Hierarchical Control for FPS NPCs: LLM Intent Selection over RL Sub-Policies

Valentina Cadena Fajardo, Saly Al Masri, Bashar Levin, Nasir Alizade
DD2430 HT25 – Project Course in Data Science,
KTH Royal Institute of Technology
Email: {vcadena, salyam, basharl, alizade}@kth.se

***Abstract*—**

I. Introduction

Modern first-person shooters (FPS) demand non-player characters (NPCs) that are both fast and reactive and strategically coherent. Reinforcement learning (RL) excels at the former: once trained, policies run at control rate and can exploit subtle movement and aiming patterns, but they struggle with long-horizon decisions (e.g., disengaging, repositioning, conserving ammunition) and can be brittle under scenario shifts. Large language models (LLMs), in contrast, are well suited for planning, abstraction, and tool selection, but are too slow and noisy to emit frame-by-frame actions in a real-time FPS. These complementary strengths motivate a hierarchical architecture in which an LLM sets high-level intent and selects among specialized, fast RL sub-policies that execute low-level control.

In this project we build and deploy such a hierarchical controller in Unity as a proof of concept. We train two RL policies in Unity ML-Agents: a navigation policy that moves a shared character controller in a randomized 36×36 arena to approach enemy targets, and a combat policy that controls continuous yaw/pitch and firing to track and shoot a moving enemy. At runtime, both policies are wrapped by a *HierarchicalControllerRuntime* that activates either the navigation or combat agent and enforces safe switching based on line-of-sight, distance, and simple timers. An external LLM-based planner, implemented as a local HTTP service backed by the LLM agents runtime, is integrated into the controller: a Unity-side `PlannerClient` periodically sends a compact, structured state summary and receives high-level "navigate" versus "combat" intents, which are smoothed by an `LLMSwitcher` and applied as mode overrides. A *StatsLogger* component collects telemetry (health, distance, visibility, selected mode, and LLM rationale) for offline analysis, and a simple key toggle allows enabling and disabling LLM control during runs.

Contributions: This work makes the following contributions:

- **LLM–RL integration API.** We implement a lightweight API that connects Unity RL policies to an external planner. A planner client serializes a structured observation (`PlannerObs`: distance, visibility, self/target health, ammunition, enemy count, and timing features) into JSON and sends it via HTTP to an LLM-backed service, which returns a response (`PlannerResp`) containing a discrete mode, a free-text reason, and a time-to-live parameter `expires_in`. On the Unity side, request cadence and robustness are controlled using configurable `queryInterval`, `requestTimeout`, and an `enableLLM` flag.
- In our implementation this service is instantiated as a locally hosted LLM agents backend running on the same machine as the Unity environment, which demonstrates that the integration is model-agnostic and can be driven by open-weight LLMs without relying on external cloud APIs.
- **Dual RL sub-policies for navigation and combat.** We train two continuous-control policies in Unity ML-Agents. The navigation policy controls continuous yaw and forward motion in a randomized arena, with reward shaping for progress, facing alignment, and stuck detection. The combat policy controls continuous yaw and pitch together with a firing scalar mapped to weapon actions, with reward shaping for predictive aiming, accurate shots, enemy elimination, and health-aware penalties.
- **Hierarchical controller with LLM-guided switching.** We design a hierarchical controller that selects between the navigation and combat sub-policies while enforcing temporal hysteresis and safety constraints. Internally, it uses line-of-sight tests, distance thresholds, and visibility/hidden timers to gate mode changes; externally, an *LLMSwitcher* aggregates LLM outputs over time, requires multiple identical decisions in a row, and enforces a minimum dwell time between switches (via parameters such as `switchCooldown` and decision-hold timers) to mitigate noisy or delayed LLM responses.
- **Evaluation protocol and initial results.** We define an evaluation protocol to compare RL-only, rule-based, and LLM+RL control using metrics such as time-to-kill (TTK) and time-to-death (TTD) proxies, distance-to-enemy statistics, and mode-switching patterns. We report learning curves demonstrating convergence of the navigation and combat policies and describe initial online runs with and without LLM guidance using the shared StatsLogger pipeline; a more extensive statistical comparison is left for future work.

