

Learning to Wait: Synchronizing Agents with the Physical World

Anonymous ACL submission

Abstract

Real-world agentic tasks, unlike synchronous Markov Decision Processes (MDPs), often involve non-blocking actions with variable latencies, creating a fundamental *Temporal Gap* between action initiation and completion. Existing environment-side solutions, such as blocking wrappers or frequent polling, either limit scalability or dilute the agent’s context window with redundant observations. In this work, we propose an **Agent-side Approach** that empowers Large Language Models (LLMs) to actively align their *Cognitive Timeline* with the physical world. By extending the Code-as-Action paradigm to the temporal domain, agents utilize semantic priors and In-Context Learning (ICL) to predict precise waiting durations ($\text{time.sleep}(t)$), effectively synchronizing with asynchronous environment without exhaustive checking. Experiments in a simulated Kubernetes cluster demonstrate that agents can precisely calibrate their internal clocks to minimize both query overhead and execution latency, validating that temporal awareness is a learnable capability essential for autonomous evolution in open-ended environments.

1 Introduction

Constructing practical agentic training environments, such as Kubernetes (K8s) cluster sandboxes (Ardebili and Bartolini, 2025), presents a fundamental challenge: establishing precise temporal alignment between actions and their delayed feedback within a non-blocking asynchronous context. Unlike the synchronous Markov Decision Process (MDP) paradigm prevalent in frameworks like Gymnasium (Towers et al., 2025), real-world agent actions (e.g., `kubectl` commands) rarely trigger an immediate convergence of the environment state. Instead, they involve significant, variable temporal latency (Bellinger et al., 2024; Biedenkapp et al., 2021). This critical distinction involves a *Temporal Gap* ($T_{\text{act}} \ll T_{\text{true}}$), which has been largely

obscured by current research trends focusing on mathematical reasoning (Shao et al., 2025) or code generation (Mai et al., 2025). In such deterministic scenarios, interaction time is typically equivalent to task completion time ($T_{\text{act}} \approx T_{\text{true}}$), leaving the complexities of asynchronous temporal alignment effectively unexplored.

As illustrated in Figure 1, we abstract this challenge as an alignment problem between the agent’s *Cognitive Timeline* and the environment’s *Physical Timeline*. To bridge the discrepancy between event initiation and completion, two distinct paradigms exist: the **Environment-side Approach** and the **Agent-side Approach**. The Environment-side Approach relies on engineering wrappers to block execution or employs naive periodic polling. However, these methods are structurally flawed. Wrappers block the agent until tasks finish, forcing a hard coupling of the cognitive clock with the physical clock. Polling, while non-blocking, incurs non-negligible query costs (Krueger et al., 2020) and, more critically, consumes the LLM’s finite context window, which is a valuable resource. Filling the context window with redundant intermediate observations dilutes information density and degrades the signal-to-noise ratio, ultimately impairing the agent’s reasoning capabilities (Liu et al., 2024). The Agent-side Approach, conversely, acknowledges the independence of the physical world, requiring the agent to utilize its cognitive capabilities to predict a necessary waiting duration (T_{sleep}) to compensate for asynchronous latency.

Guided by the vision of Artificial General Intelligence (AGI) and the principles of the Scaling Law, we argue that the Agent-side perspective represents the more scalable and generalizable direction. Environment-side engineering shifts an unscalable maintenance burden onto developers by relying on hard-coded heuristics. Conversely, while mathematical estimators offer computational efficiency (Nombebe et al., 2022; Chen et al., 2018),

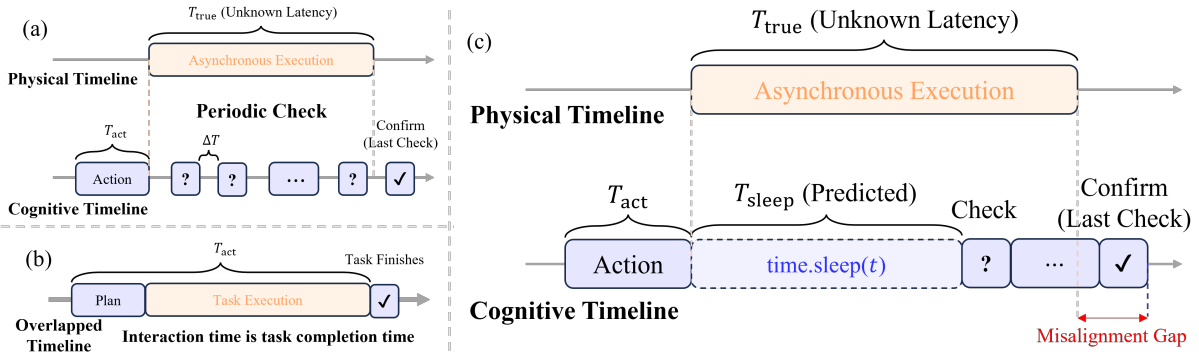


Figure 1: **The Temporal Alignment Problem.** The asynchronous nature of real-world environments creates a discrepancy between the *Physical Timeline* and the Agent’s *Cognitive Timeline*. While (a) **Periodic Check** forces alignment at prohibitively high query costs, (b) **General Agent Tasks** (e.g., Coding, Math) have obscured this challenge, as the agent’s generation time naturally spans the task duration ($T_{\text{act}} \approx T_{\text{true}}$), leaving no temporal gap to manage, (c) **Our Approach** actively predicts an optimal T_{sleep} to synchronize the agent’s internal clock with physical latency, minimizing misalignment without redundant queries.

