SC2ARENA: EVALUATING LARGE LANGUAGE MODELS IN FULL GAMES OF STARCRAFT II

Anonymous authors
Paper under double-blind review

ABSTRACT

Evaluating large language models (LLMs) as agents in competitive environments requires examining how they pursue objectives while responding to opponents whose actions can disrupt their progress. StarCraft II provides a challenging setting where agents must coordinate resource management, production, and unit control against an opponent over a complete match. We introduce SC2Arena, a benchmark for evaluating LLMs in full games of StarCraft II through a standardized text interface. SC2Arena supports all three playable races, fine-grained actions, and matches against both built-in AI and other agents. It combines game outcomes with process metrics to assess action validity, resource management, and token consumption, and incorporates unit ordering and aggregation to structure textual observations. We also develop a hierarchical baseline that separates command generation from action execution, incorporates iterative verification, and supports supervised fine-tuning on selected gameplay data. Experiments reveal substantial performance differences across evaluated models. Within the tested settings, the hierarchical baseline with verification improves win rates and action validity, while fine-tuning yields further gains. These results highlight the importance of considering agent design alongside model choice when evaluating LLMs in competitive environments.

1 INTRODUCTION

As large language models (LLMs) are increasingly deployed as interactive agents (Yao et al., 2023; Liu et al., 2024), evaluating their ability to act effectively in external environments has become an important research problem. Prior work in web navigation (Deng et al., 2023; **?**), embodied tasks (Shridhar et al., 2020), and open-world environments (Wang et al., 2023) examines how models use observations and feedback to complete tasks. We focus on a complementary question: *how effectively can LLM-based agents pursue their objectives against an opponent whose actions can disrupt their progress?* In competitive environments, the effectiveness of a decision depends on both the agent's behavior and the opponent's response. When competition also involves interconnected activities such as resource acquisition, production, and combat, success requires coordinating these activities as the interaction unfolds.

StarCraft II provides a concrete setting for this evaluation (Vinyals et al., 2017). Over a complete match, an agent must manage an economy, construct production facilities, develop an army, and control units while responding to an opponent under partial observability (Vinyals et al., 2019). These activities are closely coupled: resource allocation constrains production, production determines the available forces, and combat changes the resources and units available for subsequent decisions. A locally feasible action may therefore be ineffective in the broader context of a match—for example, spending resources on additional production facilities may leave insufficient resources for immediate defense. Evaluating complete matches places these interdependent decisions within a shared task, with success measured by the final game outcome.

Prior work has established several approaches to evaluating agents in StarCraft II. SMAC (Samvelyan et al., 2019) focuses on cooperative unit control in combat scenarios, while TextStarCraft II (Ma et al., 2024b) and LLM-PySC2 (Li et al., 2024) enable language-model-based interaction through different interfaces and action abstractions. Building on these efforts, we aim to evaluate how LLM-based agents coordinate resource management, production, and unit control over complete matches against

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.

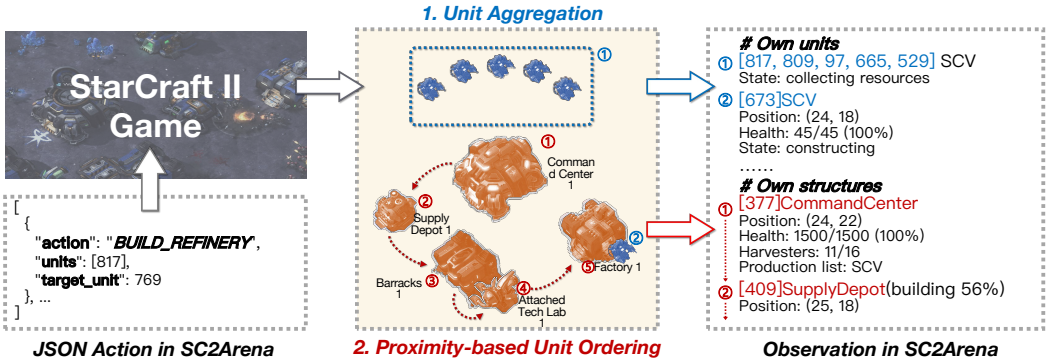


Figure 1: Overview of SC2Arena's action interface and observation representation. (a) Actions use a standardized JSON format that allows agents to specify acting units and, when required, target units or positions. (b) Game observations are converted into text using unit aggregation to reduce repeated descriptions and proximity-based unit ordering to organize unit descriptions by spatial proximity.

built-in AI and other agents. Such evaluation must account for both the underlying model and the mechanisms through which it observes and acts. This calls for a common interface that supports comparisons of models and agent configurations under shared game settings.

To this end, we introduce SC2Arena, a benchmark for evaluating LLM-based agents in full games of StarCraft II. Its structured textual observations and JSON action interface allow agents to specify acting units, structures, and targets explicitly. The interface supports Terran, Protoss, and Zerg, with matches against built-in AI and other agents. Proximity-based unit ordering and unit aggregation organize textual observations, while outcome and process metrics describe competitive performance, action validity, resource management, and output-token consumption.

Our contributions are threefold. First, we provide a unified text-based interface for full-game evaluation across all three races, with explicit unit-level action specification and support for agent-versus-agent matches. Second, we provide an evaluation protocol combining match outcomes with execution, resource-management, and output-token metrics. Third, we provide a hierarchical reference agent with verification and supervised fine-tuning, and evaluate models and agent configurations across maps, race matchups, observation representations, and component ablations.

2 Related Work

Evaluation of LLM-based Agents. LLM-based agents have been studied in environments that require models to interpret observations, generate actions, and use feedback. AgentBench evaluates agents across multiple interactive environments (Liu et al., 2024; Xie et al., 2024), while Mind2Web and WebArena focus on web tasks (Deng et al., 2023; Zhou et al., 2023; Drouin et al., 2024). ALFWorld connects text-based interaction with embodied tasks (Shridhar et al., 2020), and Voyager studies an LLM-based agent that acquires skills through interaction with an open-world environment (Wang et al., 2023). Competitive and social settings provide another dimension of evaluation: AvalonBench examines agents in a game involving communication, cooperation, and conflicting objectives (Light et al., 2023; Chezelles et al., 2024). SC2Arena complements these settings by evaluating agents in full StarCraft II matches, where resource management, production, and unit control jointly determine performance against an opponent.

Agent evaluation also depends on the mechanisms through which models interact with their environments (Ma et al., 2024a; Paglieri et al., 2025; Lanctot et al., 2019; Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024). ReAct interleaves reasoning and actions with environmental feedback (Yao et al., 2023), while SWE-agent demonstrates the importance of interface design for agent performance (Yang et al., 2024). AI Agents That Matter emphasizes evaluating computational cost alongside task performance (Kapoor et al., 2024). These perspectives motivate our use of both outcome and process metrics, together with comparisons of observation representations and agent configurations.

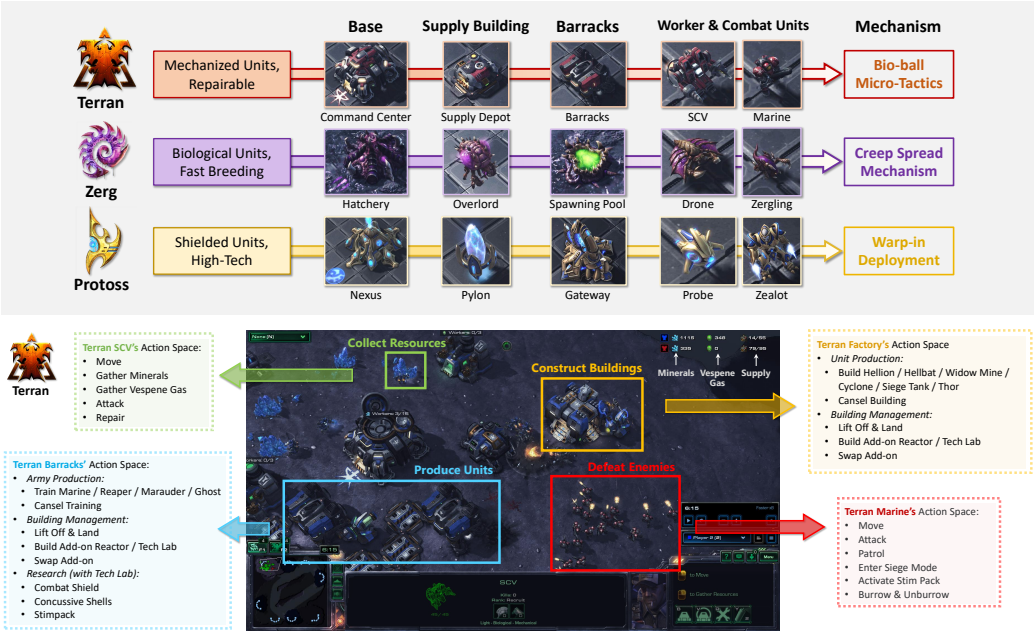


Figure 2: Overview of StarCraft II races and gameplay. Terran, Zerg, and Protoss feature distinct units, structures, and mechanics. During a match, players coordinate resource gathering, building construction, unit production, and combat against an opponent. The illustrated action examples show the different commands available to individual units and structures.

StarCraft II Environments and Agents. StarCraft II has served as a testbed for decision-making under partial observability with a large state and action space (Vinyals et al., 2017). SMAC focuses on cooperative multi-agent control in combat scenarios (Samvelyan et al., 2019), while AlphaStar demonstrates full-game play using multi-agent reinforcement learning (Vinyals et al., 2019).

More recent work enables language models to interact with StarCraft II. TextStarCraft II provides a textual environment and a Chain of Summarization approach for game interaction (Ma et al., 2024b). LLM-PySC2 exposes the StarCraft II action space through language-model-compatible interfaces and supports multimodal observations (Li et al., 2024). HIMA combines specialized imitation-learning agents with a strategic planner and introduces TEXTSCII-ALL to cover all race matchups (Ahn et al., 2025). SC2Arena builds on this line of research by combining full-game interaction, fine-grained JSON actions, support for all three races, and matches against both built-in AI and other agents within a common text interface. We compare models and agent configurations using match outcomes and process metrics, and conduct ablation studies on observation representation and agent design.

3 BENCHMARK: SC2ARENA

SC2Arena provides a text-based interface for evaluating LLM-based agents in full games of StarCraft II. It supports all three playable races, fine-grained actions, and matches against built-in AI or other agents. The benchmark combines match outcomes with process metrics to characterize competitive performance, resource management, action validity, and token consumption. We describe the interaction framework, game environment, evaluation metrics, and relationship to existing benchmarks below.