Scope and assumptions: We study a single-agent NPC in 1v1 arena combat under partial observability, with a fixed sensor suite (agent pose and velocity, enemy-relative pose and line-of-sight flags, implicit obstacle information via collisions,

ammunition, and health). The LLM-based planner receives a compact state summary at a low effective frequency on the order of 1–2 Hz, determined by its query interval and response time-to-live, and outputs high-level intents over a small discrete set (*navigate* vs. *combat*). RL policies execute continuous movement and aiming and discrete combat actions at control rate (on the order of 10–20 Hz) inside Unity. We focus on the control and switching problem—how to coordinate an LLM meta-policy with RL sub-policies under real-time constraints—rather than on dialogue, narrative generation, or multi-agent coordination.

Current status and planned extensions: At the current stage, both navigation and combat policies exhibit stable training curves in their respective environments and can be composed in the unified hierarchical controller. We already collect online logs of full episodes with and without LLM guidance (toggled at runtime) using the shared telemetry pipeline, which allows us to perform preliminary analyses of how LLM-driven switching affects engagement patterns and mode usage. For the camera-ready version, we plan to expand these experiments to larger batches of episodes, report more detailed TTK/TTD and damage statistics, and include ablations on alternative switching rules, prompt variants, and planner latency, together with qualitative visualizations of typical LLM-triggered repositioning and disengage behaviors.

II. Related Work

Section overview. Prior work spans (A) LLM-guided exploration/pretraining for RL, (B) hierarchical skill design with language (options, policy-over-skills, code-as-policy), and (C) LLMs in the decision loop (latency/safety) and compile-time LLM use to avoid run-time cost. We map where each line helps a real-time controller and where it falls short for an FPS agent that must switch between low-latency RL sub-policies online.

A. LLM-guided exploration and pretraining (offline or episodic language; fast control remains RL). ELLM uses a pretrained LLM during pretraining to propose human-meaningful goals from textual scene descriptions; the agent receives intrinsic rewards for achieving those goals, improving coverage and downstream performance in Crafter and Housekeep [1]. Crucially, no LLM is used at inference—the language model only structures exploration before task learning. This suggests a separation of timescales: language for guidance, RL for control.

Words as Beacons adopts a teacher–student scheme where an LLM emits episode-start subgoals (positional targets, object descriptors, or textual instructions). Across procedurally generated MiniGrid tasks, this curriculum accelerates learning by 30–200$\times$ while keeping the LLM out of the test-time loop [2]. Methodologically, it provides a template for coarse subgoal sequences that an FPS switcher could translate into "navigate" vs. "combat" intents.

STARLING shows another offline use: it auto-generates $\sim$100 interactive-fiction games with GPT-3 (Inform7) to pretrain text RL agents, then evaluates transfer. Results emphasize that current agents still trail humans on generalization, but the paper establishes a curriculum-generation pipeline that could analogously produce scripted FPS scenarios (e.g., "low-ammo retreat," "flank-and-burst") without invoking an LLM online [3].

Mini-gap. These approaches use language before control to improve exploration or supply training data; none arbitrate among multiple low-latency, learned skills during real-time play.

B. Hierarchies, skills, and language as the glue (LLM designs skills/switching; RL executes fast control). RL-GPT introduces a two-level framework: a slow LLM decomposes tasks and writes coded actions (code-as-policy) while a fast RL module learns the rest; the coded actions are inserted into the action space to boost sample efficiency. In MineDojo, the system reaches diamonds within roughly one day on a single RTX 3090 and attains SOTA on selected tasks—evidence that programmatic macro-actions + RL actuation is effective when control must be precise and rapid [4].

MaestroMotif operationalizes AI-assisted skill design: from natural-language skill specs, an LLM (i) derives skill-specific reward functions via preference feedback and (ii) generates code for initiation/termination and a policy-over-skills. Low-level policies are trained with RL; at deployment, the code policy composes them nearly zero-shot in NetHack, yielding a language-grounded options framework that automates much of the reward/option engineering [5].

LGSD targets semantic diversity in skills. It uses LLM-generated descriptions to define a language-distance over states (with a 1-Lipschitz constraint) so discovered skills are semantically distinct; prompts restrict the semantic subspace, and a learned inference module supports zero-shot skill selection from text. This yields interpretable skill sets (e.g., "hold-cover," "flank-left," "retreat-reload") suitable for a higher-level switcher [6].