they remain fundamentally blind to the semantic nuances of latency. A statistical model indiscriminately averages latencies for identical actions (e.g., `docker pull`), failing to distinguish arguments; in contrast, an intelligent agent leverages semantic reasoning to distinguish how specific arguments differ in action duration (Wei et al., 2023), recognizing the vast temporal disparity between pulling a lightweight base image and a massive CUDA image. Furthermore, by augmenting this intuition with external tools, such as querying search engines (Li et al., 2025) to retrieve the specific size of a novel image, agents can transform blind waiting into active temporal planning, thereby achieving precise alignment with the physical world while minimizing query overhead.

To operationalize this alignment concept, we extend the **Code-as-Action** paradigm (Wang et al., 2024) to the temporal domain. We empower the agent to actively align its cognitive clock by coding `time.sleep(t)`, thereby achieving synchronization with the physical timeline without blocking the execution flow. We validate this capability within an **Interleaved Action** framework, where the agent must execute a sequence of related yet distinct actions. This setting challenges the agent to move beyond learning a single global latency parameter, compelling it to simultaneously disentangle and calibrate the heterogeneous latency distributions of multiple actions. Leveraging In-Context Learning (ICL) (Dong et al., 2024), the agent processes historical execution feedback, specifically the temporal alignment errors from previous episodes, to iteratively fine-tune its internal clock for each action. This approach demonstrates that an agent can transcend static memorization to achieve dy-

namic, multi-objective temporal calibration in non-stationary environments. More importantly, this ability to perceive, predict, and adapt to the environment’s physical temporality is an indispensable prerequisite for building agents capable of continuous learning and self-evolution. Only by mastering the latency distribution of an asynchronous world can agents truly embark on their journey of autonomous evolution in open-ended environments.

2 Methodology

We formulate the temporal adaptation challenge as a Repeated Temporal Game where the agent balances query costs against waiting latency. Given minimal state feedback, the agent must infer the distinct temporal characteristics of each action from execution history.

2.1 Simulated Asynchronous Environment

To simulate tasks with distinct temporal properties, we construct an environment where commands trigger background processes with stochastic latencies. Unlike atomic memoryless events, complex asynchronous tasks comprise multiple independent stages. Thus, we model the completion time T_{true} using a **Gamma distribution** (Langaris, 1986) to capture this cumulative variance (details in Appendix A). This formulation offers precise control over latency statistics for a realistic simulation.

The environment provides binary feedback $S \in \{\text{PENDING}, \text{DONE}\}$ without auxiliary progress hints. Operating on an already initiated task, the agent utilizes two tools: (1) Active Wait (`time.sleep(t)`) to calibrate its internal clock, and (2) Status Check (`env.check()`) to verify com-