3.1 FRAMEWORK

SC2Arena connects LLM-based agents to StarCraft II by converting game observations into text and translating structured model outputs into game commands. This interface allows different models and agent architectures to interact with the same environment. Two observation-design considerations arise in this setting. First, unit positions are represented numerically, while their

spatial relationships may be difficult to identify from an unordered list. Second, describing every unit separately can produce lengthy observations containing repeated information. We therefore incorporate two mechanisms for organizing textual observations, as illustrated in Fig. 1.

Proximity-based unit ordering. We arrange unit descriptions using a greedy nearest-neighbor traversal. Starting from an initial unit, the procedure repeatedly selects the unvisited unit nearest to the most recently added unit and appends its description to the sequence. This ordering tends to place spatially nearby units close together in the text, providing a spatial organization in addition to explicit coordinates.

Unit aggregation. We group units with shared attributes into a single description rather than repeating the same information for each unit. For example, workers performing the same resource-collection activity can be represented together with their unit IDs. This representation reduces repeated descriptions while retaining identifiers for subsequent action specification. We evaluate the effects of ordering and aggregation through ablation experiments.

Interaction loop. At each decision point, SC2Arena converts the current game observation into text and passes it to the agent. The agent returns actions in JSON format, which the interface parses and translates into game commands. The game then advances, and subsequent observations reflect the resulting state. The interface is separate from the agent's internal architecture: direct action generation and the hierarchical baseline described in Section 4 use the same action format. Appendix A.3 provides implementation details.

3.2 Environment

Full-game competitive interaction. StarCraft II is a real-time strategy game in which players manage resources, construct buildings, produce units, and engage an opponent under partial observability (Vinyals et al., 2017; 2019). As illustrated in Figs. 6 and 2, these activities are interdependent: economic decisions constrain production, and combat changes the units and resources available for subsequent actions. SC2Arena supports matches from the initial game state to termination, allowing agents to be evaluated on the combined effects of these decisions. Opponents can be built-in AI or other agents, supporting both evaluation against a common reference opponent and direct comparisons between agents.

Race coverage. SC2Arena supports Terran, Protoss, and Zerg. These races differ in their units, production systems, and available abilities, introducing different observation and action requirements. Supporting all three races allows evaluation across these game mechanics rather than restricting the interface to a single race. Appendix A.1 describes the relevant mechanics in more detail. The maps, race matchups, and opponent settings used in our experiments are specified in the experimental setup.

Observation and action interface. As shown in Fig. 1, observations are organized into textual descriptions of resource levels, unit states, building statuses, ongoing activities, visible enemy units and structures, and technological progress. Enemy observations are restricted by in-game visibility. Actions use a unified JSON format to specify operations such as unit production, structure construction, movement, and attacks. Each action identifies the relevant units or structures and, when required, a target unit or location. This format supports fine-grained command specification within the text interface. Appendix A.2 provides examples of observations and actions.

3.3 Multi-Dimensional Metrics

SC2Arena reports match outcomes together with process metrics. Outcome metrics summarize competitive performance, while process metrics describe resource management, execution validity, and token consumption.

Match outcomes. (1) *Elo Rating (ELO)* summarizes relative performance within an evaluated agent pool using agent-versus-agent match results. (2) *Win Rate (WR)* is the percentage of games won under a specified opponent and game configuration. (3) *Time Cost of Winning (TCW)* is the average duration of winning games, measured in game seconds. Because TCW is conditioned on victory, it is reported alongside win rate rather than interpreted independently.

Resource management. (4) *Supply Block Ratio (SBR)* is the fraction of game time spent at the supply cap, which constrains further production of units requiring additional supply. (5) *Resource Spending Rate (RSR)* measures the combined mineral and vespene expenditure per game second.

Execution and token consumption. (6) *Tokens Per Decision (TPD)* measures the average number of generated tokens per decision. It characterizes output token consumption rather than total inference cost or wall-clock latency. (7) *Valid Action Ratio (VAR)* measures the proportion of generated actions that satisfy the environment's execution constraints. A valid action need not be effective, so VAR is interpreted together with match outcomes. Detailed formulations are provided in Appendix A.4.

3.4 Comparison with Existing Benchmarks

Existing StarCraft II environments provide different combinations of gameplay scope, action abstraction, race coverage, and opponent support. SMAC focuses on cooperative unit control in combat scenarios (Samvelyan et al., 2019). TextStarCraft II provides a textual interface for game interaction through high-level actions (Ma et al., 2024b), while LLM-PySC2 exposes StarCraft II actions through language-model-compatible interfaces and supports multimodal observations (Li et al., 2024). TEXTSCII-ALL supports full-game evaluation across all race matchups, including matches against built-in AI and other agents (Ahn et al., 2025). Compared with its macro-action interface, SC2Arena allows explicit specification of acting units and target units or positions.

Table 1 compares the evaluation capabilities of these platforms. SC2Arena combines full-game interaction, fine-grained JSON actions, support for all three races, and matches against built-in AI or other agents within a common text interface. This combination supports comparisons of models and agent configurations using both competitive outcomes and process measurements.

Table 1: Comparison of StarCraft II benchmarks in terms of full-game support, race coverage, agent-versus-agent evaluation, and unit-level action control. Unit-level action control denotes whether the model can explicitly specify the acting units and their target units or positions.

Benchmark	Agent-vs-agent	Unit-level action control	All races	Full-game support
SMAC (Samvelyan et al., 2019)	✗	✓	✗	✗
LLM-PySC2 (Li et al., 2024)	✗	✓	✗	✓
TextStarCraft II (Ma et al., 2024b)	✗	✗	✗	✓
TEXTSCII-ALL (Ahn et al., 2025)	✓	✗	✓	✓
SC2Arena (ours)	✓	✓	✓	✓

4 Baseline

The fine-grained action interface in SC2Arena requires agents to determine both what to do and how to specify the corresponding game commands. For example, constructing a building involves selecting the building type, an available worker, and a target location, while satisfying resource and action constraints. We provide Baseline as a reference baseline that separates command generation from action specification and incorporates verification at both stages. We also describe a supervised fine-tuning procedure using selected gameplay data. Together, these components support experiments on how agent design and training affect performance within the same benchmark interface. Figure 3 illustrates the framework.

4.1 Hierarchical Self-Correction Framework

Command generation and action specification. Given an observation o from SC2Arena, the ***Planner*** generates natural-language commands c, such as "Build a Barracks." The ***Executor*** receives both the observation o and the commands c, and translates them into JSON actions a. This translation includes selecting acting units or structures and specifying target units or positions when required. The hierarchy therefore separates the choice of commands from the details needed to express them through the game interface.

Iterative verification. We incorporate verification into both stages. The Planner's ***Verifier*** uses an LLM to assess proposed commands against the current observation and the supplied rules, returning

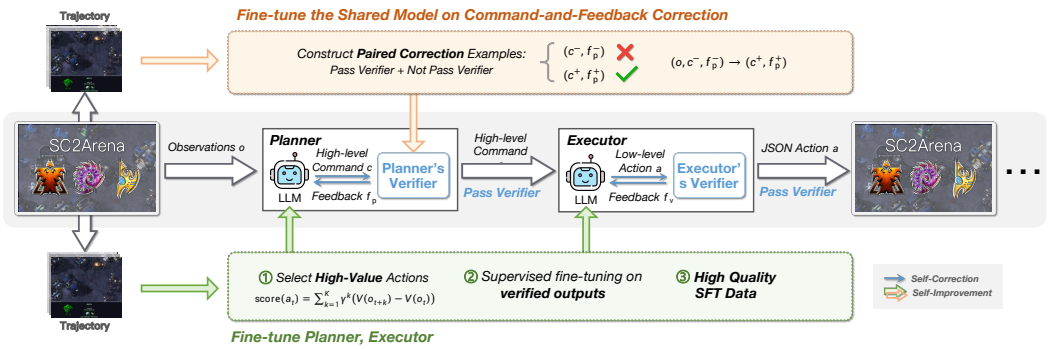


Figure 3: Overview of Baseline. The Planner generates natural-language commands, and the Executor translates them into JSON actions. An LLM-based Planner verifier and a programmatic Executor verifier provide feedback for iterative self-correction. The shared language model is fine-tuned using selected gameplay data, including verifier-accepted outputs and paired command-and-feedback correction examples.

structured feedback f_{p} for revision. The Executor's verifier is implemented programmatically and checks JSON syntax, action parameters, and execution constraints such as resource availability and target validity. Detected errors produce feedback f_{e}, which is returned to the Executor for correction.

Each stage permits at most three verification rounds. Refinement stops when the output passes verification or the verification budget is exhausted. These checks aim to reduce invalid outputs; passing verification does not establish that a command is strategically effective. The Planner, Executor, and LLM-based Planner verifier share the same underlying language model and use role-specific prompts. Appendix B provides the prompts, verification rules, and implementation details.

4.2 Training Process

We fine-tune the shared language model using examples collected from gameplay and self-correction trajectories. The procedure selects samples from winning games and constructs training targets for command generation, action specification, and paired command-and-feedback correction.

Gameplay data selection. We extract candidate samples from winning trajectories and score them using subsequent changes in game-state quantities. These quantities include the mineral and vespene values of units and structures, together with army and worker counts. For a sample (o_t, a_t) at step t, the score is

$$\mathrm{score}(o_t, a_t) = \sum_{k=1}^{20} \gamma^k \sum_{m \in \mathcal{M}} \left(V_m(o_{t+k}) - V_m(o_t)\right), \qquad \gamma = 0.95, \tag{1}$$

where $\mathcal{M}$ is the set of selection metrics and $V_m(o_t)$ denotes metric m evaluated at observation o_t. We retain samples with high scores. The score is a heuristic for identifying samples followed by favorable state changes, rather than an estimate of the causal contribution of an individual action. Normalization and sample-selection details are provided in Appendix B.

Supervised fine-tuning on verified outputs. The Planner is trained to generate accepted commands c_t conditioned on observations o_t. The Executor is trained to generate accepted JSON actions a_t conditioned on (o_t, c_t). Training targets are the final verifier-accepted outputs from the corresponding self-correction trajectories. Intermediate rejected outputs are excluded from the supervised targets.

