot, which is repeated for the displayed list. Realized execution costs and outcomes are supplied by the recorded task data during settlement.

Monopoly and synchronous competition. Both regimes use the system prompt in Table 11 and the selection template in Table 12. Each monopoly run has only one active agent with its own task pool. Nevertheless, its prompt still refers to other agents and a shared pool. We retain that wording as recorded rather than introducing a separate monopoly instruction that was not used in these runs.

Table 11: Prompt for monopoly and synchronous competition

English translation of the system prompt shared by the recorded monopoly and synchronous-competition runs. The references to competing agents are present in both sets of logs.

```
System message, translated from Chinese
# Competitive Task Selection Game
You are an agent participating in a **Competitive Task
  Selection Game**. You share the same task pool with several
  other agents and earn returns through multiple rounds of
  task competition.

## Game Rules
Tasks arrive in batches. After each batch arrives, all agents
  with sufficient funds submit their preferences
```

```
  simultaneously.
The system allocates tasks in descending order of the agents'
  **historical task completion rates**. For each agent, the
  system checks the preferences in the submitted order:
* If a preferred task has not been selected by an agent with a
  higher completion rate, it is assigned to this agent.
* If the task has already been assigned to another agent, the
  system checks the next preference.
* If all three preferred tasks have been assigned to other
  agents, this agent receives no task in this round and incurs
  no cost.

Once an agent successfully completes a task, the task is
  permanently removed from the pool.
If an agent attempts a task but fails, the task is returned to
  the pool.
After all tasks in the current batch have been processed, a
  new batch arrives and all agents with sufficient funds are
  asked again whether they wish to participate.

## Tasks
Each task contains the following information:
* `task_index`: the task number. Enter only this number when
  submitting a preference.
* `repo`: the repository name.
* `reward_k`: the reward for successfully completing the task,
  in k tokens.
* `problem_statement`: the task description.
You do not know the actual token cost required to complete a
  task, but you can estimate it.

## Wealth
1. Your current funds are denoted by `wealth`, in k tokens.
2. After successfully completing a task, you receive a reward,
  also in k tokens, which is added to wealth.
3. The actual cost of executing a task is deducted from
  wealth, regardless of whether the task is completed.

## Decision
Your objective is to maximize your final funds (net returns).
  When each batch of tasks arrives, decide:
1. Whether to skip the current batch (stop=true). You do not
  want to undertake any task in this batch and will wait for
  the next batch.
2. Otherwise (stop=false), submit 1-3 preferences in
  descending priority order. Each preference contains
  task_index and budget_k (a budget in k tokens that must be
  <= your remaining funds).

Note: Selecting stop=true only skips the current batch. When a
  new batch arrives, you will still be asked whether you wish
  to participate.
```

Table 12: Task-selection prompt for monopoly and synchronous competition

English translation of the round-based selection message. State values and task entries are inserted at runtime. The response schema permits skipping a batch or submitting up to three ranked task preferences.

```
User message template, translated from Chinese
Current state (round <ROUND>):
  - Your remaining funds: <WEALTH_K>k tokens
  - Tasks remaining in the shared pool: <NUM_AVAILABLE>
  - Your historical completion rate: <COMPLETION_RATE>
```

```
Available tasks:

================================
[task_index=<TASK_INDEX>]
  Repository: <REPO>
  Reward: <REWARD_K>k tokens
  Problem description:
<PROBLEM_STATEMENT>

<REMAINING_TASK_ENTRIES>

================================
Output your decision in JSON format, with no other text:
{
  "reasoning": "<Your reasoning, considering task returns,
    risks, budgets, and other factors>",
  "stop": <true to skip the current batch, false to submit
    preferences>,
  "preferences": [
    {"task_index": <task_index shown in the list above>,
      "budget_k": <budget amount>},
    ... up to 3 preferences, or an empty list when stop=true
  ]
}
```

Asynchronous competition. Tables 13 and 14 reproduce the system instructions and selection-message structure recovered from the asynchronous logs. The state header uses the agent's progress variable `step` in place of a shared round number. The system prompt describes independent selection and settlement, while retaining the same preference fields and budget units.

