

CMHA: A Counterfactual-Memory Hierarchical Architecture for Real-Time Strategy Games

Anonymous ACL submission

Non-archival submission

Abstract

Deploying large language models (LLMs) to real-time strategy (RTS) game environments for validating their reasoning and decision-making capabilities has become a popular research focus. Existing methods typically rely on traditional memory retrieval and unified execution via a single cloud-based LLM, which reveals severe bottlenecks in cognitive reasoning and inference latency. To address these limitations, we propose CMHA, a hierarchical agent architecture with counterfactual memory. At the macro level, CMHA employs a memory mechanism inspired by hypothetical reasoning to maintain long-term strategic consistency. At the micro level, we combine action pruning with a lightweight 2.8B tactical model fine-tuned via QLoRA to translate intents into executable actions. This architecture incurs minimal expensive deployment overhead, while sustaining tactical coherence and reducing inference latency. Experiments in the LLM-PySC2 environment demonstrate that our approach consistently outperforms cloud-based baselines while significantly reducing inference latency and token consumption.¹

1 Introduction

LLMs have exhibited strong generalization and reasoning abilities, achieving substantial breakthroughs across a broad range of downstream tasks spanning code generation and complex tool invocation (Brown et al., 2020; Touvron et al., 2023). In recent years, deploying LLMs as autonomous agents in RTS games to evaluate their reasoning and decision-making capacity has emerged as an important direction in artificial intelligence (Wu et al., 2024; Chen et al., 2024). Among existing testbeds, StarCraftII stands out as a widely adopted

¹Code is released at <https://anonymous.4open.science/r/CMHA-LLM-RTS-A87D>

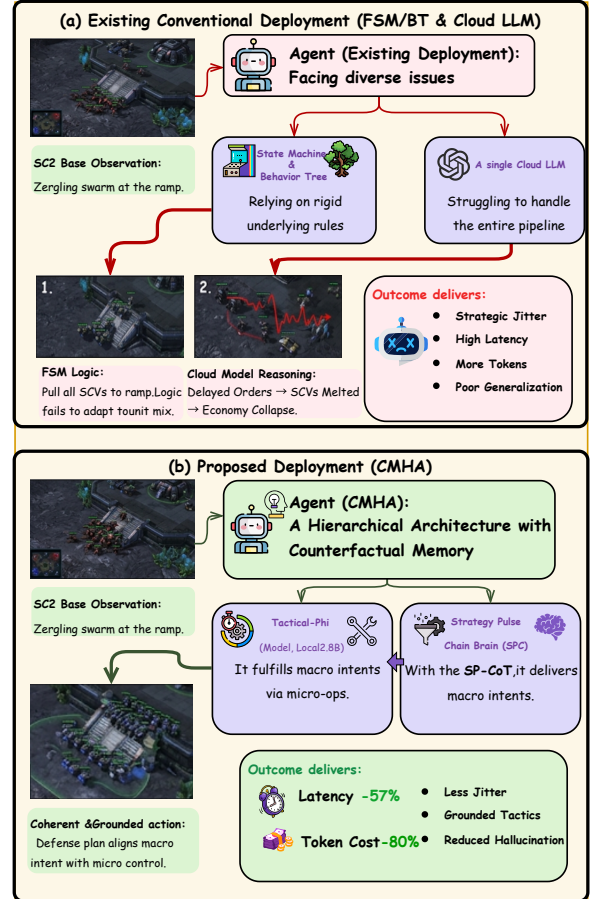


Figure 1: Comparison of existing conventional deployment and the proposed CMHA architecture.

evaluation platform, owing to its intense spatiotemporal dynamics, partial observability induced by the fog of war, and sophisticated economic and operational strategies.

Traditionally, environments such as StarCraftII have primarily been addressed through reinforcement learning paradigms. However, these approaches generally operate as uninterpretable black-box models, relying heavily on large-scale trial-and-error optimization while exhibiting limited generalization to unseen tactics and dynamic sce-

narios (Vinyals et al., 2017, 2019). Consequently, recent research has increasingly shifted toward monolithic cloud-based LLM architectures (Yao et al., 2022; Park et al., 2023). Nevertheless, such deployment paradigms still face several critical challenges. First, the inference latency introduced by cloud-based LLMs often leads to delayed tactical responses in fast-paced RTS environments. Second, as interaction histories continuously grow, conventional memory retrieval mechanisms tend to accumulate large amounts of redundant historical observations (Packer et al., 2023; Zhong et al., 2024). These redundant observations introduce additional logical noise, causing agents to frequently switch between conflicting strategic objectives in complex situations, thereby resulting in severe strategic instability.

Meanwhile, the semantic-action alignment between LLMs and low-level game environment still remains an urgent problem to be addressed (Huang et al., 2022). Several existing approaches (Colledanchise and Ögren, 2018) rely on finite-state machines or behavior trees as auxiliary control mechanisms. Although such methods can better comply with game rules, they suffer from rigid logic and restrict the reasoning and decision-making abilities of LLMs. In summary, current LLM deployment urgently requires a better paradigm that not only breaks the latency bottlenecks and cognitive redundancy brought by monolithic cloud-based LLMs, but also mitigates the adverse effects caused by low-level hallucination rates (Ji et al., 2023).

To address the challenges identified in existing works, we propose CMHAa hierarchical decision-making architecture equipped with counterfactual memory. At the macro level, we introduce a strategic planning module named Strategy Pulse Chain Brain (SPC), which operates at a slower decision frequency to perform global reasoning. Instead of storing all historical observations, the memory module of SPC selectively retains information based on hypothetical reasoning. At the micro level, we deploy Tactical-Phi, a lightweight local 2.8B model fine-tuned with QLoRA (Dettmers et al., 2023), to translate high-level strategic intents into executable low-level game actions. We further introduce action pruning to reduce the decision space, together with a self-correction sandbox mechanism that filters invalid actions during execution.

Our contributions can be summarized as follows:

- We propose CMHA, an asynchronous hierarchical decision-making framework. It not only fully leverages the reasoning and generalization capabilities of LLMs, but also forms a micro-action execution closed loop by introducing action pruning and a self-correction sandbox, thereby reducing the action hallucination rate.
- The memory mechanism of the architecture maintains long-term historical events, significantly reducing strategic jitter caused by cognitive overload (Li et al., 2026) in LLMs.
- Extensive experiments on the LLM-PySC2 benchmark demonstrate that CMHA achieves better performance, while maintaining comparable strategic depth to GPT-4omini, reducing inference latency by approximately 57% and token consumption by nearly 80%.