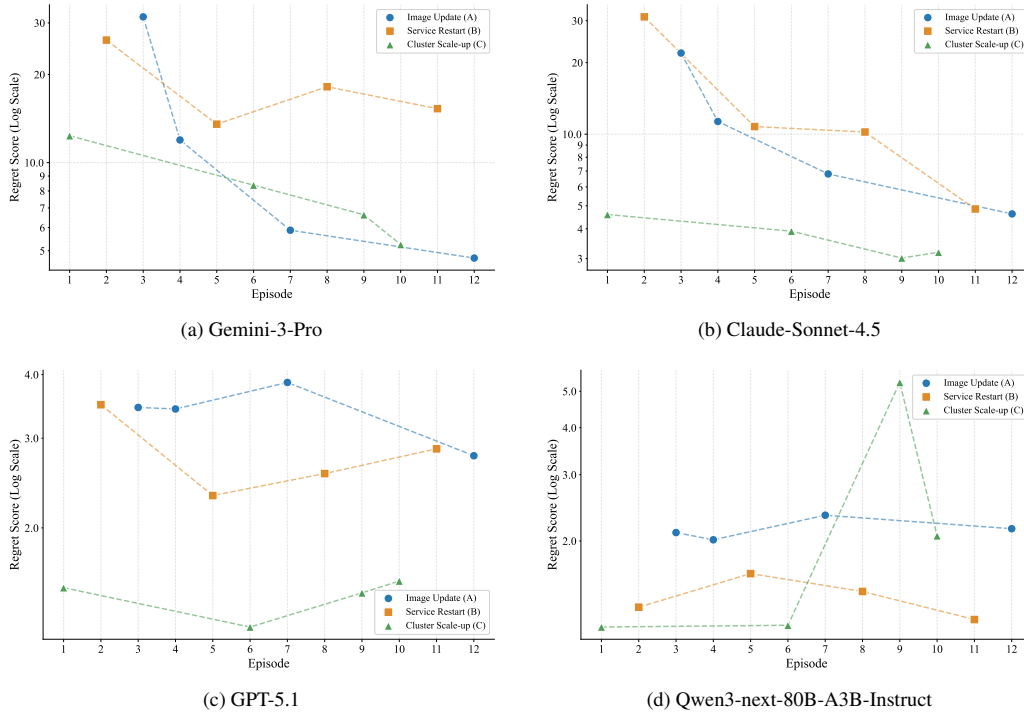


Figure 2: A comparison of Regret Scores for four representative LLMs. The score quantifies the agent’s efficiency, where a lower value indicates better performance.

pletion. The objective is to minimize check counts while aligning the wait duration with T_{true} .

2.2 Inter-Episode History Feedback

To leverage In-Context Learning (ICL), we structure the task as a multi-round repeated game. The prompt for the k -th episode includes an execution summary of the $(k - 1)$ -th episode. This summary contains the specific action performed, the number of checks and the total time consumed. Through this feedback loop, the agent incrementally refines its latency estimates for distinct actions, optimizing later predictions without updating parameters.

3 Experiments and Analysis

Our experiments will test the hypothesis that LLMs leverage semantic priors for initial coarse-grained estimation and use historical feedback to calibrate actions, bypassing cold-start issues.

3.1 Experimental Setup

We simulate three distinct tasks modeled by independent Gamma distributions (shape $\alpha = 20$, see Appendix A) with varying means: a lightweight **Image Update** (Action A, $\mu = 35\text{s}$), a medium **Service Restart** (Action B, $\mu = 45\text{s}$), and a heavy **Cluster Scale-up** (Action C, $\mu = 55\text{s}$). Full command details are in Appendix B. To simulate realistic constraints, completion times T_{true} are bounded

within intervals. The diversity requires the agent to distinguish between tasks using semantic cues, avoiding fitting a single global distribution.

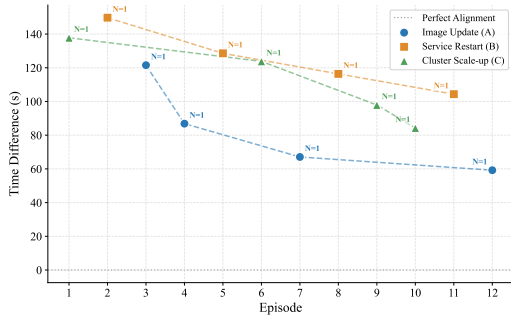
To evaluate whether temporal adaptation is a universal capability across different models, we select four representative LLMs: **Gemini-3-Pro** (Reasoning-enhanced), **Claude-Sonnet-4.5** (Code-specialized), **GPT-5.1** (General SOTA), and **Qwen3-next-80B-A3B-Instruct** (Mid-sized open-weights). This selection allows us to analyze the impact of reasoning strength, coding proficiency, and model scale on temporal planning.

3.2 Metric

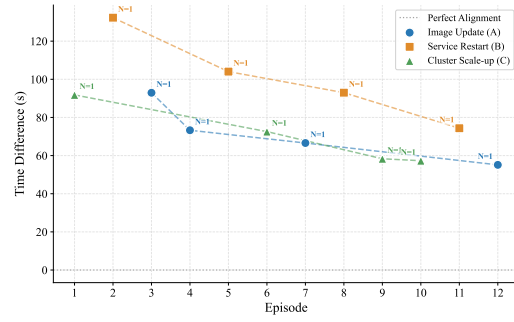
We define the **Regret Score** to quantify the deviation from optimal solution:

$$\text{Score} = N_{\text{check}} \cdot \exp\left(\frac{T_{\text{confirm}} - T_{\text{true}}}{T_{\text{true}}}\right) \quad (1)$$

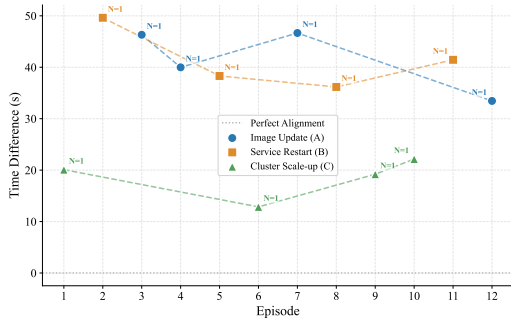
where N_{check} denotes the number of checks performed, and $(T_{\text{confirm}} - T_{\text{true}}) \geq 0$ represents the delay between the task actually becoming ready and the agent confirming it. A lower score indicates higher efficiency. This metric intentionally leverages the ground-truth T_{true} , because the focus of this work is to directly measure the agent’s core capability to align its internal timeline with physical reality, not to provide a practical training recipe in online environments.