Mini-gap. These works build interpretable skills and switchers, but none evaluate an online LLM arbiter that selects among independently trained, low-latency FPS sub-policies under strict timing.

C. LLMs in (or out of) the real-time loop: latency, safety, and "compile-time" planning. HighwayLLM combines a pretrained RL planner with an LLM that either predicts waypoint trajectories or acts as a safety layer for driving. Safety improves over RL alone, but the latency gap is stark: $\sim$6.79 s (waypoints) and $\sim$2.89 s (safety) versus $\sim$0.002 s for the RL action—orders of magnitude slower than control frequency [7]. This strongly motivates designs where LLMs do not operate at frame rate.

YOLO-MARL minimizes latency and cost by querying the LLM once per environment to generate a strategy, a state-interpretation function, and a planning function (Python). After that, MARL training proceeds and decentralized policies run without the LLM; on LBF and MPE it outperforms several baselines. Attempts at LLM-generated reward functions underperform without iterative refinement. This "compile-time LLM" pattern preserves a fast run-time loop [8].

ACE (*Think Twice, Act Once*) takes a different path: keep the LLM offline during RL training to refine trajectories (LLM as Policy Actor) and shape rewards (LLM as Value Critic), yielding higher survival/reward on large power-grid benchmarks while avoiding inference-time latency [9]. The lesson is consistent: leverage LLM reasoning between control steps; let RL handle the fast loop.

Positioning. Across A–C, language helps before or between control steps—via exploration guidance, curriculum/data generation, skill/reward/option design, or one-shot compilation of planners—while RL handles millisecond actuation. When LLMs are placed in the real-time loop, measured latencies are incompatible with FPS control budgets. We therefore target a real-time FPS setting in which an LLM operates at a low cadence to select/switch between specialized, trained RL sub-policies (navigation/combat), preserving responsiveness while adding strategic adaptation. To our knowledge, prior art has not reported LLM-driven arbitration among multiple low-latency FPS policies at run time under tight latency constraints.

III. Problem Description

Problem context. We study real-time control for a first-person-shooter (FPS) non-player character (NPC) that must navigate and fight under strict latency constraints. A precise specification is required to enable reproducible evaluation across maps and to compare solutions that separate *fast motor control* from *slower strategic selection*.

A. Formal statement

We model the environment as an episodic partially observable Markov decision process (POMDP)

$$\mathcal{M} = \big(\mathcal{S}, \mathcal{A}, \Omega, O, P, R, \gamma\big),$$

with hidden state $s_t \in \mathcal{S}$, observation $o_t \in \Omega$, action $a_t \in \mathcal{A}$, transition kernel $P(s_{t+1} \mid s_t, a_t)$, observation kernel $O(o_t \mid s_t)$, reward $R(s_t, a_t)$, and discount $\gamma \in (0, 1)$.

A solution is a *hierarchical policy* $\pi = \{\mu, (\pi_\omega)_{\omega \in \mathcal{W}}\}$ with option set

$$\mathcal{W} := \{\texttt{NAV\_LOS}, \texttt{NAV\_TARGET}, \texttt{COMBAT}\}.$$

In our implementation, `NAV_LOS` and `NAV_TARGET` are navigation sub-policies that control the shared character controller to move either toward positions affording line of sight to the enemy or toward designated pickups (e.g., health), respectively. By contrast, `COMBAT` denotes a combat sub-policy that controls continuous yaw/pitch and firing to track and shoot a visible enemy. A meta-policy $\mu(\omega_t \mid \tilde{o}_t)$ selects an option $\omega_t \in \mathcal{W}$ at a low rate based on a compact summary $\tilde{o}_t$ of recent observations, subject to a minimum dwell time, while the option-conditioned low-level policy $\pi_{\omega_t}(a_t \mid o_t)$ acts at the control rate.

The objective is to maximize expected episodic return

$$J(\pi) \;=\; \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \, R(s_t, a_t)\right], \tag{1}$$

subject to real-time feasibility and safety constraints.