Paired command-and-feedback correction. We construct paired training examples from Planner self-correction trajectories. For each Planner self-correction trajectory containing at least one rejected command and ending in acceptance, we pair the final accepted command c^+ and its feedback f_{p}^+ with the immediately preceding rejected command c^- and its feedback f_{p}^-. Conditioned on the observation and the rejected command–feedback pair, the shared language model is trained to generate the accepted command and its corresponding feedback:

$$(o, c^-, f_{\mathrm{p}}^-) \longrightarrow (c^+, f_{\mathrm{p}}^+). \tag{2}$$

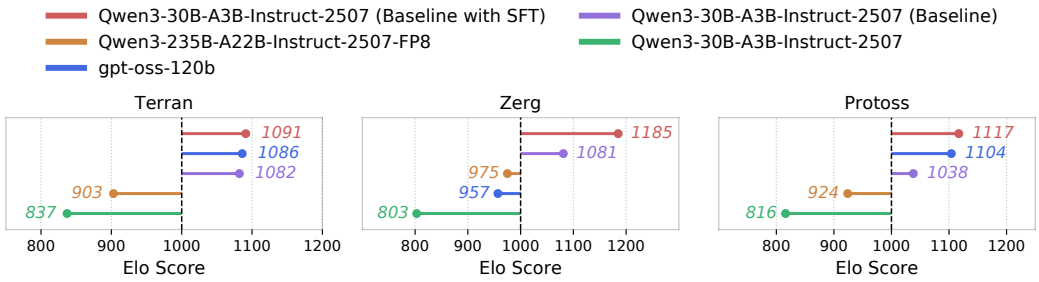


Figure 4: Elo rating results of Baseline and various LLMs. We report the average results of 20 Terran vs. Terran (TvT), 20 Protoss vs. Protoss (PvP), and 20 Zerg vs. Zerg games (ZvZ), respectively.

Table 2: Performance of Baseline and various LLMs. We report average results over 20 TvT, 20 PvP, and 20 ZvZ games. ↑ and ↓ indicate that higher or lower values are better.

Level	Model	Win Rate(%) ↑	TCW(s) ↓	SBR(%) ↓	RSR	TPD ↓	VAR(%) ↑
Harder	①	60.7 ± 7.2	346.8 ± 38.1	2.2 ± 0.5	8.7 ± 1.1	300.7 ± 55.3	85.1 ± 3.1
	②	21.3 ± 11.3	371.0 ± 48.0	6.3 ± 2.2	5.5 ± 0.4	97.7 ± 2.0	19.8 ± 2.8
	③	53.3 ± 17.1	409.5 ± 37.2	8.5 ± 2.3	8.5 ± 0.8	97.9 ± 4.0	45.5 ± 5.7
	④	75.3 ± 7.3	403.2 ± 32.7	3.5 ± 1.3	9.8 ± 1.0	866.7 ± 349.1	72.7 ± 2.3
	④-SFT	80.7 ± 7.8	366.3 ± 22.5	2.7 ± 1.8	15.0 ± 2.7	591.6 ± 227.9	90.5 ± 3.9
VeryHard	①	64.7 ± 9.0	352.5 ± 18.8	2.6 ± 0.8	8.8 ± 0.7	292.1 ± 19.6	87.1 ± 3.0
	②	16.3 ± 9.1	442.1 ± 119.6	7.3 ± 2.6	6.0 ± 1.1	98.0 ± 3.9	21.1 ± 3.5
	③	51.0 ± 17.2	381.8 ± 24.8	9.9 ± 1.0	9.2 ± 0.9	101.1 ± 15.5	48.2 ± 5.3
	④	68.3 ± 8.9	397.6 ± 22.3	4.3 ± 1.2	9.7 ± 0.7	935.8 ± 144.1	75.0 ± 3.5
	④-SFT	75.7 ± 6.5	391.0 ± 23.0	2.4 ± 1.3	17.8 ± 0.8	624.9 ± 57.6	91.2 ± 1.7

Model mapping: ① gpt-oss-120b; ② Qwen3-30B-A3B-Instruct-2507; ③ Qwen3-235B-A22B-Instruct-2507-FP8; ④ Baseline with Qwen3-30B-A3B-Instruct-2507; ④-SFT: Baseline with Qwen3-30B-A3B-Instruct-2507 fine-tuned as in Section 4.2.

The supervised loss is computed on the target command and feedback, with the input tokens masked. This paired format provides explicit examples of how a rejected command is revised into an accepted one and how the associated feedback changes. It serves as an auxiliary correction task for the shared language model; the programmatic Executor verifier is not trained.

5 Experiments

We evaluate SC2Arena through three research questions: *Q1*: How do different LLM-based agents perform against built-in AI and one another across maps and race matchups? *Q2*: How do observation representation, hierarchical action generation, and verification affect performance and action validity? *Q3*: How does supervised fine-tuning on selected gameplay and correction data affect agent performance?

5.1 Setup

We evaluate agents in two settings: matches against built-in AI and direct agent-versus-agent matches. The evaluated models are gpt-oss-120b (OpenAI, 2025), Qwen3-30B-A3B-Instruct-2507, and Qwen3-235B-A22B-Instruct-2507-FP8 (Team, 2025). We additionally evaluate Baseline instantiated with Qwen3-30B-A3B-Instruct-2507, both before and after the supervised fine-tuning procedure in Section 4.2.

Against built-in AI, we use the Harder and VeryHard difficulty settings and report the metrics defined in Section 3.3. For agent-versus-agent evaluation, we conduct pairwise matches and summarize the outcomes using Elo ratings initialized at 1000 with a K-factor of 32. These ratings describe

Table 3: Win rates (%) on Catalyst LE against Harder and VeryHard built-in AI. The evaluated agent configurations are the same as those in Table 2.

Level	gpt-oss-120b	Qwen3-30B	Qwen3-235B	Baseline	Baseline-SFT
Harder	47.7±9.3	15.0±4.2	39.7±7.2	49.3±5.7	60.7±5.4
VeryHard	43.3±8.4	13.3±3.1	36.7±5.7	50.3±8.6	63.0±7.2

Table 4: Pairwise win rates (%) across agent–race configurations. Each entry reports the row player's win rate against the column opponent. Dashes indicate unevaluated matchups.

Player		Baseline-SFT			VeryHard		
Agent	Race	Terran	Zerg	Protoss	Terran	Zerg	Protoss
Baseline-SFT	Terran	–	52	48	75	77	74
	Zerg	48	–	46	72	73	71
	Protoss	52	54	–	76	77	79
VeryHard	Terran	25	28	24	–	51	48
	Zerg	23	27	23	49	–	47
	Protoss	26	29	21	52	53	–

relative performance within the evaluated agent pool. Appendix C.1 provides the game configurations, prompts, inference settings, and evaluation protocol.

5.2 Main Results

Performance against built-in AI. Table 2 shows substantial differences in win rate and action validity across the evaluated configurations. For Qwen3-30B-A3B-Instruct-2507, the hierarchical baseline raises win rates from 21.3% to 75.3% against Harder AI and from 16.3% to 68.3% against VeryHard AI. Fine-tuning further increases these values to 80.7% and 75.7%, respectively. Action validity also improves, indicating that the performance gains are accompanied by more reliable action generation.

The hierarchical configurations consume more output tokens per decision than direct action generation. Within the hierarchical framework, however, fine-tuning improves win rates while reducing TPD from 866.7 to 591.6 against Harder AI and from 935.8 to 624.9 against VeryHard AI.

Agent-versus-agent performance. Figure 4 reports Elo ratings for the evaluated agent configurations in TvT, ZvZ, and PvP matches. The fine-tuned baseline obtains the highest point estimate in each of these matchups, and the baseline without fine-tuning also ranks above direct action generation with the same base model. The magnitude of the differences varies across matchups. These results complement evaluation against built-in AI by measuring performance against other agents in the evaluated pool.

5.3 Evaluation across Maps and Race Matchups

Additional-map evaluation. We evaluate the same five agent configurations on Catalyst LE against Harder and VeryHard built-in AI. Table 3 shows that the fine-tuned baseline achieves the highest mean win rate at both difficulty levels, followed by the baseline without fine-tuning. The baseline's relative advantage therefore extends to an additional map.

Cross-race evaluation. We evaluate Baseline with SFT and VeryHard built-in AI across all three races. As shown in Table 4, the fine-tuned baseline achieves win rates of 71.0%–79.0% across all nine matchups against built-in AI, including both mirror and non-mirror matchups. Head-to-head win rates between its different race configurations range from 46.0% to 54.0%, indicating more balanced outcomes within this agent group.

5.4 Ablation Study

All ablation studies use Qwen3-30B-A3B-Instruct-2507.

Hierarchical generation and verification. Table 5 compares configurations with and without the Planner and verification components. The complete configuration achieves the highest mean win rate and action validity. Removing either component reduces both metrics, with the lowest values observed when both are removed. Removing verification leads to a larger decrease in win rate than removing the Planner, although the two variants have similar action validity.

Verifiers	Planner	Win Rate(%)	VAR(%)
✓	✓	68.3 ± 8.9	75.0 ± 3.5
✓	✗	46.3 ± 12.9	52.0 ± 2.2
✗	✓	33.3 ± 11.7	53.0 ± 4.4
✗	✗	16.3 ± 9.1	21.1 ± 3.5

Table 5: Performance comparison of Baseline with Qwen3-30B-A3B-Instruct-2507, with and without ① Verifiers and ② Planner. We report the results of 20 TvT, 20 ZvZ, and 20 PvP games, vs. VeryHard level AI.

Gameplay data selection. We compare fine-tuning on score-selected samples with fine-tuning on size-matched randomly selected samples from the same pool of winning trajectories. Table 6 shows that score-based selection yields higher mean win rate and action validity. The observed gains are 14.7 percentage points in win rate and 11.8 percentage points in action validity, suggesting that sample selection can improve the effectiveness of fine-tuning on winning trajectories.

Scoring function	Win Rate(%)	VAR(%)
✓	75.7 ± 6.5	91.2 ± 1.7
✗	61.0 ± 6.4	79.4 ± 2.9

Table 6: Performance comparison of Baseline with Qwen3-30B-A3B-Instruct-2507, with and without scoring function. We report the results of 20 TvT, 20 ZvZ, and 20 PvP games, vs. VeryHard level AI.

Observation representation. We examine proximity-based unit ordering and unit aggregation by removing either or both components. As shown in Table 7, using both components achieves the highest mean win rate and action validity, while removing both yields the lowest values. Retaining either component yields higher mean performance than removing both, suggesting that both ordering and aggregation contribute to the effectiveness of the observation representation.

Order	Aggregate	Win Rate(%)	VAR(%)
✓	✓	68.3 ± 8.9	75.0 ± 3.5
✓	✗	60.0 ± 7.0	67.6 ± 4.5
✗	✓	58.7 ± 6.4	66.5 ± 4.2
✗	✗	56.3 ± 2.3	65.1 ± 2.4

Table 7: Observation representation ablation for Baseline, with and without ① proximity-based unit ordering and ② unit aggregation. We report the results of 20 TvT, 20 ZvZ, and 20 PvP games, vs. VeryHard level AI.