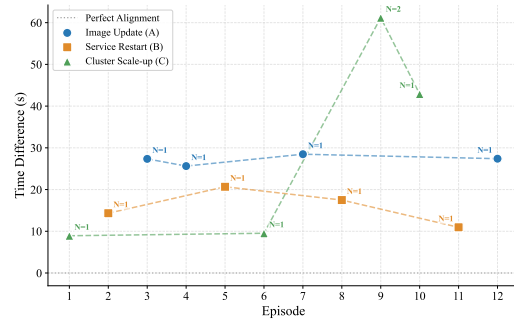
(a) Gemini-3-Pro



(b) Claude-Sonnet-4.5



(c) GPT-5.1



(d) Qwen3-next-80B-A3B-Instruct

Figure 3: Analysis of the temporal prediction error ($T_{\text{confirm}} - T_{\text{true}}$) over episodes. The annotation $N = 1$ on the data points indicates the model only checks once.

3.3 Results and Analysis

Our experiments confirm that LLMs can indeed calibrate their *Cognitive Timeline* to the physical world via In-Context Learning (ICL), though distinct behavioral patterns emerge across model architectures (Figure 2 and Figure 3).

Dynamic Calibration via ICL. Reasoning-enhanced and code-specialized models, specifically **Gemini-3-Pro** and **Claude-Sonnet-4.5**, validate the core hypothesis. Both models exhibit a clear learning curve: they initiate with a conservative strategy (high safety buffer to ensure task completion), resulting in higher initial Regret Scores. However, leveraging historical feedback, they rapidly reduce the prediction error ($T_{\text{confirm}} - T_{\text{true}}$) over episodes. This demonstrates a genuine ability to treat time as an optimizable variable, dynamically trimming the wait buffer while maintaining the optimal checking frequency.

Zero-shot Grounding vs. Rigid Heuristics. The behavior of **GPT-5.1** and **Qwen3-next** reveals a nuance between *capability* and *strategy*. **GPT-5.1** shows a flatter trend but maintains a consistently low Regret Score and Time Difference from the outset. This suggests superior zero-shot temporal grounding; its initial semantic priors were already precise enough to bypass the heavy calibration

phase needed by others. Conversely, while **Qwen3-next** achieves numerically competitive scores, a granular inspection of its trajectory reveals a failure in temporal adaptation. Instead of learning from context, the model defaults to a rigid heuristic (a static `time.sleep(60)`), which coincidentally covers the maximum latency ($\approx 55s$) and results in the clear plateau observed in Figure 3. While effective in this specific bounded environment, this lack of plasticity implies a failure to generalize, highlighting the distinction between blind robustness and intelligent temporal alignment.

4 Conclusion

This work bridges the Temporal Gap in asynchronous environments by empowering agents to actively manage their *Cognitive Timeline*. We demonstrate that through the Code-as-Action paradigm, LLM can leverage semantic priors and historical feedback to align itself with physical latency dynamics. While model behaviors vary, the results confirm that temporal awareness is a learnable capability. This shift from environment-side engineering to agent-side adaptation not only minimizes query overhead but also establishes a fundamental prerequisite for self-evolving agents to efficiently interact with the non-blocking, variable-latency reality of real-world systems.

259 Limitations

260 While our results demonstrate the efficacy of Agent-
261 side temporal alignment, we acknowledge two
262 boundaries of the current study that invite future
263 exploration. First, to ensure experimental control
264 and reproducibility, we utilized a simulated
265 asynchronous environment governed by Gamma
266 distributions. While this statistically mirrors the
267 multi-stage nature of real-world operations (e.g.,
268 container provisioning), it abstracts away low-level
269 system noise (e.g., network jitter or preemption)
270 that might introduce additional stochasticity in a
271 live production cluster. Second, our evaluation
272 focuses on the critical cold-start and rapid adapta-
273 tion phases (first 12 episodes) of independent tasks.
274 We prioritize this few-shot regime because real-
275 world agents frequently encounter novel commands
276 where long-term asymptotic convergence is less rel-
277 evant than immediate efficiency. We acknowledge
278 that the two-phase strategy in our prompt (predict
279 a conservative delay, then gradually decrease) con-
280 stitutes an expert-guided heuristic. While this vali-
281 dated ICL as an effective, light-weight calibration
282 mechanism in a few-shot setting, a crucial future
283 step is to explore how to train LLMs via RL to en-
284 code the latent temporal distribution into weights
285 and further realize the self-evolving agent’s adapt-
286 ability in real cluster operations.