Real-time feasibility.: In Unity, the RL sub-policies run at a control frequency $f_{\text{ctrl}} \in [10, 20]\,\text{Hz}$ with actuation latency $\ell^{\text{act}} \le 1/f_{\text{ctrl}}$. Meta-decisions are triggered by a planner client with query interval $0.5\,\text{s}$, so the meta-decision frequency satisfies $0 < f_{\text{sel}} \le 2\,\text{Hz}$. The effective rate is lower in practice because requests are only sent when a coarse state signature changes and are further throttled by time-to-live values (`expires_in`) returned by the planner. Temporal hysteresis is enforced by a cooldown of $1.0\,\text{s}$ in the hierarchical controller and additional decision-hold logic in the LLM switcher (including a requirement for multiple consistent decisions), yielding a minimum dwell time of roughly 1–2 s between mode switches. Planner requests use a timeout of $1.0\,\text{s}$, so $\ell^{\text{meta}} \le 1.0\,\text{s}$; observed LLM response times are well within this budget and remain slow compared to the faster control loop.

Failover policy.: The external planner is advisory rather than mandatory. A boolean flag `enableLLM`, exposed via a runtime toggle, controls whether planner outputs are used. When `enableLLM` is false, or when HTTP requests issued by the planner client fail (e.g., due to timeouts or network errors), the system logs a warning and continues to operate under the last selected mode and the onboard RL sub-policies. Thus, control automatically defaults to the local hierarchical controller when the external planner is unavailable or disabled, and LLM guidance acts as an optional overlay instead of a hard dependency.

B. Scope and assumptions

We consider a single agent in **1v1** arena combat with **partial observability** (the agent receives local sensory features; no access to opponent internals). Dynamics are stationary during an episode. Physics, collision layers, and map geometry follow Unity defaults unless specified. Episodes terminate on task completion or timeout.

C. Instance space / environment

Episodes are drawn from a family of square arenas of side length 36 units (half-size 18), centered at an arena center c (in our experiments, c is either the origin or the parent transform of the agent). At $t = 0$, the player body is spawned at height $y = 0$ with position

$$x \sim \mathcal{U}[c_x - 18,\, c_x + 18], \qquad z \sim \mathcal{U}[c_z - 18,\, c_z + 18],$$

and a random yaw angle in $[0, 2\pi)$. The enemy (navigation goal) is placed independently at a random position in the same square at height $y \approx 0.25$. Static colliders tagged `Walls` or `Obstacle` define hard barriers; they shape navigation through collision penalties and stuck detection and also determine line-of-sight corridors via raycasts in the hierarchical controller. The nominal episode horizon is

$$T_{\max} = 30\,\text{s},$$

but episodes may terminate earlier if either the player or the current target reaches zero health, or if task-specific timeouts in the training environments (e.g. the navigation agent's `maxEpisodeTime`) are reached.

D. Inputs, outputs, and interfaces

At each control step the solver receives a compact observation vector and must emit a control action. Observation and action spaces are specific to the active sub-behavior (navigation or combat).

Navigation features: For navigation, the low-level policy observes a goal position expressed in the agent-local frame, normalized by the arena half-size (three scalars), local planar velocity (v_x, v_z) normalized by the maximum move speed, the agent's orientation encoded as $(\sin \text{yaw}, \cos \text{yaw})$, a grounded flag, and a normalized distance-to-goal term (distance divided by twice the arena half-size). Together, these features describe where the enemy/goal lies relative to the agent, how the agent is currently moving, and whether it is on the ground.

Combat features: For combat, the low-level policy observes the enemy direction in the agent-local frame, a normalized enemy distance (actual distance divided by a maximum spawn distance), a smoothed estimate of enemy velocity in the agent-local frame, the agent's own health as a fraction of its maximum health, and a binary "enemy alive" flag. This focuses the representation on relative geometry, target motion, and survivability.

Actions: The navigation policy outputs continuous rotation and movement commands (continuous yaw and forward motion, plus an optional jump scalar) that are mapped to turning, walking, and jumping via a shared character controller. The combat policy outputs three continuous scalars: a yaw command in $[-1, 1]$ (scaled to a yaw rate of approximately $200°$/s), a pitch command in $[-1, 1]$ (integrated and clamped between $-60°$ and $+60°$), and a firing scalar in $[-1, 1]$ that is interpreted as "fire" when above a threshold (mapped to weapon actions via the `PlayerWeaponsManager`). Combat actions do not move the character body; they only orient the agent and control firing, while navigation is solely responsible for locomotion.