2 Related Work

2.1 RTS Environments for LLM Agents

Early research on StarCraftII primarily interacted with the game through reinforcement learning and the underlying interfaces of PySC2 (Vinyals et al., 2017). A pivotal breakthrough was AlphaStar (Vinyals et al., 2019), which demonstrated the potential of it in StarCraftII, yet it cannot avoid its fundamental limitations in generalization capability and computational efficiency. With the emergence of LLMs, the research paradigm has gradually shifted. Works such as TextSC2 (Ma et al., 2024) and SwarmBrain (Shao et al., 2024) began to abstract game APIs into natural language, attempting to construct experimental environments tailored for LLMs. Subsequently, to address the shortcomings of SwarmBrain and TextSC2, LLM-PySC2 (Li et al., 2026) proposed the first environment supporting the full PySC2 action space, while providing a structured game Wiki knowledge base.

2.2 Memory Information

Applying LLMs to long-horizon tasks requires robust reasoning and memory information management. Although Chain of Thought (CoT) (Wei et al., 2022) and Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) have strengthened the memory utilization and analytical capabilities of LLMs, they often lead to redundant contextual information in StarCraftII, which involves thousands of time steps. To handle redundant observations,

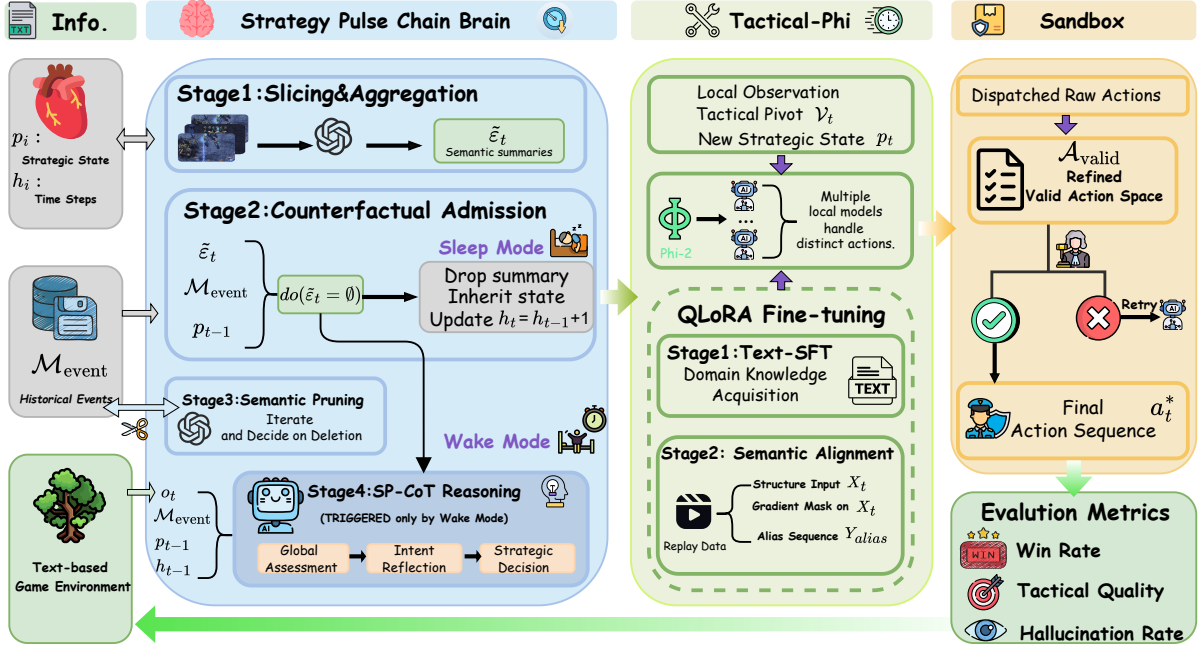


Figure 2: An overview of our framework CMHA, which consists of three parts: (a) **Strategy Pulse Chain (SPC) Brain**: manages counterfactual memory and conducts high-level strategic reasoning via SP-CoT; (b) **Tactical-Phi Adapter**: generates executable raw actions based on macro-level intentions; (c) **Action Sandbox**: verifies action validity and performs self-correction to reduce hallucinations.

Chain of Summarization (CoS) (Ma et al., 2024) proposed a method for periodically aggregating frame information. Recent advances have explored more complex memory and planning structures. COMPASS (Li et al., 2025) constructs a global entity memory through multi-hop communication, while StarWM (Zhang et al., 2026) introduces a structured world model to refine policies by predicting future dynamics. However, existing memory paradigms predominantly focus on the accumulation of short-term contextual information and lack effective mechanisms for long-term memory scheduling and maintenance.

2.3 Semantic-Action Alignment

Regarding the alignment of high-level LLM instructions with low-level executable actions, recent studies have explored a variety of action grounding strategies. For instance, several works introduce dedicated low-level modules to constrain the output space of LLMs. SwarmBrain (Shao et al., 2024) leverages finite state machines to construct a low-level reflex network, while Adaptive Command (Li et al., 2025) integrates behavior trees to guarantee rigid rule execution. Although these approaches achieve promising progress, they fail to fully exploit the reasoning potential of LLMs and

still suffer from poor generalization ability. Subsequent efforts such as LLM-PySC2 (Li et al., 2026) adopt a monolithic cloud LLM to handle the entire pipeline from strategic planning to primitive action execution, which nevertheless introduces non-negligible inference latency and inherent action hallucinations. Meanwhile, SC-Phi2 (Khan and Sukthankar, 2024) demonstrates the feasibility of fine-tuning a local lightweight model with SC2-specific textual datasets for macro management tasks.

3 Method

In this section, we elaborate the overall methodology of our proposed CMHA framework. The general architecture of CMHA is visualized in Figure 2. We first present the overall framework and design motivations in Section 3.1. Section 3.2 elaborates on macro-level components and investigates the working mechanism of dual counterfactual memory in depth. Afterwards, Section 3.3 illustrates the micro execution closed loop involving dynamic space pruning and self-correcting action sandbox mechanisms. Finally, we specify the two-stage fine-tuning strategy for the local Tactical-Phi model in Section 3.4.

3.1 CMHA Overview

At each decision time step t , the agent receives high-dimensional local and global observations o_t and outputs a valid and grounded action sequence a_t^* . To address the limitations of previous frameworks, we propose a hierarchical architecture based on counterfactual memory. CMHA maintains a set of cross-step macro state variables, including the strategic pulse p_t (attack/expand/retreat/defend) representing the current macro intention, the strategic inertia h_t recording the duration of the current intention, and the memory bank $\mathcal{M}_{event}$ storing all valid historical events. As illustrated in Figure 2, CMHA decouples the entire pipeline into three asynchronous layers with strict boundaries:

- **Strategy Pulse Chain Brain:** This layer takes the global observation o_t and historical state variables $(\mathcal{M}_{event}, p_{t-1}, h_{t-1})$ as input and is triggered asynchronously at a slow clock cycle. It integrates memory with Strategic Pulse Chain-of-Thought (SP-CoT) for reasoning, and aggregates the global plan into a high-level intention tuple $(p_t, \mathcal{V}_t)$ containing strategic pulse and tactical pivots, which is then sent to the middle-layer tactical adapter. This mechanism ensures the stability and effectiveness of long-term reasoning.
- **Tactical Adapter:** This layer receives local observations and the top-down macro intention $(p_t, \mathcal{V}_t)$, running at a fast clock frequency. To maintain lightweight characteristics, the tactical adapter only generates raw actions a_{raw} after reasoning independently.
- **Action Sandbox:** To reduce the hallucination rate, this module takes the raw action a_{raw} generated by the middle layer as input and performs validity checking based on the current environment. If the checked action is still invalid, the self-correction mechanism will be triggered. Finally, this layer outputs the final executable action a_t^* to the game engine.