287 References

288 Mohsen Seyedkazemi Ardebili and Andrea Bartolini.
289 2025. [Kubeintellect: A modular llm-orchestrated
290 agent framework for end-to-end kubernetes manage-
291 ment](#). *Preprint*, arXiv:2509.02449.

292 Colin Bellinger, Mark Crowley, and Isaac Tamblyn.
293 2024. [Dynamic observation policies in observa-
294 tion cost-sensitive reinforcement learning](#). *Preprint*,
295 arXiv:2307.02620.

296 André Biedenkapp, Raghu Rajan, Frank Hutter, and
297 Marius Lindauer. 2021. [Temporl: Learning when to
298 act](#). In *Proceedings of the 38th International Confer-
299 ence on Machine Learning*, volume 139 of *Proceed-
300 ings of Machine Learning Research*, pages 914–924.
301 PMLR.

302 W. C. Chen, A. Tareen, and J. B. Kinney. 2018. [Den-
303 sity estimation on small data sets](#). *Phys. Rev. Lett.*,
304 121:160605.

305 Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan
306 Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu,
307 Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui.
308 2024. [A survey on in-context learning](#). In *Proceed-
309 ings of the 2024 Conference on Empirical Methods*

in Natural Language Processing, pages 1107–1128, 310
Miami, Florida, USA. Association for Computational 311
Linguistics. 312

David Krueger, Jan Leike, Owain Evans, and John Sal- 313
vatier. 2020. [Active reinforcement learning: Observ- 314
ing rewards at a cost](#). *Preprint*, arXiv:2011.06709. 315

C. Langaris. 1986. [The waiting-time process of a queue- 316
ing system with gamma-type input and blocking](#). *317
Journal of Applied Probability*, 23(1):166–174. Ac- 318
cessed: 2025-12-08. 319

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yu- 320
jia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng 321
Dou. 2025. [Search-o1: Agentic search-enhanced 322
large reasoning models](#). In *Proceedings of the 2025 323
Conference on Empirical Methods in Natural Lan- 324
guage Processing*, pages 5420–5438, Suzhou, China. 325
Association for Computational Linguistics. 326

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, 327
Michele Bevilacqua, Fabio Petroni, and Percy 328
Liang. 2024. [Lost in the middle: How language mod- 329
els use long contexts](#). *Transactions of the Association 330
for Computational Linguistics*, 12:157–173. 331

Xinji Mai, Haotian Xu, Zhong-Zhi Li, Xing W, Weinong 332
Wang, Jian Hu, Yingying Zhang, and Wenqiang 333
Zhang. 2025. [Agent rl scaling law: Agent rl with 334
spontaneous code execution for mathematical prob- 335
lem solving](#). *Preprint*, arXiv:2505.07773. 336

Thobeka Nombebe, James Allison, Leonard Santana, 337
and Jaco Visagie. 2022. [On fitting the lomax dis- 338
tribution: a comparison between minimum distance 339
estimators and other estimation techniques](#). *Preprint*, 340
arXiv:2207.06086. 341

Zhihong Shao, Yuxiang Luo, Chengda Lu, Z.Z. Ren, 342
Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and 343
Xiaokang Zhang. 2025. [Deepseekmath-v2: Towards 344
self-verifiable mathematical reasoning](#). 345

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. 346
Balis, Gianluca De Cola, Tristan Deleu, Manuel 347
Goulão, Andreas Kallinteris, Markus Krimmel, Ar- 348
jun KG, Rodrigo Perez-Vicente, Andrea Pierré, 349
Sander Schulhoff, Jun Jet Tai, Hannah Tan, and 350
Omar G. Younis. 2025. [Gymnasium: A standard 351
interface for reinforcement learning environments](#). 352
Preprint, arXiv:2407.17032. 353

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, 354
Yunzhu Li, Hao Peng, and Heng Ji. 2024. [Executable 355
code actions elicit better LLM agents](#). In *Proceed- 356
ings of the 41st International Conference on Machine 357
Learning*, volume 235 of *Proceedings of Machine 358
Learning Research*, pages 50208–50232. PMLR. 359

Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert 360
Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, 361
Da Huang, Denny Zhou, and Tengyu Ma. 2023. 362
[Larger language models do in-context learning dif- 363
ferently](#). *Preprint*, arXiv:2303.03846. 364

Table 1: Details of the three simulated Kubernetes tasks. Each action is characterized by its semantic name, mean completion time (μ), the full kubectl command, and the rationale for its expected latency, which is derived from the complexity of the underlying cluster operations.