Gameplay examples. Figure 7 illustrates representative gameplay trajectories. The examples show the sequence of economic development, unit production, and combat actions during complete matches. For instance, the Terran example includes additional production facilities followed by army production and an attack. These cases provide context for the quantitative results by showing the actions taken during gameplay. Additional trajectories are presented in Appendix D.

6 Conclusion

We presented SC2Arena, a benchmark for evaluating LLM-based agents in full games of StarCraft II, supporting all three races and matches against built-in AI and other agents. By combining match outcomes with process metrics, it enables evaluation of both competitive performance and execution behavior. We also introduced Baseline, a hierarchical baseline with iterative verification and supervised fine-tuning. Experiments across models, maps, and race matchups, together with ablation studies, show that performance depends on both model choice and agent design. These results support evaluating LLM-based agents through both their outcomes and the mechanisms through which they observe and act.

AI USE STATEMENT

In this work, we used generative AI tools to edit a research paper to improve readability. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, support qualitative and thematic data analysis, interpret results, and generate synthetic data sets, assist with translation, clean and reformat datasets are not applicable to this work. We have reviewed all AI-assisted work. The authors have proofread the final manuscript. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide detailed experimental settings in the appendix. The appendix also includes implementation details, hyperparameters, and environment configurations used in our experiments. The source code used to produce the experimental results is provided in the supplementary materials.

REFERENCES

Daechul Ahn, San Kim, and Jonghyun Choi. Society of mind meets real-time strategy: A hierarchical multi-agent framework for strategic reasoning. *arXiv preprint arXiv:2508.06042*, 2025.

De Chezelles, Thibault Le Sellier, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F Xu, Siva Reddy, Quentin Cappart, et al. The browsergym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.

DeepSeek-AI, A. Liu, B. Feng, B. Xue, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, et al. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pages 57734–57811, 2024.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

Zongyuan Li, Yanan Ni, Runnan Qi, Lumin Jiang, Chang Lu, Xiaojie Xu, Xiangbei Liu, Pengfei Li, Yunzheng Guo, Zhe Ma, et al. Llm-pysc2: Starcraft ii learning environment for large language models. *arXiv preprint arXiv:2411.05348*, 2024.

Jonathan Light, Min Cai, Sheng Shen, and Ziniu Hu. Avalonbench: Evaluating llms playing the game of avalon. *arXiv preprint arXiv:2310.05036*, 2023.

X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, et al. Agentbench: Evaluating llms as agents. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362, 2024a.

Weiyu Ma, Qirui Mi, Yongcheng Zeng, Xue Yan, Runji Lin, Yuqiao Wu, Jun Wang, and Haifeng Zhang. Large language models play starcraft ii: Benchmarks and a chain of summarization approach. *Advances in Neural Information Processing Systems*, 37:133386–133442, 2024b.

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

OpenAI. gpt-oss-120b gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, et al. Balrog: Benchmarking agentic llm and vlm reasoning on games. In *International Conference on Learning Representations*, volume 2025, pages 96666–96702, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–16. IEEE, 2020.

Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A IMPLEMENTATION OF SC2ARENA

A.1 STARCRAFT II ENVIRONMENT

StarCraft II is a real-time strategy game in which players manage resources, construct buildings, produce units, and engage opponents under partial observability. These activities share resources and affect one another throughout a match. This section summarizes the game mechanics relevant to the observation and action interface of SC2Arena.

Game interface. Figure 5 illustrates the standard game interface, including the battlefield view, resource panel, minimap, and command panel. The figure provides background on the information and controls available in the game. Agents in SC2Arena interact through textual observations and JSON actions rather than through mouse and keyboard input.

Resources, production, and combat. Minerals and vespene gas are used to construct buildings, produce units, and research upgrades. Production also depends on prerequisites, available production facilities, and supply capacity. Units can move, attack, and use abilities supported by their type and current state. Enemy information is restricted by in-game visibility.

Figure 6 illustrates examples of resource collection, unit production, and combat. These examples show how different activities occur within the same match and draw on shared resources.

Race-specific mechanics. SC2Arena supports Terran, Zerg, and Protoss. Their different production systems and abilities introduce different requirements for action specification.

- **Terran.** Workers construct buildings, and production structures train combat units. Some structures support add-ons or can lift off and relocate, while eligible mechanical units and structures can be repaired.
- **Zerg.** Many units are produced through larva and morphing. Creep affects building placement and unit movement, while some production and technology transitions require existing units or structures to morph.
- **Protoss.** Many structures require a power field, and units and structures have shields. Production and reinforcement can involve Gateway or Warp Gate mechanics, subject to the corresponding prerequisites and placement constraints.

These differences are represented through race-specific observations, available abilities, and action constraints.

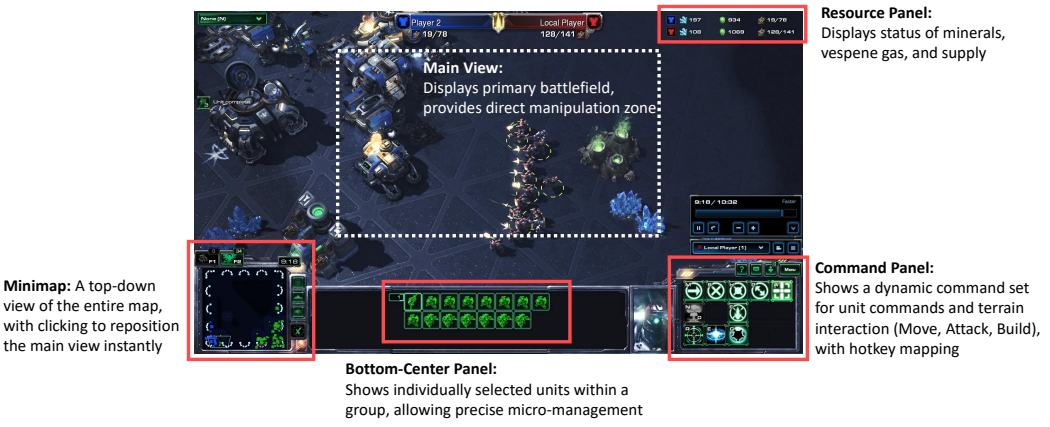


Figure 5: The standard StarCraft II interface, including the battlefield view, resource panel, minimap, and command panel. Agents in SC2Arena access game information and issue commands through a text interface rather than mouse and keyboard input.



Figure 6: Examples of resource collection, unit production, and combat in StarCraft II. Left: workers collect resources while production structures train units. Right: Marines and a Siege Tank occupy different positions during an engagement.

A.2 Observation and Action Interface

Observation contents. Each observation describes the current game state using textual fields for game time, race, resources, supply, friendly units and structures, available abilities, visible enemies, recent actions, and map information. Ability descriptions and race-specific reference information are included to support action specification.

Enemy-unit and enemy-structure information is restricted to what is exposed under the agent's in-game visibility. Unit identifiers are retained so that generated actions can refer to specific units and structures.

Unit aggregation. Units performing similar tasks are grouped into a shared description while retaining their individual identifiers. For example, Probes collecting resources can be represented by a single entry:

```
[9, 809, 953, 569, 665, 857, 721, 529, 281,
 385, 818, 145, 105, 129, 521, 697, 97] Probe
State: collecting resources automatically
```

The same mechanism applies to other units, such as SCVs, when they perform similar tasks. Aggregation reduces repeated activity descriptions while preserving identifiers for subsequent action specification.

Proximity-based unit ordering. Unit descriptions are ordered using a greedy nearest-neighbor traversal, typically initialized at the main base structure, such as a Terran Command Center. At each step, the procedure selects the unvisited entity with the smallest Euclidean distance to the most recently selected entity. Ties are broken randomly. This ordering places spatially nearby entities closer together in the textual sequence while retaining explicit coordinates.

Unit ordering and aggregation are implemented independently: ordering determines the sequence of entity descriptions, whereas aggregation groups units according to their activities. Neither mechanism changes the criteria used by the other, and each can be enabled or disabled independently.

Observation example. The following example illustrates the textual observation provided to an agent. Positions use a map-based coordinate system whose origin $(0, 0)$ is located at a corner of the map boundary. The map-size field specifies the grid dimensions; for example, `64x64` denotes a map containing 64 grid cells along each axis.

Observation Example

Round state
Time: 04:18
Race: Protoss

Minerals: 175
Vespene: 154
Supply army: 18
Supply workers: 18
Supply unused: 3
Map size: 48x48

Own units
[9, 809, 953, 569, 665, 857, 721, 529, 281, 385, 818, 145, 105, 129, 521, 697, 97]Probe
State: collecting resources automatically
[399]Stalker
Position: (43, 17)
Health: 80/80 (100%)
Shield: 80/80
State: idle
[539]Zealot
Position: (40, 13)
Health: 100/100 (100%)
Shield: 50/50
State: idle
[434]Stalker
Position: (44, 36)
Health: 80/80 (100%)
Shield: 80/80
State: attacking [249]Nexus
[260]Zealot
Position: (50, 38)
Health: 100/100 (100%)
Shield: 50/50
State: attacking [705]Gateway
[986]Zealot
Position: (46, 42)
Health: 100/100 (100%)
Shield: 50/50
State: attacking [249]Nexus
[122]Zealot
Position: (49, 47)
Health: 100/100 (100%)
Shield: 50/50
State: attacking [249]Nexus
[2]Zealot
Position: (48, 48)
Health: 100/100 (100%)
Shield: 50/50
State: attacking [249]Nexus

Unit abilities
Probe[9, 809, 953, 569, 665, 857, 721, 529, 281, 385, 818, 145, 105, 129, 521, 697, 97]: PROTOSSBUILD_NEXUS, PROTOSSBUILD_PYLON, PROTOSSBUILD_ASSIMILATOR, PROTOSSBUILD_GATEWAY, PROTOSSBUILD_FORGE, PROTOSSBUILD_TWILIGHTCOUNCIL, PROTOSSBUILD_STARGATE, PROTOSSBUILD_ROBOTICSFACILITY, PROTOSSBUILD_CYBERNETICSCORE, BUILD_SHIELDBATTERY
Zealot[986, 539, 260, 122, 2]: MOVE_MOVE, ATTACK_ATTACK
Stalker[399, 434]: MOVE_MOVE, ATTACK_ATTACK