3.2 Memory and Reasoning Mechanism

This module divides the maintenance and utilization of memory into four sequential stages, enabling the SPC brain to generate stable and effective macro strategies via SP-CoT reasoning.

Stage 1: Slicing and Aggregation

To eliminate high-frequency observation redundancy introduced by the game environment, the

system adopts an unbiased sliding window mechanism to compress the continuous time flow. Given a fixed step size w , the continuous game observation stream is split into independent observation viewports $W_t = \{o_{t-w+1}, \dots, o_t\}$. For each window, the system invokes the top-level large language model to perform multi-frame semantic aggregation, reducing dense information frames into structured high-level semantic segments $\tilde{\varepsilon}_t$:

$$\tilde{\varepsilon}_t \sim \text{LLM_Summarize}(W_t, \text{Prompt}_{\text{agg}}), \quad (1)$$

where LLM_Summarize denotes the context aggregation operation performed by the large language model, and $\text{Prompt}_{\text{agg}}$ represents the structured instruction for guiding spatiotemporal feature extraction.

Stage 2: Counterfactual Admission

To maintain strategic stability and prevent the unbounded inflation of the memory pool, the system executes a semantic-driven counterfactual intervention on $\tilde{\varepsilon}_t$ prior to reasoning. The top-level LLM is evaluated with a counterfactual challenge: given the established historical memory bank $\mathcal{M}_{event}$, whether the forceful erasure of the current episode, denoted as $do(\tilde{\varepsilon}_t = \emptyset)$, would yield a fundamental deficit in comprehending the current state o_t , or critically compel a disruption of the existing strategic pulse p_{t-1} . The model outputs a boolean routing signal $\mathbb{I}_{\text{keep}} \in \{0, 1\}$ based on semantic attribution:

$$\mathbb{I}_{\text{keep}} = \text{LLM_Admit}(o_t, p_{t-1}, \mathcal{M}_{event}, do(\tilde{\varepsilon}_t = \emptyset)), \quad (2)$$

where LLM_Admit denotes the counterfactual evaluation function executed by the LLM, $do(\cdot)$ represents the forceful intervention operator in causal inference, and $\mathcal{D}_{\text{all}}$ is the auxiliary global domain knowledge.

Based on the value of $\mathbb{I}_{\text{keep}}$, the SPC Brain routes the macro state machine into two discrete modes:

- **Sleep Mode ($\mathbb{I}_{\text{keep}} = 0$):** The episode is verified to lack substantial causal effects. The brain rejects global replanning, discards the event, inherits the current strategic pulse ($p_t = p_{t-1}$), and increments the strategic inertia ($h_t = h_{t-1} + 1$).
- **Wake Mode ($\mathbb{I}_{\text{keep}} = 1$):** The episode is recognized as a strategic pivot that fundamentally alters situational cognition. The temporary snippet $\tilde{\varepsilon}_t$ is formally instantiated as a core memory ε_t and appended to the causal

memory bank $\mathcal{M}_{\text{event}}$, which subsequently activates the reasoning pipeline in Stage 4.

Stage 3: Semantic Pruning

Meanwhile, the system periodically prunes invalid memories from $\mathcal{M}_{\text{event}}$. For each historical event $\varepsilon_i \in \mathcal{M}_{\text{event}}$, the system prompts the LLM to determine whether the latest observation logically semantically covers it or if its associated historical intention has terminated:

$$\mathbb{I}_{\text{prune}} = \text{LLM_Prune}(o_t, \varepsilon_i), \quad (3)$$

where LLM_Prune denotes the pruning evaluation function implemented by the LLM, and $\mathbb{I}_{\text{prune}} \in \{0, 1\}$ is the output boolean judgment; if $\mathbb{I}_{\text{prune}} = 1$, the event is pruned to ensure the purity of the context for subsequent reasoning.

Stage 4: SP-CoT Reasoning

Only in Wake Mode does the top-level brain perform reasoning, a mechanism that effectively reduces latency and token consumption. Based on o_t , $\mathcal{M}_{\text{event}}$, and the previous round of strategic inertia h_{t-1} , the brain conducts reasoning following SP-CoT:

$$\mathcal{C}_t \sim \text{LLM_Reason}(\mathcal{M}_{\text{event}}, p_{t-1}, h_{t-1}, \text{Prompt}_{\text{sp}}), \quad (4)$$

where LLM_Reason denotes the SP-CoT reasoning process, $\text{Prompt}_{\text{sp}}$ represents the prompt instruction for constraining the chain of thought, and $\mathcal{C}_t$ is the reasoning trajectory. Specifically, SP-CoT requires $\mathcal{C}_t$ to follow the paradigm of "enemy situation assessment $\rightarrow$ historical intention reflection $\rightarrow$ macro-strategic decision". Finally, the system derives the new tuple $(p_t, \mathcal{V}_t)$ from the end of the reasoning trajectory $\mathcal{C}_t$ and updates h_t according to the following logic:

$$h_t = \begin{cases} h_{t-1} + 1 & \text{if } p_t = p_{t-1} \\ 0 & \text{if } p_t \neq p_{t-1} \end{cases}. \quad (5)$$

3.3 Action Execution Closed-loop

Through dynamic pruning and self-correction mechanisms, the CMHA constructs a micro-execution loop at the middle and bottom layers. This mechanism reduces potential hallucinations of LLMs and substantially improves the executability of the output actions. Before outputting the action list to the tactical unit, the physical attributes of the unit (such as energy value and skill cooldown) are verified; any action that does not meet the attribute

constraints is removed from the list, resulting in the legal action space $\mathcal{A}_{\text{valid}}$.

The tactical unit generates the raw action a_{raw} , then sends it to the underlying self-correction sandbox to check its compliance with $\mathcal{A}_{\text{valid}}$. If format errors or unexecutable issues occur, a retry mechanism is triggered, which generates feedback and requires the tactical unit to regenerate the action (with a maximum of 2 retries). If the action still cannot be normally executed after retries, the sandbox replaces it with a pre-set safe fallback action. Finally, the legal action a_t^* is issued for execution, and this process can be expressed as:

$$a_t^* = \text{Sandbox}(a_{\text{raw}}, \mathcal{A}_{\text{valid}}). \quad (6)$$

3.4 Tactical-Phi Fine-Tuning

Considering the inherent limitations of cloud-based large models in aligning semantic instructions with executable actions, we locally deploy the 2.8B-parameter Phi-2 model (Li et al., 2023) and conduct two-stage fine-tuning via QLoRA. Detailed training configurations are provided in Appendix.