Action	Task Name	Mean Latency (μ)	Command & Rationale
A	Image Update	35s	<code>kubectl set image deployment/webapp-frontend new-container=nginx:1.23.4</code> <i>A lightweight operation that triggers a rolling update. The latency is dominated by control plane metadata changes, pod scheduling, and pulling a potentially cached image layer.</i>
B	Service Restart	45s	<code>kubectl rollout restart statefulset/prometheus-db</code> <i>A medium-weight operation that initiates a graceful, ordered restart of a stateful application. It is inherently slower as it must terminate and replace pods sequentially to maintain stability and data consistency.</i>
C	Cluster Scale-up	55s	<code>kubectl scale statefulset/etcd-cluster --replicas=5</code> <i>A heavy-duty operation involving not just pod scheduling but also provisioning new persistent volumes (high I/O latency) and secure cluster quorum joining (network discovery and state sync).</i>

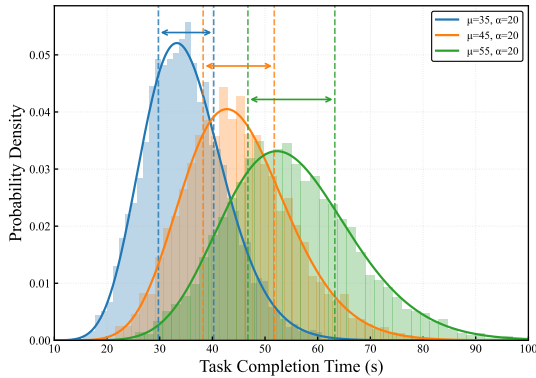


Figure 4: Probability density functions of the simulated task latencies. We model T_{true} using Gamma distributions to capture the multi-stage nature of asynchronous operations.

A Distribution Visualization

Figure 4 illustrates the probability density functions (PDF) of the task completion times (T_{true}) used in our simulated environment. We explicitly choose the **Gamma distribution** over the exponential distribution typically used in queuing theory. While a single atomic event (like a packet arrival) is memoryless and exponential, complex Kubernetes operations are composed of a sequence of dependent sub-stages (e.g., API validation \rightarrow Pod termination \rightarrow Image pulling \rightarrow Container startup). The sum of multiple independent exponential variables follows a Gamma distribution.

B Kubernetes Task Command Details

This section provides the full kubectl commands used in our experiments and a brief explanation of their expected operational latency in Table 1.

C Impact of Inference Latency on Temporal Calibration

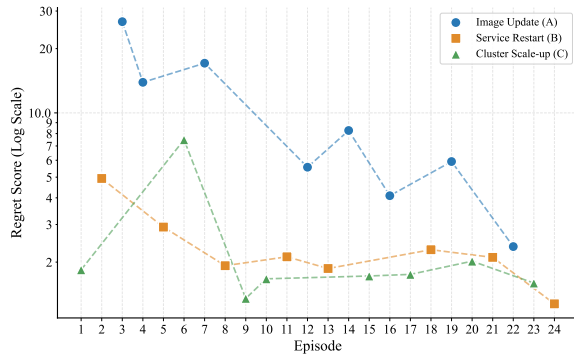
While a theoretical formulation might suggest explicitly subtracting the generation latency (T_{gen}) from the target wait time, we omit this step to maintain an end-to-end adaptive capability. Our experiments indicate that T_{gen} exhibits high variance, rendering a static subtraction mechanism ineffective. Specifically, the agent might typically operate at two extremes: it either outputs the sleep command immediately, making T_{gen} negligible, or engages in redundant reasoning where T_{gen} alone suffices to cover the task duration, eliminating the need for additional waiting. Consequently, we treat the inference latency as an intrinsic component of the total system delay. This approach compels the agent to predict the residual wait time required after generation, thereby ensuring generalization across different models and inference infrastructures.

D Prompt Details

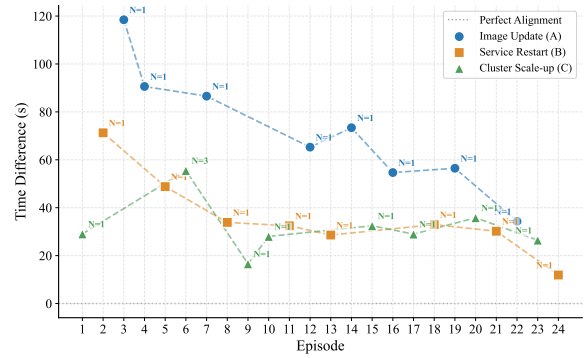
Table 2 presents the exact prompt structure used in our experiments, formatted to illustrate the inputs and instructions provided to the agent.