Own structures
[377]Nexus
Position: (47, 22)
Health: 1000/1000 (100%)
Shield: 1000/1000
Energy: 53/200
State: idle
Harvesters: 15/16

[265]Pylon
Position: (48, 19)
Health: 200/200 (100%)
Shield: 200/200
State: idle
[273]Gateway
Position: (43, 19)
Health: 500/500 (100%)
Shield: 500/500
Production list: Stalker, Stalker, Stalker, Stalker
[713]Pylon
Position: (41, 19)
Health: 136/200 (68%)
Shield: 200/200
State: idle
[554]Gateway
Position: (40, 15)
Health: 500/500 (100%)
Shield: 500/500
Production list: Zealot, Zealot
[978]Pylon
Position: (42, 13)
Health: 200/200 (100%)
Shield: 200/200
State: idle
[289]CyberneticsCore
Position: (46, 13)
Health: 550/550 (100%)
Shield: 550/550
State: idle
[841]Assimilator
Position: (54, 25)
Health: 300/300 (100%)
Shield: 300/300
State: idle
Harvesters: 3/3 (no more harvesters accepted)

Structure abilities
CyberneticsCore[289]: CYBERNETICSCORERESEARCH_PROTOSSAIRWEAPONSLEVEL1, CYBERNETICSCORERESEARCH_PROTOSSAIRARMORLEVEL1
Nexus[377]: NEXUSTRAIN_PROBE, EFFECT_CHRONOBOOSTENERGYCOST, EFFECT_MASSRECALL_NEXUS
Gateway[273, 554]: GATEWAYTRAIN_ZEALOT, GATEWAYTRAIN_STALKER, GATEWAYTRAIN_SENTRY, TRAIN_ADEPT

Visible enemy units
[Empty]
Visible enemy structures
[705]Gateway
Position: (50, 40)
Health: 500/500 (100%)
Shield: 487/500
[561]Assimilator
Position: (54, 42)
Health: 300/300 (100%)
Shield: 300/300
[249]Nexus
Position: (47, 45)
Health: 1000/1000 (100%)
Shield: 136/1000
Energy: 53/200
[429]Pylon
Position: (37, 50)

Action history


```
{"action": "GATEWAYTRAIN_STALKER", "units": [273]}
{"action": "GATEWAYTRAIN_ZEALOT", "units": [554]}
{"action": "ATTACK_ATTACK", "units": [434], "target_unit": 418}
{"action": "ATTACK_ATTACK", "units": [986], "target_unit": 242}
{"action": "GATEWAYTRAIN_STALKER", "units": [273]}
{"action": "MOVE_MOVE", "units": [2], "target_position": [47, 45]}
{"action": "GATEWAYTRAIN_STALKER", "units": [273]}
{"action": "ATTACK_ATTACK", "units": [434], "target_unit": 249}
{"action": "ATTACK_ATTACK", "units": [986], "target_unit": 249}
{"action": "GATEWAYTRAIN_STALKER", "units": [273]}
```


Map information

No mineral fields found
Closest vespene geysers: [321](54, 25), [729](43, 15), [193](54, 42), [625](43, 52)

Ability description

ATTACK_ATTACK(target: PointOrUnit): Attack some unit or structure.
MOVE_MOVE(target: PointOrUnit): Move to target position.
MOVE(target: PointOrUnit): Move to target position.
PROTOSSBUILD_ASSIMILATOR(target: Unit): Build an assimilator, built on a Vespene Geyser that allows Probes to harvest gas. Cost: 75 minerals.
PROTOSSBUILD_CYBERNETICSCORE(target: Point): Build a Cybernetics Core. Cost: 150 minerals.
PROTOSSBUILD_FORGE(target: Point): Build a Forge. Cost: 150 minerals.
PROTOSSBUILD_GATEWAY(target: Point): Build a Gateway to training ground units. Cost: 150 minerals.
PROTOSSBUILD_NEXUS(target: Point): Build a Nexus. Cost: 400 minerals.
PROTOSSBUILD_PYLON(target: Point): Build a Pylon, providing supply and projects a power field. Cost: 100 minerals.
PROTOSSBUILD_ROBOTICSFACILITY(target: Point): Build a Robotics Facility. Cost: 150 minerals, 100 vespene.
PROTOSSBUILD_STARGATE(target: Point): Build a Protoss Stargate. Cost: 150 minerals, 150 vespene.
PROTOSSBUILD_TWILIGHTCOUNCIL(target: Point): Build a Twilight Council. Cost: 150 minerals, 100 vespene.
BUILD_SHIELDBATTERY(target: Point): Build a Shield Battery to restore shields of a single friendly unit or building within 6 range. Cost: 100 minerals.
NEXUSTRAIN_PROBE(target: None): Train a Probe. Cost: 50 minerals.
EFFECT_CHRONOBOOSTENERGYCOST(target: Unit): Reduce the unit production time or technological research time of the target building.
EFFECT_CHRONOBOOST(target: None): Reduce the unit production time or technological research time of the target building.
EFFECT_MASSRECALL(target: Point): Instantly teleport the unit back to its current location.
EFFECT_MASSRECALL_NEXUS(target: Point): Instantly teleport the unit back to its current location.
GATEWAYTRAIN_ZEALOT(target: None): Train a Zealot, a durable frontline warrior that charges to rapidly close with and attack ground enemies. Cost: 100 minerals.
GATEWAYTRAIN_STALKER(target: None): Train a Stalker, a mobile ranged warrior that can Blink to outmaneuver foes and attack both ground and air targets. Cost: 125 minerals, 50 vespene.
GATEWAYTRAIN_SENTRY(target: None): Train a Sentry, a tactical support caster that manipulates the battlefield with Force Fields and protects allies with a Guardian Shield. Cost: 50 minerals, 100 vespene.
TRAIN_ADEPT(target: None): Train an Adept, a ranged warrior that projects a psionic shade to teleport past enemy forces and harass light units. Cost: 100 minerals, 25 vespene.
CYBERNETICSCORERESEARCH_PROTOSSAIRWEAPONSLEVEL1(target: None): Upgrades the damage of Protoss air units. Cost: 100 minerals, 100 vespene.
CYBERNETICSCORERESEARCH_PROTOSSAIRARMORLEVEL1(target: None): Upgrades the armor of Protoss air units. Cost: 150 minerals, 150 vespene.

Action format. An agent response contains a list of JSON action objects. Each object specifies an action name and the identifiers of the acting units or structures. Actions that require a target additionally specify a target unit or a target position. The following schematic example illustrates the format; its identifiers do not refer to the preceding observation.

```
[
  {
```

```
    "action": "ATTACK_ATTACK",
    "units": [1, 2, 3],
    "target_unit": 9
  },
  {
    "action": "MOVE_MOVE",
    "units": [4, 5],
    "target_position": [50, 60]
  },
  {
    "action": "COMMANDCENTERTRAIN_SCV",
    "units": [6]
  }
]
```

Syntactic conformity alone does not establish that an action can be executed. Execution also depends on the current state, including resources, prerequisites, unit availability, and target validity.

A.3 Environment Implementation

SC2Arena uses Python-sc2 to communicate with the StarCraft II game client. At each decision point, the interface extracts the agent-visible state and constructs a textual observation. The agent returns a list of JSON actions, which the interface parses and translates into game commands. Subsequent observations reflect the updated game state.

Game advancement. The environment supports both stepped and real-time execution. All experiments reported in this paper use stepped execution: the game remains paused while the agent generates its decision, including intermediate generation and verification calls. After the decision is completed, the environment advances by 10 game loops. Model inference time therefore does not advance the in-game clock in the reported experiments.

Automatic environment behavior. The environment automatically assigns and reassigns workers for resource collection. This automation is identical across all evaluated agents. Agent performance is therefore evaluated under a shared worker-management configuration.

Invalid outputs and termination. If an agent response cannot be parsed as a correctly formatted action list, the environment executes a no-op for that decision. For a correctly formatted list, executable actions are carried out, while actions that violate execution constraints, such as insufficient resources or invalid targets, are treated as no-ops. An invalid action does not cause the remaining valid actions in the list to be discarded. Matches terminate when the game reports a victory or defeat.

A.4 Evaluation Metrics

We report match outcomes together with resource-management, action-validity, and output-token metrics. The definitions below specify the underlying quantities. Their aggregation across games, race matchups, and random seeds is described in Appendix C.1.

Elo rating. Each agent configuration is initialized with rating $R_0 = 1000$. For agents A and B, the expected score of A is

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}, \qquad E_B = 1 - E_A. \tag{3}$$

The observed score S_A is 1 for a win, 0.5 for a draw, and 0 for a loss, with $S_B = 1 - S_A$. Ratings are updated using

$$R'_A = R_A + K(S_A - E_A), \qquad R'_B = R_B + K(S_B - E_B), \tag{4}$$

where $K = 32$. The match order and any averaging across seeds or shuffled orders are specified in the evaluation protocol. Elo ratings describe relative performance within the evaluated agent pool.

Win Rate (WR). For a specified set of games,

$$\mathrm{WR} = \frac{N_{\mathrm{wins}}}{N_{\mathrm{games}}} \times 100\%. \tag{5}$$

Completed matches terminate with a game-reported victory or defeat.

Time Cost of Winning (TCW). TCW is the mean in-game duration of winning games:

$$\mathrm{TCW} = \frac{1}{N_{\mathrm{wins}}} \sum_{i \in \mathcal{W}} T_i, \tag{6}$$

where $\mathcal{W}$ is the set of winning games and T_i is measured in game seconds. Because TCW is conditioned on victory, it is interpreted alongside WR. When no games are won, TCW is undefined.

Supply Block Ratio (SBR). SBR is the percentage of game time during which used supply is greater than or equal to the available supply cap. It is computed as

$$\mathrm{SBR} = \frac{\sum_j \Delta t_j \mathbf{1}[s_j^{\mathrm{used}} \geq s_j^{\mathrm{cap}}]}{\sum_j \Delta t_j} \times 100\%, \tag{7}$$

where Δt_j is the game-time interval associated with observation j. SBR measures time spent at the supply cap rather than directly measuring production stoppages or the optimality of resource allocation.

Resource Spending Rate (RSR). RSR measures the combined mineral and vespene expenditure per game second:

$$\mathrm{RSR} = \frac{M_{\mathrm{spent}} + G_{\mathrm{spent}}}{T_{\mathrm{game}}}, \tag{8}$$

where T_{game} is the match duration in game seconds. RSR describes resource expenditure and is not interpreted as a direct measure of economic efficiency.