General pre-trained models lack domain knowledge for in-game scenarios. In the first stage, we perform text-supervised fine-tuning using the SC2 Text Dataset (Khan and Sukthankar, 2024). We freeze all backbone weights and only update attention layers and feed-forward networks inside LoRA modules during gradient optimization.

The second stage focuses on semantic alignment. We selected high-quality human-played game recordings and reconstructed the input-output space of supervised samples with reference to the MSC dataset (Wu et al., 2017). The whole pipeline leverages reverse reasoning without manual annotations. On the input side, we replaced the original high-dimensional tensor representation with a structured text summary X_t , which integrates three types of feature information: resource and population status discretized into categorical intervals (low, medium, high), building construction progress, and tech tree development status. For the output side, to avoid prefix bias caused by the original action templates in PySC2, we replaced the raw action labels with concise alias descriptions. When sampling underlying actions, we extracted four consecutive alias action sequences to form the final target sequence Y_{alias} . Additionally, to ensure the model focuses on action prediction, we adopted an output masking strategy during training: we masked the gradient of X_t when calculating

the cross-entropy loss, only performing supervised training on the alias action sequence. The alignment loss is defined as:

$$\mathcal{L}_{\text{align}} = - \sum_{j=1}^{|Y_{\text{alias}}|} \log P(y_j | X_t, y_{<j}; \Theta_{\text{tuned}}). \quad (7)$$

where Θ_{tuned} represents the model parameters updated by QLoRA. In practical deployment, the generated alias actions need to be mapped back to the original in-game operation commands.

4 Experiment

4.1 Experimental Setup

Environment and Task All experiments are carried out on StarCraftII (v5.0.15) via the LLM-PySC2 environment (Li et al., 2026). The local model is deployed on a single RTX 3090 GPU. Unlike previous simplified simulation platforms, LLM-PySC2 enables full access to original game actions. Following the methodology of it, We adopt the Simple64 map for all tests and use built-in AI opponents across all difficulty levels ranging from 1 to 7. All matches are conducted under the ECSB (highest complexity) mode and repeated twelve times. We also evaluate our method on typical tasks selected from SMAC test suites (Samvelyan et al., 2019).

Baseline Due to the complexity of the LLM-PySC2 environment, agents designed for simplified environments in prior work cannot be directly deployed or fairly compared. Therefore, we adopt the built-in multi-agent architecture of LLM-PySC2 as our baseline. GPT-4o-mini (OpenAI, 2024), and DeepSeek-V3 (DeepSeek-AI, 2025) are used as inference models.

Evaluation Metric To comprehensively quantify the model performance, we define the following six evaluation metrics:

- **Win Rate (WR):** The winning ratio of games completed under designated tasks.
- **Strategic Switch Frequency (SSF):** The average number of strategy intent p_t changes per match, which evaluates the strategic stability enhanced by the memory mechanism.
- **Hallucination Frequency (HF):** The average number of invalid or unexecutable actions generated per match.

- **Retry Success Frequency (RSF):** The average number of successful corrections by the sandbox mechanism per match, used to measure the effectiveness of the sandbox.

- **Decision Latency (DL):** The average end-to-end time required to complete a single decision step.

- **Token Cost (TC):** The average token consumption per game match.

4.2 Main Result

As shown in Table 1, we first evaluate our framework under the ECSB mode. Experimental results show that the cloud-based baseline collapses completely at Level 3 with the winning rate dropping to 0%, since it fails to maintain stable strategic coherence in long-duration confrontations. In comparison, our proposed CMHA obtains nearly 100% winning rates at low difficulty levels, reaches 50-60% winning rate at Level 4, and maintains usable performance until Level 6. The distinct performance gap fully demonstrates the strong reasoning capability of CMHA in complex scenarios.

When the tactical adapter employs Tactical-Phi, CMHA achieves win rates extremely close to those of the pure-cloud CMHA across all difficulty levels. Moreover, the performance remains poor when using the original Phi-2 validating the effectiveness of the locally fine-tuned model. However, when this locally fine-tuned model is applied to the baseline method, its win rates drop significantly across all levels. This is because the baseline adopts a single-layer architecture, and the small model causes the baseline to lose powerful inference capabilities. In contrast, the hierarchical design of CMHA compensates for this limitation.

Beyond macro-level operational control, we further evaluate the micro-operation capability of our architecture on the SMAC benchmarks in Table 2. CMHA obtains notably better KD and WR than baseline methods across all tasks. Notably, the baseline still maintains steady performance when integrated with locally fine-tuned models. This observation further confirms our previous analysis on hierarchical architecture design.

4.3 Analysis

Memory Mechanism To verify the effect of the counterfactual memory mechanism on strategic stability, we compare the full CMHA architecture

Method / WR	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
Baseline (4omini)	75.0%	16.7%	0.0%	0.0%	0.0%	0.0%	0.0%
Baseline (deepseek-v3)	58.3%	25.0%	0.0%	0.0%	0.0%	0.0%	0.0%
CMHA (4omini)	100.0%	100.0%	91.7%	50.0%	16.7%	8.3%	0.0%
CMHA (deepseek-v3)	100.0%	100.0%	100.0%	58.3%	8.3%	8.3%	0.0%
Baseline (Phi-2)	8.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Baseline (TP)	16.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
CMHA (4omini+Phi-2)	50.0%	33.3%	16.7%	0.0%	0.0%	0.0%	0.0%
CMHA (4omini+TP)	91.7%	91.7%	91.7%	41.7%	16.7%	8.3%	0.0%

Table 1: Win Rates across different difficulty levels. (TP denotes our fine-tuned Tactical-Phi model)

Method / KD (WR)	2s3z	3s5z	1c3s5z	3s5z_vs_3s6z	2c_vs_64zg	3s_vs_3z
Baseline (4omini)	0.66 (20%)	0.39 (0%)	1.01 (50%)	0.29 (0%)	0.54(0%)	0.09 (0%)
Baseline (TP)	0.70 (30%)	0.45 (5%)	0.82 (65%)	0.34 (0%)	0.61 (0%)	0.14 (5%)
CMHA (4omini+TP)	0.83 (35%)	0.56 (100%)	0.90 (80%)	0.21 (10%)	0.72 (0%)	0.17 (5%)
CMHA (4omini)	0.74 (35%)	0.67 (100%)	0.92 (85%)	0.26 (30%)	0.81 (5%)	0.23 (0%)

Table 2: Kill/Death Rates and Win Rates on SMAC benchmark tasks.