E. Timing and real-time constraints

Low-level control in Unity ML-Agents executes at a decision frequency $f_{\text{ctrl}} \in [10, 20]$ Hz, with actuation latency bounded by $\ell^{\text{act}} \leq 1/f_{\text{ctrl}}$ and additional interpolation in the Unity update loop during inference. The meta-policy is queried by a planner client with a configurable query interval of 0.5 s, so the nominal meta-decision frequency satisfies $0 < f_{\text{sel}} \leq 2$ Hz; in practice the effective rate is lower because requests are sent only when a coarse state signature changes and are further throttled by time-to-live values returned by the planner. Meta-decisions use an HTTP timeout of 1.0 s, so $\ell^{\text{meta}} \leq 1.0$ s, and observed LLM round-trip times remain well within this budget. Combined with dwell-time enforcement and hysteresis in the hierarchical controller and LLM switcher, these choices ensure that the symbolic real-time constraints specified in Sec. III-A are satisfied by construction.

F. Constraints and resources

Solutions operate on a single workstation-class GPU/CPU; memory and wall-clock budgets for evaluation are fixed across methods. Safety guards include collision handling and switch gating (e.g., no switches during critical animations or very short time intervals). Additional implementation details and resource limits are kept consistent across all evaluated controllers.

G. Success criteria and evaluation targets

Primary metrics are **win rate** (%), **time-to-kill (TTK)** and **time-to-death (TTD)** in seconds. Secondary metrics include **damage dealt/received**, **navigation success rate** and **path efficiency** (actual/shortest path), **number of policy switches**, **switch quality** (e.g., time spent in combat while the enemy is visible versus not visible), and **switch latency** (meta-intent timestamp to low-level takeover). In our experiments Sec. V we run on the order of 10–20 episodes per condition (e.g., LLM-on vs. LLM-off) and aggregate metrics across runs. Concretely, we expect that enabling LLM guidance should not increase TTD (the agent should not die faster), should reduce average distance to the enemy when the enemy is not currently visible (more proactive searching), and should reduce the fraction of clearly inappropriate switches (e.g., remaining in combat with no line of sight). Due to time constraints, our empirical analysis in Sec. **??** focuses on TTK/TTD proxies, distance-to-enemy statistics, and mode-switching patterns; the remaining metrics are left as future work. All metrics are computed from CSV logs produced by the `StatsLogger` component, which records distance, visibility, health, ammunition, active mode, and LLM-provided reasons at a fixed sampling interval.

H. Non-goals

We do **not** address multi-agent coordination beyond 1v1, dialogue or narrative generation, or learning map design; these are out of scope.

Implementation mapping.: In the Unity implementation, the meta-policy $\mu(\cdot \mid \tilde{o}_t)$ is realized by an external LLM service that is queried over HTTP by a `PlannerClient` component. The compact summary $\tilde{o}_t$ is encoded as a `PlannerObs` structure containing, among others, the agent–target distance, a visibility flag, the target and self health values, ammunition in the current clip, enemy count, time since the target was last seen, a cover flag, an estimated weapon range, and time since damage was last taken. The option-conditioned low-level policies are instantiated as Unity ML-Agents navigation and combat policies and are orchestrated at runtime by the `HierarchicalControllerRuntime` and `LLMSwitcher` components.

Notation (selected)

$\mathcal{M}$: POMDP; $\mathcal{S}, \mathcal{A}, \Omega, O, P, R, \gamma$: standard components; μ: meta-policy; $\mathcal{W} = \{\texttt{NAV}, \texttt{COMBAT}\}$: option set; π_ω: low-level policy; f_{ctrl}: control frequency; f_{sel}: meta-decision frequency; $\tau_{\min}$: minimum dwell; $T_{\max}$: episode timeout; arena side length $= 36$ units.

IV. Methodology and Implementation

Our system follows the hierarchical structure defined in Section 3: a slow *meta-policy* (the LLM) selects among several fast, specialized RL sub-behaviors. In the implemented

system, these sub-behaviors consist of a navigation policy that controls continuous locomotion and a combat policy that controls aiming and firing; both are instantiated as Unity ML-Agents policies and are orchestrated at runtime by a `HierarchicalControllerRuntime` component.