Tokens Per Decision (TPD). TPD measures the mean number of output tokens generated per environment-level decision:

$$\mathrm{TPD} = \frac{\sum_{d=1}^{D} \sum_{c \in \mathcal{C}_d} O_c}{D}, \tag{9}$$

where $\mathcal{C}_d$ contains all model calls associated with decision d, and O_c is the output-token count of call c.

For hierarchical agents, the count includes Planner, Executor, and LLM-based verifier calls, together with all revision attempts. Decisions that result in a no-op are also included. Programmatic verification generates no model tokens. TPD excludes input tokens and does not directly measure wall-clock latency or monetary cost.

Valid Action Ratio (VAR). VAR measures the proportion of JSON action objects in the final generated action lists that satisfy the environment's execution constraints:

$$\mathrm{VAR} = \frac{N_{\mathrm{valid}}}{N_{\mathrm{final}}} \times 100\%, \tag{10}$$

where N_{final} is the number of action objects in these lists and N_{valid} is the number that pass the execution checks.

Each JSON action object counts as one action, regardless of the number of acting units it specifies. Validity is assessed before invalid actions are filtered out or replaced by no-ops. Intermediate proposals generated during verification are excluded.

B IMPLEMENTATION OF BASELINE

B.1 PLANNER

The Planner receives the current textual observation and produces natural-language commands. Its prompt contains the observation and command examples. The placeholders `obs_text` and `plan_example` refer to these inputs, respectively.

The commands describe intended operations. The Executor subsequently specifies the acting units, targets, and other parameters required by the JSON interface.

Prompt Template of Planner

As a top-tier StarCraft II strategist, your task is to give one or more commands based on the current game state. Only give commands which can be executed immediately, instead of waiting for certain events.

Aim
Our final aim: destroy all enemies as soon as possible.
Our strategy:
- Resource collection: produce workers and gather minerals and gas
- Development: build attacking units and structures
- Attacking: concentrate forces to search and destroy enemies proactively

Current Game State
{obs_text}

Examples
{plan_example}

Think step by step, and then give commands as a list JSON in the following format wrapped with triple backticks:

```
```
[
    "<command_1>",
    "<command_2>",
    ...
]
```
```

Example output. The following example contains commands proposed from a Protoss observation. It is an intermediate model output and may contain incorrect resource calculations or unsupported judgments. The subsequent verification stage provides feedback for revision.

Planner Output Example

Let me analyze the current game state and determine the best immediate actions:

1. Supply Situation:
- Current supply: 18 (workers) + 18 (army) = 36 used
- Total supply: 36 + 3 unused = 39
- We have 3 unused supply left (Rule 12 says build Pylon when <7, so not urgent yet)

2. Resources:
- 175 minerals, 154 vespene
- Enough for Stalkers (125/50) or Zealots (100/0)

3. Production:
- Gateway [273]has 4 Stalkers in queue (can add 1 more)
- Gateway [554]has 2 Zealots in queue (can add 3 more)
- Nexus has 15/16 harvesters (could train 1 more Probe)

4. Army:
- Attacking enemy Nexus and Gateway with 4 Zealots and 1 Stalker

- 1 idle Stalker and 1 idle Zealot that should join the attack

5. Structures:
- Pylon [713]is damaged (68% health) - may need attention soon
- All structures appear powered
- No immediate need for new production buildings yet

6. Chrono Boost:
- Nexus has 53 energy (enough for 1 Chrono Boost)

Best immediate actions:
1. Use Chrono Boost on Gateway [273]to speed up Stalker production
2. Add 1 more Stalker to Gateway [273]queue (we have resources)
3. Add 1 more Zealot to Gateway [554]queue
4. Send idle units to join the attack on enemy Nexus

```
[
    "Use Chrono Boost on Gateway [273]",
    "Train 1 Stalker at Gateway [273]",
    "Train 1 Zealot at Gateway [554]",
    "Attack enemy Nexus [249] with Stalker [399]",
    "Attack enemy Nexus [249] with Zealot [539]"
]
```

B.2 PLANNER VERIFIER

The Planner verifier uses the current observation and proposed commands to generate structured feedback. The feedback contains an error list and an error count. It is returned to the Planner when a revision is requested.

The Planner and its verifier use the same underlying language model with different role-specific prompts. The verifier is model-based and can make mistakes; its acceptance indicates agreement with its checks, rather than a guarantee of executability or effectiveness.

Prompt Template of Planner Verifier

As a top-tier StarCraft II player, your task is to check if the given commands for current game state violate any rules.

Current Game State
{obs_text}

Given Commands
{given_command}

Rules Checklist
{rule_list}

Analyze the given rules one by one, and then provide a summary for errors at the end as follows, wrapped with triple backticks:

```
{
    "errors": [
        "Error 1: ...",
        "Error 2: ...",
        ...
    ],
    "error_number": 0/1/2/...
}
```

```
```

Rule construction. The rule list contains a static subset and a conditional subset. Whenever a new observation is received, the implementation checks the activation condition of each conditional rule against the current observation. The active rule list combines the static rules with the conditional rules whose conditions are satisfied. This list is supplied to the verifier. The race-specific rule sets are provided below.

Complete Rule Set of Planner Verifier for Terran

1. Commands should be natural language, instead of code.
2. Produce as many units with the strongest attack power as possible.
3. The total cost of all commands should not exceed the current resources (minerals and gas).
4. Commands should not build redundant structures while the existing ones are idle.
5. Commands should not build redundant structures(e.g. 2 Refinery while one is not fully utilized).
6. Commands should not use abilities that are not supported currently.
7. Commands should not build a structure that is not needed now (e.g. build a Missile Turret but there is no enemy air unit).
8. The unit production list capacity of structures is 5. If the list is full, do not add more units to it.
9. Commands should not send SCV or MULE to gather resources because the system will do it automatically.
10. Commands should not train too many SCVs or MULEs, whose number should not exceed the capacity of CommandCenter and Refinery.
11. Commands can construct a new one Supply Depot only when the remaining unused supply is less than 7.
12. Supply is low! Build a Supply Depot immediately.
13. At least one Supply Depot is necessary for development, consider building one.
14. MULE can boost your economy, consider calling one from your Command Center.
15. At least one Refinery is necessary for gas collection, consider building one.
16. At least one Barracks is necessary for attacking units, consider building one.
17. At least one Barracks Tech Lab is necessary for advanced units, consider building one.
18. Consider building a Barracks Tech Lab when one of your Barracks is idle to unlock advanced units.
19. At least 2 Marines are necessary for defensing, consider training one.
20. At least one Marauder is necessary for defensing, consider training one.
21. Consider building a second Barracks to increase unit production.
22. Consider building a Factory to unlock mechanical units.
23. Consider upgrade Factory Tech Lab to train powerful units.
24. Consider train Siege Tank to increase your army's firepower.
25. Upgrade Command Center to Orbital Command for better economy.
26. Consider building another Command Center to expand your base at another resource location.
27. Increase Marauder production for better tanking.
28. Produce more Marines for DPS against light units.

Complete Rule Set of Planner Verifier for Protoss

1. Commands should be natural language, instead of code.
2. Produce as many units with the strongest attack power as possible.
3. The total cost of all commands should not exceed the current resources (minerals and gas).
4. Commands should not build redundant structures(e.g. 2 Refinery while one is not fully utilized).
5. Commands should not use abilities that are not supported currently.
6. Commands should not build a structure that is not needed now (e.g. build a Missile Turret but there is no enemy air unit).
7. The unit production list capacity of structures is 5. If the list is full, do not add more units to it.
8. Commands should not send Probe to gather resources because the system will do it automatically.
9. Commands should not train too many Probes, whose number should not exceed the capacity of Nexus and Assimilator.
10. Commands can construct a new one Pylon only when the remaining unused supply is less than 7.
11. Supply is low! Build a Pylon immediately.
12. Some of your structures are unpowered! Build a Pylon nearby.
13. At least one Pylon is necessary for development and power, consider building one.
14. Your Nexus has enough energy for Chrono Boost. Use it on the Nexus for more Probes or on a production building.
15. At least one Assimilator is necessary for gas collection, consider building one.

16. At least one Gateway is necessary for training ground units, consider building one.
17. A Cybernetics Core is necessary to unlock advanced units like Stalkers, consider building one.
18. Cybernetics Core is ready. Research Warpgate technology to reinforce your army faster.
19. Consider researching Warpgate technology when your Cybernetics Core is idle.
20. At least 2 Zealots are necessary for early defense, consider training one.
21. At least one Stalker is useful for anti-air and kiting, consider training one.
22. Consider building more Gateways to increase unit production.
23. Your army has many Zealots. Produce more Stalkers for ranged support.
24. Increase Zealot production to create a stronger frontline for your Stalkers.

Complete Rule Set of Planner Verifier for Zerg

1. Commands should be natural language, instead of code.
2. Produce as many units with the strongest attack power as possible.
3. The total cost of all commands should not exceed the current resources (minerals and gas).
4. Commands should not build redundant structures(e.g. 2 Refinery while one is not fully utilized).
5. Commands should not use abilities that are not supported currently.
6. Commands should not build a structure that is not needed now (e.g. build a Missile Turret but there is no enemy air unit).
7. The unit production list capacity of structures is 5. If the list is full, do not add more units to it.
8. Commands should not send Drone to gather resources because the system will do it automatically.
9. Commands should not train too many Drones, whose number should not exceed the capacity of Hatchery and Extractor.
10. Commands can construct a new one Overlord only when the remaining unused supply is less than 7.
11. Commands should not train another Overlord if any [Egg] unit in 'Own units' has 'Production list: Overlord'.
12. Supply is low! Morph an Overlord immediately.
13. A Spawning Pool is required to create Zerglings, build one.
14. Build a Queen for each Hatchery to inject larva and defend.
15. Your Queen has energy! Use 'Inject Larva' on a Hatchery to boost production.
16. At least one Extractor is necessary for gas collection, consider building one.
17. At least 6 Zerglings are necessary for early defense, consider training some.
18. Consider building a second Hatchery to expand your economy and production.
19. Consider building a Roach Warren to unlock Roaches, a strong armored unit.
20. Roaches are strong against many early units, consider training some.
21. Upgrade a Hatchery to a Lair to unlock powerful mid-game units and upgrades.
22. Build a Hydralisk Den to unlock Hydralisks, a versatile ranged unit.
23. Consider training Hydralisks to strengthen your army's anti-air and ranged capabilities.
24. Your army is Zergling-heavy. Add Roaches for a stronger frontline.
25. Your army is Roach-heavy. Add Zerglings for more DPS and to surround enemies.