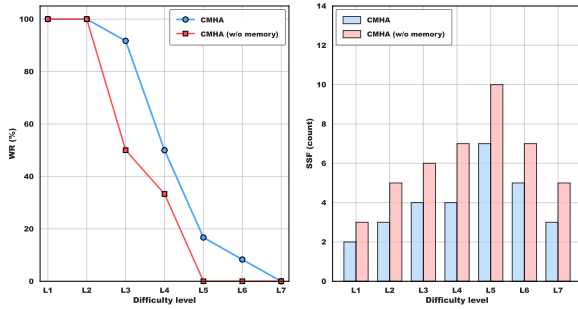


Figure 3: Impact of the counterfactual memory mechanism on win rate and strategic switch frequency.

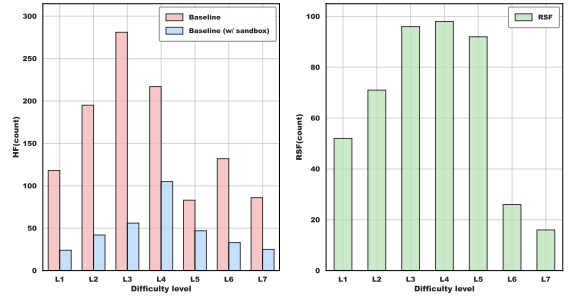


Figure 4: Self-correcting sandbox performance: hallucination frequency and retry success frequency.

with its variant with conventional memory mechanism, denoted as CMHA (w/o memory). As shown in Figure 3, both variants achieve 100% win rate at low difficulty levels. Their performance gap widens as difficulty increases. The memory-free variant drops to 50% win rate at Level 3 and becomes ineffective from Level 5 onward. In contrast, the full CMHA maintains effective combat capability up to Level 6. Furthermore, SSF data shows that the memory-free architecture exhibits significantly higher strategic switch counts at all difficulty levels. These results demonstrate that the counterfactual memory mechanism effectively prevents strategic jitter by dynamically filtering historical information.

Action Closed-loop Mechanism In this subsec-

tion, we compare the baseline and the baseline (w/ sandbox), and measure two metrics: hallucination frequency (HF) and retry success frequency (RSF). The experimental methodology mirrors that of the main experiment, involving macro-level matches of varying difficulty conducted on designated maps. As shown in Figure 4, the baseline (w/ sandbox) exhibits a substantial drop in HF across all difficulty levels. Meanwhile, the sandbox achieves nearly 100 correction successes per episode at Level 3 and Level 4, providing reliable guarantees for successful action execution.

Lightweight Model As verified in our main results, Tactical-Phi achieves performance comparable to the cloud-based large language model (GPT-4o-mini) when used as a tactical adapter. To further

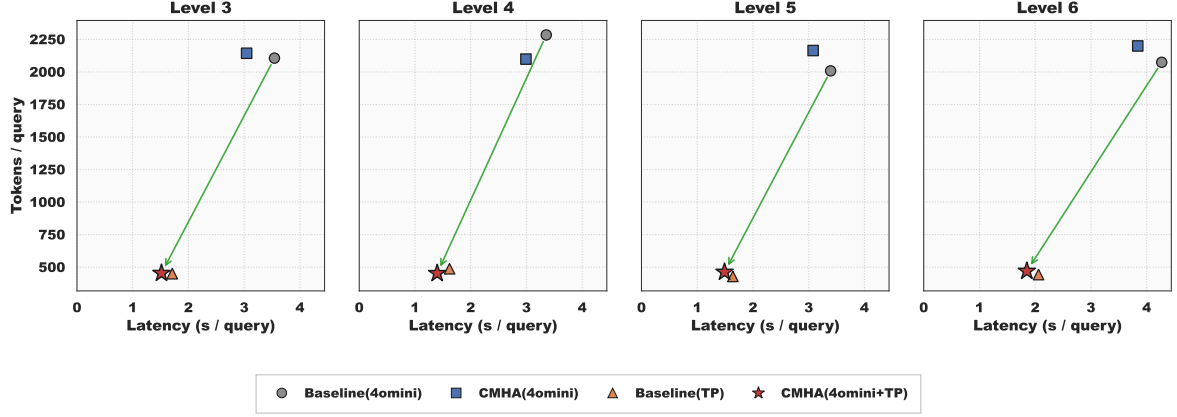


Figure 5: Comparison of decision latency and token consumption under various difficulty levels.

evaluate the efficiency advantage of CMHA, we measure decision latency and token cost across four settings in matches of Level 36 difficulty. As shown in Figure 5, both Baseline (4omini) and CMHA (4omini) exceed 3 seconds per query in latency, with token costs above 2000, especially at higher difficulty levels. In contrast, CMHA (4omini+TP) reduces average inference latency by approximately 57% and token consumption by nearly 80%. These results demonstrate that CMHA significantly lowers inference latency and token overhead without performance degradation, offering a deployment paradigm that balances efficiency and capability for RTS environments.

4.4 Case Study

We extract two cases from Level 5 match replays to compare practical behaviors between the CMHA agent and baseline methods.

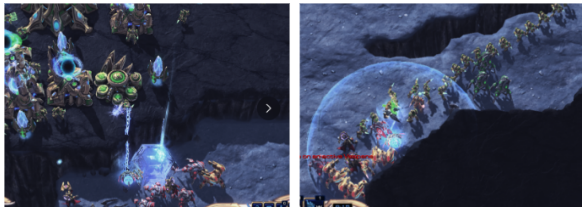


Figure 6: CMHA: Precision Defense and Strategic Counter-Attack.

As illustrated in Figure 6, the opponent launches a tentative raid on our main base with limited units. With the support of counterfactual memory, CMHA accurately identifies this action as a local tactical attack rather than a full-scale all-in strike. The agent keeps its original strategic stance instead of blind defense, and promptly assembles Stalker units to

counterattack the enemy base. By contrast, the baseline method shows obvious deficiencies under the same scenario.



Figure 7: Baseline: Strategic Jitter and Formational Collapse.

As shown in Figure 7, the baseline misjudges the tentative harassment as a full enemy offensive. It immediately abandons the unfinished strategic plan and switches to blind defense, which eventually leads to total defeat. The comparison demonstrates that CMHA delivers deeper reasoning and more stable strategic decision making.

5 Conclusion

We propose CMHA, a novel hierarchical architecture with counterfactual causal memory designed for StarCraftII. This architecture is driven by two core components, namely the Strategy Pulse Chain (SPC) brain and the lightweight Tactical-Phi model. The SPC manages memory mechanisms and adopts SP-CoT to enhance sufficient depth and stability of reasoning. Experimental results show that our method outperforms baseline models consistently, and significantly reduces inference latency and token consumption, thus exhibiting strong deployment potential.

Limitations

Despite the strong performance of the CMHA architecture in complex environments, its fixed sliding-window-based event segmentation mechanism may exhibit limited flexibility in capturing highly dynamic event boundaries. This mechanism may fragment continuous in-game events during matches, which deserves warrants further investigation in future studies. Furthermore, all experimental evaluations in this work are conducted exclusively on StarCraft II, and the generalization of the proposed method to other task types and application domains has yet to be fully explored.