Role	Content
System	You are a helpful AI agent. To solve the task, you must use the <code>execute_python_code</code> tool. Do not write code in your response directly.
User	<p>[CONTEXT] You are an intelligent agent operating in a Kubernetes environment. You have just executed the following <code>kubectl</code> command: <code>> > {kubectl command}</code> Pay attention! You just edited the cluster and now need to wait for the command to complete. If you check too early, you'll find the task is still pending which will incur serious accidents. If you wait too long, you'll incur unnecessary delays. Your goal is to synchronize with the completion of this task effectively. The task is running in the background with a variable latency.</p> <p>[HISTORICAL TRAJECTORIES] To act, you MUST analyze the historical trajectories for this specific command and then use the <code>execute_python_code</code> tool. Provide the code to run as the <code>'code'</code> parameter. {history}</p> <p>[AVAILABLE ACTIONS]</p> <p>1. Check Status - Description: Check if the target time has elapsed. This is the only way to successfully finish the task. - Tool Usage: Call <code>execute_python_code</code> with parameter <code>code="check()"</code>. - Output: "Status: PENDING..." (incomplete) or "Status: DONE..." (terminates episode).</p> <p>2. Wait - Description: Wait for a specific duration. - Tool Usage: Call <code>execute_python_code</code> with parameter <code>code="import time; time.sleep(seconds)"</code>. - Output: "Execution successful. Time has passed."</p> <p>[OPTIMIZATION GOAL] Your performance is evaluated by a Regret Score (Lower is better). The score is a combination of two factors: 1. Check Count (High Priority): Every check after the first one adds a large penalty. Your primary goal is to achieve <code>Check Count = 1</code>. 2. Wait Precision: The time you wait should be as close as possible to the true (hidden) completion time. Waiting excessively long will also increase your Regret Score.</p> <p>[TWO-PHASE STRATEGY] Follow this procedure to make your decision: Phase 1: Establish a Safe Baseline - If this is the first time you see this specific command, you have no history. Make a conservative first guess based on your semantic prior knowledge. A long wait is better than failing. Goal: get the first data point with <code>Check Count = 1</code>. Phase 2: Cautious Optimization - Once you consistently achieve <code>Check Count = 1</code>, analyze the history for the <i>current command type only</i>. - Find your last successful wait time (<code>Last_Wait</code>). - Propose a new wait time that is slightly less than Last_Wait (e.g., 10-20% reduction) but not so low that you risk failing.</p> <p>[SYSTEM INSTRUCTIONS] At each step, you must decide whether to wait or check and call the <code>execute_python_code</code> tool accordingly. Use wait to approach the target time, and check only when you are confident the time has elapsed.</p>
History Format	<p>Episode i: Command = <code>'kubectl command'</code>, Your Executed Sleep Time = ts, Check Count = N</p>

Table 2: The complete prompt template used for the Agent-side Approach. The prompt provides the agent with the semantic context, historical execution data, and a clear two-phase strategy to minimize the Regret Score.



(a) Regret Score Trajectory



(b) Time Difference Trajectory

Figure 6: **Temporal Adaptation Dynamics of kimi-k2-0905.** (a) The significant reduction in Regret Score demonstrates the agent’s rapid convergence towards an optimal checking strategy. (b) The Time Difference metric reveals how the agent minimizes the *Temporal Gap*, effectively synchronizing its *Cognitive Timeline* with the asynchronous physical latency. Annotations ($N = x$) indicate the check count for specific episodes.

E Evolutionary Trajectory of Alignment

To provide a granular perspective on how Large Language Models align their internal *Cognitive Timeline* with the external *Physical Timeline*, we conducted an extended long-horizon evaluation (24 episodes) utilizing **kimi-k2-0905**. This qualitative case study empirically validates the *Two-Phase Strategy* proposed in our methodology. By analyzing the Chain-of-Thought (CoT) reasoning traces, we demonstrate how the agent transitions from utilizing generalized semantic priors to performing precise, history-driven calibration.

Phase 1: Zero-Shot Temporal Grounding In the cold-start phase, the agent lacks historical execution data. Consequently, it must rely on *Zero-Shot Temporal Grounding*—inferring physical latency directly from the semantic complexity of the command. As shown in Table 3, when the agent first encounters the *Cluster Scale-up* task (Episode 1), it identifies key terms such as "etcd" and "distributed consensus." Recognizing that maintaining state consistency is time-consuming, it sets a conservative baseline of 60 seconds to ensure task success. Importantly, this reasoning is context-aware rather than static. In Episode 2, when facing a *Service Restart*, the agent explicitly compares it to the previous scaling task. It reasons that a "restart" involves a two-step process (termination followed by creation) and is therefore computationally heavier than simple scaling. Based on this semantic derivation, it proactively increases the safety buffer to 90 seconds. This demonstrates that the agent

can construct a hierarchical mental model of task latency solely through semantic reasoning.

Phase 2: Adaptive Calibration via Feedback

Once historical data becomes available, the agent shifts its strategy to *Adaptive Calibration*. The objective changes from **ensuring success** (maximizing safety) to **minimizing regret** (optimizing efficiency). By analyzing the *Inter-Episode History Feedback*, the model detects trends in the execution time. For example, in the later stages of the experiment (Episode 24), the agent observes a sequence of successful executions where the wait time decreased from 90s to 45s. Recognizing that the system has stabilized, the agent abandons the initial conservative estimate and applies a decay factor (approximately 10%) to the wait time. This behavior confirms that the agent can treat time as an optimizable variable. Through In-Context Learning, the agent effectively fine-tunes its internal clock to match the stochastic reality of the environment, achieving asymptotic convergence without updating its parameters.