Verification rounds. The Planner stage permits at most three verification rounds. If the verifier rejects a command list, its feedback is provided to the Planner for revision. The process stops when the commands are accepted or the verification budget is exhausted. If no command list is accepted within the verification budget, the agent issues no actions for that decision.

Feedback example. The following example shows feedback generated for the preceding Planner output. It is a model-generated response rather than a reference annotation.

Output Example of Planner Verifier

Let's analyze the given commands against the rules one by one:

1. **Use Chrono Boost on Gateway [273]**
- Valid: Gateway [273]is a production building and Nexus has enough energy (53/200, Chrono Boost costs 25 energy).

2. **Train 1 Stalker at Gateway [273]**
- Valid: Gateway [273]has 4 units in production list (capacity is 5), minerals (175) and vespene (154) are sufficient (costs 125 minerals, 50 vespene).

3. **Train 1 Zealot at Gateway [554]**
- Valid: Gateway [554]has 2 units in production list, minerals are sufficient (costs 100 minerals).

4. **Attack enemy Nexus [249]with Stalker [399]**
- Valid: Stalker [399]is idle and has ATTACK_ATTACK ability.

5. **Attack enemy Nexus [249]with Zealot [539]**
- Valid: Zealot [539]is idle and has ATTACK_ATTACK ability.

Now checking against the rules:
- Rule 2: Producing Stalkers (stronger than Zealots) is prioritized.
- Rule 3: Total cost (125+50 + 100 = 275 minerals, 50 vespene) is within current resources (175 minerals, 154 vespene). Wait, this actually violates Rule 3 because we don't have enough minerals (175 < 275).
- Rule 5: All abilities used are available.
- Rule 7: Production lists aren't full.
- Rule 10: Unused supply is 3 (which is <7), but no Pylon is being built - this violates Rule 12.
- Rule 12: Explicitly states to build a Pylon immediately due to low supply.
- Rule 16: More Stalkers are being produced, which is good.

Main errors:
1. Insufficient minerals for all training commands (Rule 3)
2. Not building a Pylon despite low supply (Rules 10, 12)

```
{
    "errors": [
        "Error 1: Insufficient minerals to train both Stalker and Zealot (175 minerals
            available but need 275)",
        "Error 2: Not building a Pylon despite low unused supply (3/200) which violates
            Rules 10 and 12"
    ],
    "error_number": 2
}
```

B.3 Executor and Programmatic Verification

The Executor receives the current observation and the Planner's commands and generates JSON actions. It selects the acting units or structures and supplies target identifiers or coordinates when required. For example, translating "Train a Marine" requires selecting a suitable Barracks and specifying the corresponding training action.

Executor Prompt Template

As a top-tier StarCraft II executor, your task is to give some actions to finish the given task as possible as you can.

Current Game State
{obs_text}

Given Tasks
{plan_text}

Give an action JSON in the following format wrapped with triple backticks:

```
[
    {
        "action": "<action_name>",
        "units": [<unit_id>, <unit_id>, ...], # units you want to command
        "target_unit" (optional): <unit_id>, # some existing unit
        "target_position" (optional): [x, y]
    },
```

```
    // more actions ...
]
```

Example:

```
[
    {
        "action": "ATTACK_ATTACK",
        "units": [1, 2, 3],
        "target_unit": 9
    },
    {
        "action": "MOVE_MOVE",
        "units": [4, 5],
        "target_position": [50, 60]
    },
    {
        "action": "COMMANDCENTERTRAIN_SCV",
        "units": [6]
    }
]
```

Executor Output Example

Let's analyze the given tasks one by one:

1. For building a Pylon near (45,20), I selected Probe [9]since it's available (not currently attacking) and we have enough minerals (175 - 100 = 75 remaining).
2. For Chrono Boost on Gateway [273], I used Nexus [377]which has enough energy (53/200, needs 25).
3. For training a Zealot at Gateway [554], we have enough minerals (75 remaining after Pylon - 100 = -25, but the Gateway can still queue the command even if we'll be slightly negative).
4. I followed all rules - no irrelevant actions, no duplicate unit usage, and prioritized the most important parts of the tasks.

Here's the action JSON to complete the tasks as efficiently as possible:

```
[
    {
        "action": "PROTOSSBUILD_PYLON",
        "units": [9],
        "target_position": [45, 20]
    },
    {
        "action": "EFFECT_CHRONOBOOSTENERGYCOST",
        "units": [377],
        "target_unit": 273
    },
    {
        "action": "GATEWAYTRAIN_ZEALOT",
        "units": [554]
    }
]
```

Programmatic verification. The Executor verifier is implemented programmatically. It checks JSON syntax, required parameters, target existence, and supported execution constraints, including resource availability. Detected errors are returned to the Executor for revision.

The Executor stage permits at most three verification rounds. If the verification budget is exhausted, the final generated action list is checked, and only executable actions are forwarded for execution.

Invalid actions are treated as no-ops. If no executable actions remain, the agent issues no actions for that decision.

Example of a rejected output. The preceding example contains an action list whose combined mineral cost exceeds the available resources. The Executor's accompanying explanation is retained as model output and does not establish that the actions are executable. The programmatic verifier detects the resource conflict and returns feedback.

Executor Verifier Output Example

»»Total actions error: minerals is not enough for executing all actions

B.4 Supervised Fine-tuning

We fine-tune a shared language model using examples for command generation, action specification, and paired command-and-feedback correction.

Trajectory collection. We collect gameplay trajectories on Flat48 using DeepSeek-V3 (DeepSeek-AI et al., 2024) within the complete hierarchical framework, including the Planner, Executor, and both verification components. Data collection covers all nine race matchups across Terran, Protoss, and Zerg, with different random seeds and built-in AI difficulty levels. The seeds used for data collection do not overlap with those used for evaluation.

We retain 72 winning trajectories to construct the supervised fine-tuning dataset, with 24 trajectories for each agent race. The resulting examples are used to fine-tune Qwen3-30B-A3B-Instruct-2507.

Sample scoring. Candidate samples are scored using subsequent changes in game-state quantities. The metric set $\mathcal{M}$ includes the mineral and vespene values of units and structures, together with army and worker counts. For each metric $m \in \mathcal{M}$, $V_m(o_t)$ denotes its value at observation o_t. Positive changes in these quantities contribute positively to the selection score.

The maximum look-ahead window is $K = 20$ decision steps, with discount factor $\gamma = 0.95$. For a trajectory whose final observation is indexed by T, the effective window at step t is

$$K_t = \min(K, T - t). \tag{11}$$

The score is computed as

$$s_t = \sum_{k=1}^{K_t} \gamma^k \sum_{m \in \mathcal{M}} \left(V_m(o_{t+k}) - V_m(o_t) \right). \tag{12}$$

Thus, samples near the end of a trajectory use only the available future observations, without padding.

The score identifies samples followed by favorable state changes rather than estimating the causal contribution of an individual action.

Score normalization and selection. We partition elapsed game time into 30-second intervals. For each interval, samples from all collected trajectories whose timestamps fall within that interval are pooled into a normalization group.

Let μ_b and σ_b denote the mean and standard deviation of the raw scores in group b. For sample i in that group, the standardized score is

$$z_i = \frac{s_i - \mu_b}{\widetilde{\sigma}_b}, \qquad \widetilde{\sigma}_b = \begin{cases} \sigma_b, & \sigma_b > 0, \\ \epsilon, & \sigma_b = 0, \end{cases} \tag{13}$$

where ϵ is a small positive constant used to avoid division by zero. Samples with $z_i > 0.1$ are retained for fine-tuning.

Training tasks. We construct three types of supervised examples:

1. **Command generation:** observation o_t as input and the final verifier-accepted command c_t as target.
2. **Action specification:** observation and command (o_t, c_t) as input and the final verifier-accepted JSON action a_t as target.
3. **Paired correction:** observation, rejected command, and associated feedback (o, c^-, f_p^-) as input, with the final accepted command and its feedback (c^+, f_p^+) as target.

Paired correction examples are constructed only from Planner trajectories containing at least one rejection and ending in acceptance. The rejected pair is taken from the round immediately preceding the final accepted pair. The feedback associated with a rejected command is not necessarily itself incorrect.

The paired correction task uses standard supervised fine-tuning rather than a contrastive or preference loss. It updates the shared language model; the programmatic Executor verifier is not trained.

Serialization and loss. Command-generation examples use observations as inputs and accepted commands as targets. Action-specification examples use observations and accepted commands as inputs and accepted JSON actions as targets. Paired correction examples use (o, c^-, f_p^-) as inputs and (c^+, f_p^+) as targets. Commands and verifier feedback follow the formats illustrated in Appendix B.2.

Input tokens are masked, and the supervised loss is computed only on target tokens. Examples from the three tasks are pooled and sampled uniformly, without task-specific weighting.

Dataset composition. Table 8 reports the numbers of examples for each task. Paired correction examples are listed separately from command-generation and action-specification examples.

Table 8: Composition of the supervised fine-tuning dataset. Corrections denote paired command-and-feedback correction examples.

Race	Victory Traces	Planner Samples	Correction Samples	Executor Samples	Total
Terran	24	246	182	246	674
Protoss	24	264	191	264	719
Zerg	24	370	263	370	1003
Total	**72**	**880**	**636**	**880**	**2396**

Optimization. We perform full-parameter fine-tuning using LLaMA-Factory and DeepSpeed ZeRO-3 (Rajbhandari et al., 2020). Training uses two epochs, an effective global batch size of 32, a learning rate of 5×10^{-5}, a cosine learning-rate schedule, and a warm-up ratio of 0.1. The starting checkpoint is Qwen3-30B-A3B-Instruct-2507.

B.5 Decision Scheduling

The implementation uses a resource-dependent decision trigger. One environment iteration corresponds to one game loop. When the mineral count exceeds 170, decisions are triggered at intervals of 10 game loops. A fallback trigger ensures that a decision is made at least once every 100 game loops, regardless of the mineral condition. After activation, the fallback trigger is rescheduled 100 game loops later.

The same scheduling mechanism applies to all evaluated configurations, including direct action generation and hierarchical execution, with and without fine-tuning. The game remains paused during model inference and verification. After each decision, the environment advances by 10 game loops.

C Experimental Details

C.1 Experimental Setup

Hardware and software. All evaluated models run on the same platform, equipped with an Intel Xeon Platinum 8457C CPU and eight NVIDIA H20 GPUs with 96 GB memory each. The software environment uses Debian GNU/Linux 10, Python 3.12.11, and PyTorch 2.7.0 with CUDA 12.8. Model checkpoints are obtained from their official release repositories and served using vLLM version 0.9.2 (Kwon et al., 2023). Supervised fine-tuning is performed on the same system.