References

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Junzhe Chen, Xuming Hu, Shuodi Liu, Shiyu Huang, Wei-Wei Tu, Zhaofeng He, and Lijie Wen. 2024. Llmarena: Assessing capabilities of large language models in dynamic multi-agent environments. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13055–13077.
- Michele Colledanchise and Petter Ögren. 2018. *Behavior trees in robotics and AI: An introduction*. CRC Press.
- DeepSeek-AI. 2025. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115.
- Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pages 9118–9147. PMLR.
- Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.
- Muhammad Junaid Khan and Gita Sukthankar. 2024. Sc-phi2: A fine-tuned small language model for starcraft ii build order prediction. *AI*, 5(4):2338–2352.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.
- Yu Li and 1 others. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Technical Report*.
- Zhiyuan Li, Wenshuai Zhao, and Joni Pajarinen. 2025. Cooperative multi-agent planning with adaptive skill synthesis. *arXiv preprint arXiv:2502.10148*.
- Zongyuan Li, Yanan Ni, Runnan Qi, Chang Lu, Lumin Jiang, Xu Xiaojie, Xiangbei Liu, Pengfei Li, Yunzheng Guo, Zhe Ma, and 1 others. 2026. Llm-pysc2: Starcraft ii learning environment for large language models. *Advances in Neural Information Processing Systems*, 38:127595–127657.
- Weiyu Ma, Qirui Mi, Yongcheng Zeng, Xue Yan, Yuqiao Wu, Runji Lin, Haifeng Zhang, and Jun Wang. 2024. Large language models play starcraft ii: Benchmarks and a chain of summarization approach. *Advances in Neural Information Processing Systems*, 37:133386–133442.
- OpenAI. 2024. Gpt-4o mini. <https://openai.com/index/gpt-4o-mini>.
- Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. Memgpt: towards llms as operating systems.
- Joon Sung Park, Joseph O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.
- Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. 2019. The starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*.
- Xiao Shao, Weifu Jiang, Fei Zuo, and Mengqing Liu. 2024. Swarmbrain: Embodied agent for real-time strategy game starcraft ii via large language models. *arXiv preprint arXiv:2401.17749*.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko

Georgiev, and 1 others. 2019. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, and 1 others. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Huikai Wu, Yanqi Zong, Junge Zhang, and Kaiqi Huang. 2017. Msc: A dataset for macro-management in starcraft ii. *arXiv preprint arXiv:1710.03131*.

Yue Wu, Xuan Tang, Tom Mitchell, and Yuanzhi Li. 2024. Smartplay: A benchmark for llms as intelligent agents. In *International Conference on Learning Representations*, volume 2024, pages 1543–1561.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Yixin Zhang, Ziyi Wang, Yiming Rong, Haoxi Wang, Jinling Jiang, Shuang Xu, Haoran Wu, Shiyu Zhou, and Bo Xu. 2026. World models for policy refinement in starcraft ii. *arXiv preprint arXiv:2602.14857*.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731.

A Implementation Details of the Strategic Pulse Chain Brain

The strategic layer in our implementation is instantiated by a shared StrategicPulseBrain (SPC), rather than by a single monolithic prompt. At each strategic tick, the final input to the top-level model is composed from three sources: (i) a fixed system prompt in query_strategic, (ii) a dynamic macro prompt assembled in StrategicPulseBrain.maybe_run_tick, and (iii) an optional doctrine block injected by the full-game experiment script. This design keeps the high-level reasoning policy stable while allowing the prompt to incorporate the latest battlefield context and long-term memory.

A.1 Fixed Strategic Prompt

The fixed system prompt defines the top-level model as the macro-strategic module for StarCraft II under the following default setting:

- map: Simple64;
- our race: Protoss;
- opponent: built-in AI.

The prompt further instructs the model to recall a set of common Protoss plans, including 4-Gate rush, Stalker-Sentry pressure, Immortal all-in, Chargelot-Archon timing, fast expand into macro, Colossus deathball, Dark Templar harassment, Oracle opening, and Void Ray rush. Strategy selection is conditioned on four factors: map layout and expansion geometry, opponent race, current game stage and resource status, and known tendencies of the built-in StarCraft II AI.

The system prompt also enforces three high-level decision rules. First, the agent should not overcommit to passive defense; if our army is already active while the opponent is still scaling, the model should prefer attack or pressure. Second, if resources are floating, the model should spend them on units, production capacity, expansions, or technology rather than delaying action. Third, if an initial plan has clearly failed, the strategic module should switch to a new plan instead of preserving an invalid macro intention.

The final response is required to contain the following tagged fields:

- [Memory]** a JSON object, preferably with keys such as enemy_race_guess, enemy_tech_guess, our_plan, risks, and open_tasks; each key may carry value, confidence, and ttl.
- [Pulse]** one of ATTACK, DEFEND, EXPAND, or RETREAT.
- [Pivot]** a short English description of the tactical focus.
- [Horizon]** an integer indicating how many in-game seconds the recommendation is expected to remain valid.

The model may produce free-form reasoning before these fields, but the tagged fields are mandatory.

A.2 Dynamic Macro Prompt

At every top-level tick, SPC appends a dynamic macro prompt that restricts the output to the following slots:

- [Pulse]**
- [Pivot]**

771
772
773
774

775
776
777
778
779
780

781
782
783
784
785
786

787
788
789
790
791
792
793

794
795

796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817

[CommanderIntent]
[DeveloperIntent]
[Horizon]
[Memory]

This prompt explicitly forbids the model from outputting executable action names. Instead, the top-level model only produces macro intent and memory updates.

The dynamic context includes the current strategic state:

- current_pulse;
- current_pivot;
- pulse_inertia_h_t;
- current_commander_intent;
- current_developer_intent;
- last_decision_horizon_s.

It then appends [SPCLongTermMemoryPrev], which contains the surviving long-term memory slots from the previous strategic tick, together with the full CommanderObservationText. If enabled, the prompt can further include [CommanderExecutionFeedback], [StrategicDoctrine], and [Playbook] blocks.

A.3 Strategic Doctrine Used in the Full-Game Setting

In the default full-game experiment, the strategic doctrine is enabled and provides map-specific guidance for Protoss play on Simple64. The doctrine assumes up to three base locations and up to four Assimilators. It also reminds the model to monitor Nexus energy and to use Chrono Boost on key structures.

The doctrine exposes two default candidate plans. The first is a *Zealot and Stalker Tactic*, framed as a strong all-in window within the first eight in-game minutes. Its core structure is based on Warp Gate production, with one Cybernetics Core, one Twilight Council, and one Forge, together with Warp Gate and Charge as key technologies. The recommended composition is approximately Zealot:Stalker = 1:1 or 2:1, with timing anchors such as two Nexuses and two Gateways by 2:00, three to four Warp Gates with a Cybernetics Core by 4:00, and four to five Warp Gates plus Twilight Council and the key upgrades by 8:00.