Ep	Task Type	Chain-of-Thought Reasoning (Compressed by LLM)	T_{sleep}
<i>Phase 1: Zero-Shot Temporal Grounding (Establishing Safe Baselines)</i>			
1	Cluster Scale-up	I need to analyze the situation and make a decision about waiting for the command to complete. Since I have no historical trajectories for this specific command, I need to establish a safe baseline. The command is scaling a StatefulSet, which involves bringing up new pods in sequence with proper ordering and stability checks. For an 'etcd' cluster, this is particularly important as 'etcd' requires careful pod management due to its distributed consensus algorithm. While a safe conservative estimate would be around 90-120 seconds, I will start with a moderately conservative approach. I will wait 60 seconds as my initial safe baseline to give the StatefulSet controller enough time to create and stabilize the new pods.	60s
2	Service Restart	This is the first time I am encountering the 'kubectl rollout restart' command in the historical trajectory. Looking at the reference point: Episode 1 ('scale') took 60s. However, a rollout restart operation is more complex —it needs to gracefully terminate existing pods and create new ones. This process depends on factors like the number of replicas, pod termination grace periods, image pull times, and resource availability. Since this is my first attempt and I need to establish a safe baseline, I will choose a conservative wait time that is longer than the 60s used in the reference scaling example. I will wait 90 seconds as a reasonable, conservative first guess.	90s
3	Image Update	The command is 'kubectl set image'. Looking at the history, I see Episode 1 (Scale, 60s) and Episode 2 (Restart, 90s). I do not have history for this exact command type, but general Kubernetes knowledge suggests that updating a deployment requires rolling out new pods and terminating old ones. This process is structurally more similar to a 'rollout restart' than a simple 'scale' operation. The process usually takes longer than simple scaling operations depending on the deployment size. Given no specific history, I need a safe baseline. I will use a conservative estimate significantly longer than the historical times (60s and 90s) to ensure I achieve 'Check Count = 1'. I will wait 120 seconds.	120s
<i>Phase 2: Adaptive Calibration (Optimization & Convergence)</i>			
4	Image Update	Looking at the historical trajectories, I can see that this exact command was executed in Episode 3 with a wait time of 120.0s and completed successfully (Check Count = 1). This means I have perfect history for this specific command. Following Phase 2 (Cautious Optimization), I should try to reduce the wait time from the previous 120.0s by a safe margin to improve efficiency. A 10% reduction seems reasonable—that is $120s \times 0.9 = 108s$. This represents a small but meaningful improvement in precision while maintaining a high confidence of success. I will wait for 108 seconds.	108s
6	Cluster Scale-up	Looking at the historical trajectories, I can see this exact command 'kubectl scale' was executed in Episode 1 with a wait time of 60.0s and successful completion (Check Count = 1). Since I have history for this specific command and it was successful, I should use this as my baseline. I do not need to reduce the time yet because I only have one data point and I want to ensure success. My priority is to guarantee the task completes rather than optimization at this early stage. I will wait the same 60s to guarantee the task completes.	60s
11	Service Restart	Looking at the historical data for 'rollout restart', I can see a clear trend: Episode 2 (90s), Episode 5 (80s), and Episode 8 (70s). All were successful with Check Count = 1. The trend shows the wait times decreasing from 90s → 80s → 70s. The most recent successful wait was 70s. Since I am consistently achieving success but want to optimize precision, I will reduce the wait time by a small margin (about 15%) from the last successful attempt. Calculation: $70s \times 0.85 \approx 60s$.	60s
20	Cluster Scale-up	This is a repeated operation. From the history, I see the following wait times: Episodes 1, 9, 10, 15, 17 used various times ranging from 58-65s, all with Check Count = 1. Episode 6 (75s) had Check Count = 3, which indicates the wait was too short (likely an outlier). The most recent successful executions (Episodes 15 and 17) used 65s and 58s respectively. Since I want to cautiously optimize while maintaining Check Count = 1, I will use a wait time that is slightly less than the most recent successful time. I will wait 60 seconds, which is a balanced choice based on the historical data.	60s
24	Service Restart	Looking at the historical trajectories, I see the following execution times: 90s (Ep 2) → 80s → 70s → 60s → 55s → 50s → 45s (Ep 21). I notice a clear trend of decreasing wait times over the episodes, with the most recent successful execution taking only 45s. This suggests the system has been optimized and is now working more efficiently. Following the optimization strategy, I should reduce the wait time slightly from the last successful execution. I will decrease by about 10% from 45s to approximately 40s.	40s

Table 3: The partial CoT trajectories of kimi-k2-0905.