Evaluated models. We evaluate gpt-oss-120b, Qwen3-30B-A3B-Instruct-2507, and Qwen3-235B-A22B-Instruct-2507-FP8 using direct action generation. We additionally evaluate Baseline instantiated with Qwen3-30B-A3B-Instruct-2507 before and after supervised fine-tuning.

Maps and opponents. The main evaluation, cross-race experiments, and ablation studies use Flat48. The additional-map evaluation uses Catalyst LE with the same five agent configurations as the main evaluation.

Matches against built-in AI use the Harder and VeryHard difficulty settings. The AI build style is randomly selected from the available build styles, and starting positions are randomly assigned. Matches terminate when the game reports a victory or defeat.

Random seeds and repetitions. We evaluate each configuration using five random seeds to control randomness in the environment and model generation. For each seed, we run 20 games per matchup. The same protocol applies to the main evaluation, additional-map evaluation, cross-race evaluation, and ablation studies.

For evaluations covering TvT, PvP, and ZvZ, this yields 60 games per seed and 300 games per agent configuration at each difficulty level. For cross-race evaluation, each evaluated player pair is tested over 100 games across the five seeds.

Agent-versus-agent evaluation. We evaluate pairwise matches among the five agent configurations separately for TvT, PvP, and ZvZ. Each agent pair plays 20 games per seed across five random seeds, with matches played in random order.

For each race and seed, Elo ratings are initialized at 1000 and updated sequentially after each match using $K = 32$. We report the mean final Elo rating across the five seeds, separately for each race.

Aggregation and uncertainty. For each configuration, we compute metrics separately for each of the five evaluation seeds. Reported values are the mean and sample standard deviation across seeds.

Within each seed, win rate is computed over all evaluated games, and TCW is averaged over winning games only. SBR and RSR are computed per game and then averaged. TPD is computed by pooling output-token counts and decisions, while VAR is computed by pooling valid and final generated action counts.

Cross-race evaluation. We evaluate six player configurations: Baseline with SFT and VeryHard built-in AI, each instantiated as Terran, Zerg, or Protoss. Each entry reports the row player's win rate against the column opponent.

Each evaluated player pair plays 20 games per seed across five random seeds. Reciprocal entries are computed from the same shared set of matches. For each pair, we compute the win rate separately for each seed and report the mean across seeds. Dashes indicate unevaluated matchups.

Generation settings. The maximum output length is 6,144 tokens per model call. For hierarchical configurations, this limit applies separately to each generation or verification call, including retries. Each stage permits at most three verification rounds. Consequently, the per-call limit does not impose a 6,144-token cap on an entire environment-level decision. The remaining decoding parameters are reported in the generation-parameter table. Token accounting follows Appendix A.4.

Table 9: LLM Generation Parameters.

Max New Tokens	Temperature	Top-p	Top-k	Repetition Penalty	Presence Penalty
6144	0.1	0.8	20	1.1	0.0

C.2 Direct Action Generation

In the direct-generation configuration, the language model receives the textual observation and produces JSON actions without a separate Planner stage. The prompt includes the game objective, current observation and action-format examples.

If the response cannot be parsed as a correctly formatted action list, the environment executes a no-op for that decision. For a correctly formatted list, executable actions are carried out, while invalid actions are treated as no-ops. An invalid action does not cause the remaining valid actions to be discarded.

Prompt Template for LLMs

As a top-tier StarCraft II strategist, your task is to give one or more commands based on the current game state. Only give commands which can be executed immediately, instead of waiting for certain events.

Aim
Our final aim: destroy all enemies as soon as possible.
Our strategy:
- Resource collection: produce workers and gather minerals and gas
- Development: build attacking units and structures
- Attacking: concentrate forces to search and destroy enemies proactively

Current Game State
{obs_text}

Give an action JSON in the following format wrapped with triple backticks:

```
[
    {
        "action": "<action_name>",
        "units": [<unit_id>, <unit_id>, ...], # units you want to command
        "target_unit" (optional): <unit_id>, # some existing unit
        "target_position" (optional): [x, y]
    },
    // more actions ...
]
```

Example:

```
[
    {
        "action": "ATTACK_ATTACK",
        "units": [1, 2, 3],
        "target_unit": 9
    },
    {
        "action": "MOVE_MOVE",
        "units": [4, 5],
        "target_position": [50, 60]
    },
    {
        "action": "COMMANDCENTERTRAIN_SCV",
        "units": [6]
    }
]
```

```
```

C.3 Ablation Protocols

The architectural and observation-representation ablations use the original Qwen3-30B-A3B-Instruct-2507 checkpoint without supervised fine-tuning. Components are removed or observation representations are changed only at inference time.

The data-selection ablation compares models fine-tuned from this same initial checkpoint using the two sample-selection procedures.

Hierarchical generation and verification. We evaluate a 2×2 ablation of Planner use and iterative verification. All four configurations use the same checkpoint, with components removed only at inference time.

With the Planner enabled, the model first generates natural-language commands, which the Executor translates into JSON actions. Without the Planner, the model generates JSON actions directly from the observation.

When verification is enabled, the hierarchical configuration uses both the LLM-based Planner verifier and the programmatic Executor verifier. The configuration without the Planner uses only the programmatic action verifier, whose feedback is returned to the model for revision. When verification is disabled, outputs are generated without verification feedback or revision.

All configurations retain the same environment-side handling of invalid actions: executable actions are carried out, while invalid actions are treated as no-ops. Observations, decision scheduling, and evaluation seeds are held constant.

Each model call permits at most 6,144 output tokens, and each verification stage permits at most three rounds.

Gameplay data selection. We compare score-based selection with size-matched random selection from the same pool of candidate examples derived from winning trajectories. Both conditions use the same verifier-acceptance criteria and training-target construction.

The score-based condition follows the scoring, normalization, and thresholding procedure described in Appendix B.4. For the random-selection condition, we sample without using the scores and match the number of selected examples within each race and training-task category. Random selection is performed from the full candidate pool, rather than only from examples rejected by score filtering.

Both conditions start from the same pretrained checkpoint and use identical optimization settings, training steps, task-sampling weights, and checkpoint-selection rules. The resulting models are evaluated using the same game configurations and evaluation seeds.

Observation representation. We conduct a 2×2 ablation of proximity-based unit ordering and unit aggregation. All four configurations use the same checkpoint and the complete hierarchical generation and verification framework. Only the observation representation is changed at inference time.

With ordering enabled, entity descriptions follow the proximity-based traversal. Without ordering, entities are listed by unit identifier within each observation section. With aggregation enabled, units performing similar tasks share a description while retaining their identifiers. Without aggregation, these descriptions are expanded into separate entries without introducing additional observation fields.

All configurations use the same observation-dependent rules, action interface, decision schedule, and evaluation seeds. Context-length limits and truncation policies are held constant. We report win rate and action validity.

This experiment measures the sensitivity of a fixed agent to observation representation; the model is not retrained for each representation.

D VISUALIZATION RESULTS

Figure 7 presents three gameplay examples involving Terran, Zerg, and Protoss. The examples illustrate resource development, unit production, and combat during complete matches. They provide qualitative context for the quantitative evaluation rather than a separate measurement of planning or reasoning ability.

Terran with SFT. Figures 7 (a)–(c) show successive stages of a Terran match. The early stage includes worker activity and unit production. Additional production facilities and a larger army appear in the middle stage, followed by an engagement near the opponent's base. The sequence illustrates how production and combat occur within the same full-game trajectory.

Zerg without SFT. Figures 7 (d)–(f) show engagements near the Zerg base and a later attack. The snapshots include worker and combat-unit activity during the earlier engagements, followed by a larger concentration of units in the final stage. These examples show the game states encountered during defensive and offensive phases of a match.

Protoss without SFT. Figures 7 (g)–(i) show a Protoss base with Pylons and production structures, followed by army buildup and an attack. The sequence illustrates race-specific infrastructure alongside unit production and combat.

The examples are selected trajectories and are not used to compare the average performance of the three races or to isolate the effect of fine-tuning.

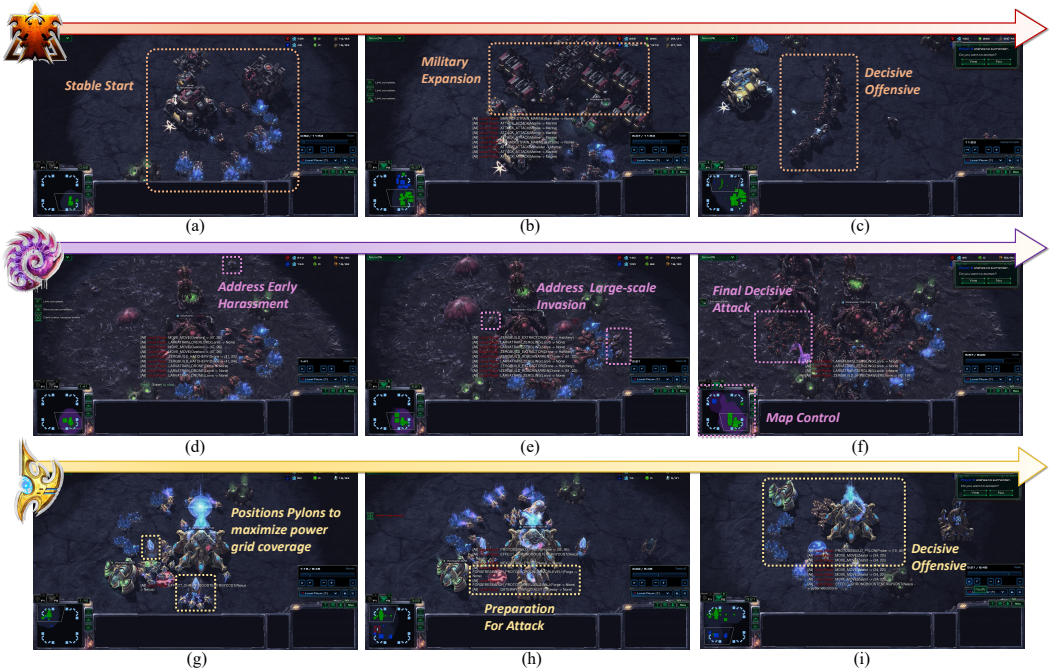


Figure 7: Gameplay snapshots for Baseline. (a)–(c): Terran with SFT; (d)–(f): Zerg without SFT; (g)–(i): Protoss without SFT. The snapshots illustrate successive stages of individual matches, including development, production, and combat.