The second is a *Carrier Tactic*, framed as a more stable late-game plan. It recommends Zealot plus

Stalker defense before 6:00 and Carrier-centered play afterwards, supported by up to four Stargates, one Fleet Beacon, and air upgrades. Representative timing anchors include two Nexuses and two Gateways by 2:00, two Stargates plus Fleet Beacon and the first Carrier in production by 6:00, and three to four Carriers with four Stargates by 10:00.

The doctrine also encodes several global economic heuristics. For example, it prefers a second Nexus when the worker count exceeds 16, a third Nexus when the worker count exceeds 35, two Assimilators after 25 workers, and four Assimilators after 45 workers. It keeps producing Probes until the worker count reaches 80, adds a Pylon whenever remaining supply is below four, prioritizes missing technology structures for the chosen plan, and dispatches a Probe scout when the economy is comfortable. In the Carrier plan, Chrono Boost is prioritized for Stargates first, then key technologies, and then Nexuses. Finally, the doctrine discourages attacks below 60 army supply, allows attacks once army supply reaches 60, and strongly favors aggression once army supply reaches 80.

A.4 Operationalization of Aggregation, Admission, and Pruning

The four-stage description in Section 3.2 should be understood as a conceptual abstraction of the released implementation. Concretely, these functions are realized through the interaction between the fixed strategic prompt, the dynamic macro prompt, structured long-term memory, and the surrounding control logic, rather than through four separately named prompt headers.

Observation aggregation. In the released implementation, macro-level aggregation is realized by concatenating the current strategic state, the surviving long-term memory, and the latest CommanderObservationText. If enabled, the prompt additionally includes execution feedback, doctrine, and playbook information. In the default full-game configuration, command feedback is disabled, playbook mode is disabled, and strategic doctrine is enabled.

Counterfactual admission. In the released code, the admission decision is implemented implicitly rather than through a separately labeled prompt field. Operationally, the strategic module is asked to preserve or revise the current plan based on whether the newly observed change is strategically consequential. In particular, the fixed system

818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840

841
842

843
844
845
846
847
848
849
850

851
852
853
854
855
856
857
858
859
860

861
862
863
864
865
866
867

prompt requires the model to change plans once the current one has failed. In the optional playbook mode, an interrupt override can also force a temporary switch to DEFEND when enemy pressure rises abruptly, after which the system may return to the original playbook. This control pattern serves as the concrete implementation of the counterfactual admission behavior described in the main text.

Semantic pruning. Long-term memory is not accumulated indefinitely. The memory ingester only accepts non-empty key-value entries, while confidence and ttl are clamped to valid ranges. At each strategic tick, the decay procedure decrements ttl by one and removes entries whose ttl falls below the threshold. Only the surviving slots are rendered back into [SPCLongTermMemoryPrev] for the next prompt. In this sense, the pruning mechanism is implemented as structured memory filtering and time-to-live decay.

Macro reasoning. The top-level system prompt provides the general macro-strategic principles, the doctrine block injects concrete Protoss rules for Simple64, and the dynamic prompt forces the model to emit only the macro-level slots Pulse, Pivot, CommanderIntent, DeveloperIntent, Horizon, and Memory. This combination operationalizes the macro reasoning stage of CMHA.

A.5 Default Full-Game Strategic Configuration

The default full-game launcher enables the strategic layer with the following switches:

- ENABLE_SPC_STRATEGIC_LAYER = True;
- SPC_SHARED_STRATEGIC_BRAIN = True;
- SPC_STRATEGIC_OWNER_AGENT = Commander;
- SPC_STRATEGIC_INTERVAL_GAME_SECONDS = 15.0;
- COMMANDER_LLM_INTERVAL_GAME_SECONDS = 5.0;
- SPC_INCLUDE_COMMANDER_FEEDBACK = False;
- script = False.

Therefore, the default setup uses the shared SPC strategic layer together with the doctrine block, but without command feedback and without playbook-mode overrides.

B Additional Details of Tactical-Phi Fine-Tuning

B.1 Base Model and Two-Stage Objective

Tactical-Phi is built on microsoft/phi-2, whose local checkpoint corresponds to the PhiForCausalLM architecture. We train the model in two stages. Stage 1 performs StarCraft II domain adaptation on an SC2 text corpus so that the base model first acquires SC2-specific text understanding, Protoss macro templates, and basic instruction-following ability. Stage 2 then turns the model into a developer-side tactical-action mapper for LLM-PySC2: the input is a local macro summary together with heuristic tactical guidance, and the output is the next four legal Protoss macro actions in alias form. During deployment, the alias actions are deterministically mapped back to the original action templates before legality filtering and execution.

B.2 Stage 1 and Stage 2 Hyperparameters

Table 3 summarizes the main training settings.

In Stage 2, we apply output-only masking: tokens corresponding to the observation and context side are assigned label -100, and the loss is computed only on the target alias action sequence. This choice encourages the model to focus on action generation rather than reconstructing the textual observation.

B.3 Instruction-Action Dataset Construction

The instruction-action data are derived from human-played replays distributed through the MSC dataset. Our Protoss pipeline uses feature files from MSC-master, rather than directly parsing raw SC2Replay files. We keep only the Protoss perspective and use three matchup families: Protoss versus Protoss, Protoss versus Terran, and Protoss versus Zerg.

The input is not the original tensor state. Instead, we convert GlobalFeatureVector and SpatialFeatureTensor into a textual macro summary that describes economy, production, technology, and local battlefield context. The output side is constructed in an event-driven manner: from the current step, the pipeline scans forward and

Setting	Stage 1	Stage 2
Base checkpoint	microsoft/phi-2	microsoft/phi-2 + Stage 1 adapter
Loading precision	8-bit quantized	8-bit quantized
Objective	SC2 domain adaptation	instruction-action alignment
LoRA rank	32	96
LoRA alpha	64	192
LoRA dropout	0.05	0.05
Target modules	Wqkv, fc1, fc2	Wqkv, fc1, fc2, out_proj
Optimizer	adamw_8bit	adamw_8bit
Learning rate	2.5×10^{-5}	5×10^{-5}
Per-device batch size	1	1
Gradient accumulation	8	8
Effective batch size	8	8
Epochs	160	2
Maximum sequence length	820	256
LR scheduler	cosine	constant
Warmup steps	80	—
Mixed precision	bf16	bf16

Table 3: Training hyperparameters for Tactical-Phi. Both stages use parameter-efficient fine-tuning in a QLoRA-style setup on top of an 8-bit loaded base model.

collects the next four legal Protoss macro action events. Only windows that contain four valid future actions are retained.

Several filtering rules are used during construction. Unsupported actions are skipped. Exact low-information repetitions are suppressed conservatively, while reasonable consecutive build actions are preserved. The current version also injects a short heuristic commander guidance string that summarizes the macro situation for the developer-side model. Finally, the output is written in alias-action form rather than raw Build_/Train_/Research_ templates, which reduces the dominance of shared prefixes in the training loss.