Table 13: Prompt for asynchronous competition

English translation of the asynchronous-competition system prompt as recorded in the execution logs.

```
System message, translated from Chinese
# Asynchronous Task Selection Game
You are an agent participating in an **Asynchronous Task
  Selection Game**. You share the same task pool with several
  other agents and earn returns by continuously selecting
  tasks.

## Game Rules
Tasks arrive in batches. Unlike the synchronous mode, there
  are **no shared rounds**. Each agent independently repeats
  "select a task -> settle the task -> select the next task".
  As soon as an agent completes or abandons its current task,
  it can immediately select another task without waiting for
  other agents.
The system schedules agents according to their progress rather
  than their round counts. When you are called, the task pool
  you see reflects its current, up-to-date state.
Each time you are called, submit up to 3 preferences in
  descending priority order. The system first checks whether
  your first preferred task is still in the pool. If it is,
  that task is assigned to you. Otherwise, the system checks
  your second and third preferences. If none of the three
  tasks is available, you receive no task and incur no cost on
  this occasion. You will be asked again later.

Once an agent successfully completes a task, it is permanently
  removed from the pool.
If an agent attempts a task but fails, the task is returned to
  the pool. Other agents, including you, may select it later.
```

```
When too few tasks remain in the current batch, the system
  automatically replenishes it with the next batch.

## Tasks
Each task contains the following information:
* `task_index`: the task number. Enter only this number when
  submitting a preference.
* `repo`: the repository name.
* `reward_k`: the reward for successfully completing the task,
  in k tokens.
* `problem_statement`: the task description.
You do not know the actual token cost required to complete a
  task, but you can estimate it.

## Wealth
1. Your current funds are denoted by `wealth`, in k tokens.
2. After successfully completing a task, you receive a reward,
  also in k tokens, which is added to wealth.
3. The actual cost of executing a task is deducted from
  wealth, regardless of whether the task is completed.

## Decision
Your objective is to maximize your final funds (net returns).
  Each time you are asked, decide:
1. Whether to skip the current batch (stop=true). You do not
  want to undertake any task in this batch and will wait to be
  asked again when the next batch arrives.
2. Otherwise (stop=false), submit 1-3 preferences in
  descending priority order. Each preference contains
  task_index and budget_k (a budget in k tokens that must be
  <= your remaining funds).

Note: Selecting stop=true only skips the current batch. When a
  new batch arrives, you will still be asked whether you wish
  to participate.
```

Table 14: Task-selection prompt for asynchronous competition

English translation of the asynchronous selection message. The runtime progress field `<STEP>` replaces the round field used in Table 12.

```
User message template, translated from Chinese
Current state (your progress step=<STEP>):
  - Your remaining funds: <WEALTH_K>k tokens
  - Tasks remaining in the shared pool: <NUM_AVAILABLE>
  - Your historical completion rate: <COMPLETION_RATE>

Available tasks:

================================
[task_index=<TASK_INDEX>]
  Repository: <REPO>
  Reward: <REWARD_K>k tokens
  Problem description:
<PROBLEM_STATEMENT>

<REMAINING_TASK_ENTRIES>

================================
Output your decision in JSON format, with no other text:
{
  "reasoning": "<Your reasoning, considering task returns,
    risks, budgets, and other factors>",
  "stop": <true to skip the current batch, false to submit
```

```
    preferences>,
  "preferences": [
    {"task_index": <task_index shown in the list above>,
      "budget_k": <budget amount>},
    ... up to 3 preferences, or an empty list when stop=true
  ]
}
```

Budget adjustment. The three regimes use the budget-adjustment message in Table 15 when the committed budget is insufficient. The agent may abandon the task or add to the budget. The field `additional_budget_k` specifies the increment, not a replacement total. The displayed remaining funds exclude the budget already committed to the task.

Table 15: Budget-adjustment prompt shared by the three regimes

English translation of the budget-adjustment message. The response specifies either abandonment or an additional budget amount.

```
User message template, translated from Chinese
Insufficient budget!

You allocated <YOUR_BUDGET_K>k tokens to
  task_index=<TASK_INDEX>, but the task has not yet been
  completed.

Your current remaining funds: <REMAINING_WEALTH_K>k tokens

You have two choices:
1. give_up: abandon the task and lose the <YOUR_BUDGET_K>k
  tokens already invested.
2. increase: add to the budget, with an additional amount
  between 0 and <REMAINING_WEALTH_K>k.

Output your decision in JSON format, with no other text:

- Abandon:
  {"reasoning": "<Your reasoning>", "action": "give_up"}

- Add to the budget:
  {"reasoning": "<Your reasoning>", "action": "increase",
    "additional_budget_k": <additional amount>}
```