After generating the full Stage 2 corpus, we build a higher-quality subset according to state matching and short-window continuity. An additional first-action cap is applied only to the training split; validation and test splits keep their natural distribution. The final developer-alignment dataset used in our current experiments is `protoss_macro_retrain_v4_alias_hq12h`.

B.4 Dataset Statistics

Table 4 reports the dataset sizes before and after high-quality filtering. The validation and test splits remain unchanged, while the training split is filtered more aggressively.

For the final `v4 alias_hq12h` subset, the reward distribution is as follows. The training split contains 44,801 winning instances and 7,645 losing instances; the validation split contains 44,142 winning instances and 7,653 losing instances; the test

Dataset	Train	Val	Test
v4 alias	212,772	51,795	109,818
v4 alias_hq12h	52,446	51,795	109,818

Table 4: Dataset sizes for the full Stage 2 corpus and the final high-quality subset.

Alias action	Original template
B:Pylon(screen)	<Build_Pylon_Screen(screen)>
B:Assimilator(tag)	<Build_Assimilator_Near(tag)>
T:Probe()	<Train_Probe()>
R:WarpGate()	<Research_WarpGate()>

Table 5: Examples of alias actions used during Stage 2 training.

split contains 92,511 winning instances and 17,307 losing instances.

B.5 Alias Actions

Table 5 shows representative alias actions and their corresponding original templates.

B.6 Hardware

Our reproduction environment uses four NVIDIA GeForce RTX 3090 GPUs with 24 GB memory each. In practice, a single training run typically occupies one GPU, and the 8-bit model loading is used to control memory consumption.

C Experimental Protocol

C.1 Full-Game Evaluation

The full-game launcher executes one selected difficulty level per invocation, rather than sweeping all difficulties automatically. The difficulty is controlled by the environment variable `SC2_DIFFICULTY_LEVEL`, whose default value is 3

(medium). The available levels are 1: `very_easy`, 2: `easy`, 3: `medium`, 4: `medium_hard`, 5: `hard`, 6: `harder`, 7: `very_hard`, 8: `cheat_vision`, 9: `cheat_money`, and 10: `cheat_insane`. The default `MAX_EPISODES` is 5. The main-paper results focus on Levels 1–7; these are obtained by rerunning the same script under different difficulty settings.

C.2 SMAC Evaluation

The SMAC launcher defaults to `3s_vs_3z`. It does not explicitly pass `-max_episodes` to `pysc2.bin.agent`, whose default is one episode. The map wrapper in `llm_smac.py` sets `game_steps_per_episode = 22 * 60 * 30`, which corresponds to a 30-minute cap in game steps. The SMAC maps reported in the main paper are obtained by rerunning the same pipeline with the corresponding map names.

C.3 Model and Inference Settings

The top-level cloud model is configured as `gpt-4o-mini`. The local code and logs identify the model name, but they do not record a provider-specific snapshot ID or dated revision. The inference temperature is fixed to 0.1. The code does not explicitly set `max_tokens` or the context length in the chat-completion calls, so both remain at provider defaults.

Several additional defaults affect runtime behavior. `MAX_LLM_QUERY_TIMES` is 5. In the full-game launcher, `MAX_LLM_WAITING_TIME` is increased from 15 to 90 seconds, and `MAX_LLM_RUNTIME_ERROR_TIME` is increased from 45 to 120 seconds. The global game-time conversion uses `SC2_GAME_LOOPS_PER_SECOND = 22.4`. Both the full-game and SMAC launchers run with `-parallel 1`.

C.4 Randomness

Although `pysc2.env.sc2_env.SC2Env` supports a `random_seed` argument, neither `experiment_full_game.py` nor `experiment_llm_smac.py` passes an explicit seed. Consequently, the default launchers do not fix random seeds.

D Action Sandbox and Self-Correction

D.1 Retry Budget

The tactical sandbox allows up to three self-correction attempts. This retry budget is distinct

from the API-layer retry mechanism of the LLM client, which may retry model queries up to five times.

D.2 Legality Checks

The sandbox first extracts valid action names from the current *Valid Actions* text with regular expressions. Communication actions such as `MessageTo` and `ListenTo` are not treated as game actions. A candidate output is rejected under the following conditions:

- the output is empty;
- the output contains no bracketed action of the form `<...>`;
- when a non-empty valid-action whitelist is available, at least one game action name is outside the whitelist;
- the output contains only `MessageTo` or `ListenTo` while the current agent is not the Commander.

If the valid-action whitelist is empty, the sandbox performs only format checking.

D.3 Strategic Context Injected During Retries

Before each retry, the sandbox prepends a *Tactical Decision Context (SPC)* block to the observation. This block always includes the current strategic pulse, tactical pivot, strategic inertia h_t , and action feedback, together with an explicit instruction that action names must match the subsequent valid-action list exactly.

If the current agent is the Commander, the retry prefix additionally includes the SPC-level `CommanderIntent` and `DeveloperIntent`. The Commander is instructed to align with the `CommanderIntent` and to send exactly one `MessageTo` command to the Developer while preserving the semantics of the `DeveloperIntent`. If the current agent is the Developer, the prefix injects several resource-spending heuristics, such as continuously converting high minerals into army and production capacity, avoiding prolonged mineral floating before supply blocks, and prioritizing unit production, core production buildings, expansion, and ground upgrades when resources are abundant.

If the previous attempt failed, the sandbox appends a `[RetryFeedback]` block that explains why the prior output was illegal.

D.4 Fallback Policy

The final fallback is not a single hard-coded action. The system first tries a Developer-side forced-production fallback in the following priority order:

```
WarpTrain_Stalker → WarpTrain_Zealot
→ Train_Stalker → Train_Zealot
→ Build_Gateway → Build_Nexus
→ Research_ProtossGroundWeapons
→ Research_ProtossGroundArmor →
Research_WarpGate → Build_Assimilator
```

If script mode is enabled, Train_Immortal may also be inserted into this list.

If no Developer-specific fallback applies, the system selects a pulse-conditioned safe action: <All_Units_Defend()> under DEFEND when available, <All_Units_Retreat()> under RETREAT when available, otherwise <All_Units_Attack()>, then <Hold_Position()>, then <Stop()>, and finally <No_Operation()>.

D.5 Observed Self-Correction Example

A representative failure-and-recovery case occurs on the Developer side. In one logged interaction, the model first proposed <Build_Probe()>. The sandbox rejected it because the action name did not appear in the current *Valid Actions* list. The retry prompt then injected the same error message into [RetryFeedback]. On the next attempt, the model corrected its output to <Build_Pylon_Screen(screen)>. This example illustrates that the sandbox enforces an exact whitelist over currently available action names, and that self-correction is driven by explicit legality feedback rather than by silent post-processing.