In the initial conceptual design, the LLM was assumed to operate at 5 Hz and to output one of the discrete intents

$$\{\texttt{NAV\_LOS},\ \texttt{NAV\_TARGET},\ \texttt{COMBAT}\},$$

At training time, `NAV_LOS` and `NAV_TARGET` correspond to two distinct navigation training environments, but in the deployed system they give rise to a single navigation sub-policy, so the meta-policy effectively selects between a navigation mode and `COMBAT` at runtime and must obey a minimum dwell time to avoid rapid oscillation. In the final implementation, the planner is queried by a Unity-side client at a nominal interval of 0.5 s and only when a coarse state signature changes; together with planner-provided time-to-live values, this yields an effective meta-decision frequency on the order of 1–2 Hz, in line with the real-time constraints discussed in Section 3.

The selected sub-behavior then controls the agent at the environment's control frequency (10–20 Hz). All sub-behaviors are implemented in Unity ML-Agents using PPO and run on-device to guarantee deterministic, low-latency actuation.

An LLM policy switcher selects which sub-behavior to activate based on the current game state. We then compare this modular approach against a single end-to-end RL baseline trained on the same overall objective.

We describe each sub-behavior compactly and consistently below.

A. Navigation-to-LOS sub-behavior

This environment serves as one of two training scenarios for the navigation policy family; the resulting policy is later deployed as the navigation sub-policy within the arena task described in Section 3.

Objective: Reach any position where a straight ray to the enemy is unobstructed, i.e., achieve line-of-sight (LOS).

Observations (15-dim): A compact vector encoding geometry, collision sensing, and cover-point discovery:

- Agent (x, z) and enemy (x, z), normalized (4)
- Enemy direction in agent-local frame (2)
- LOS flag (1)
- Current "cover point": (x, z), presence flag, normalized distance (4)
- Three short-range wall feelers (forward, left, right) (3)
- Agent speed (1)

Actions (discrete multi-branch):

- Move: {none, forward, backward}
- Strafe: {none, left, right}
- Turn (yaw): {none, left, right}

Reward shaping: Encourages efficient discovery of cover and disciplined locomotion:

- Progress reward for reducing distance to the current cover point
- "Discovery" reward when identifying a better cover point (closer to the turret) via ray-cast analysis
- Penalties for collisions, revisiting recent positions, being stuck, or unnecessary backward motion
- Small per-step penalty for efficiency
- Large terminal reward when LOS becomes true

B. Navigation-to-Target Sub-Behavior

This sub-behavior provides a complementary navigation training scenario focused on reaching pickups such as health or ammunition, but in the final system its learned behaviour is subsumed into the shared navigation sub-policy used at runtime.

Objective: Move efficiently toward a designated pickup (e.g., health or ammo) while avoiding obstacles.

Observations:

- Relative target direction in local frame
- Agent velocity (normalized)
- Orientation encoding $(\sin \text{yaw}, \cos \text{yaw})$
- Grounded flag
- Normalized distance to target

Actions: Continuous:

- Yaw rotation
- Forward/backward movement

Reward shaping:

- Positive reward for reducing distance to target
- Terminal reward upon reaching the target
- Step penalty for idling or wandering
- Overshoot penalty if distance increases after close approach
- Penalties for repeated collisions or excessive turning
- Timeout penalty on failure to reach the target

C. Combat Sub-Behavior

Objective: Acquire LOS, track a moving enemy, and eliminate them rapidly using predictive aiming.

Observations (14-dim):

- Enemy direction (local frame)
- Normalized enemy distance
- Enemy velocity (local frame)
- Agent health
- Enemy-alive flag
- Last-seen enemy position

Actions (continuous):

- Yaw rotation
- Pitch rotation
- Fire trigger (activates when > 0.5)

Reward shaping:

- Aim-alignment reward: dot-product between weapon direction and enemy direction
- Fire-accuracy reward: positive when firing while well aligned; negative otherwise
- Kill reward with small bonus for faster kills
- Penalties for self-damage
- Terminal penalties: -1 on agent death, -0.5 on timeout

Training notes: Each sub-behavior was trained with PPO for $\geq$2M steps. Dense shaping was essential: without alignment and timing penalties the combat policy collapsed to circling or continuous firing. The final Combat policy reliably tracks moving enemies and performs short, controlled bursts when aligned.

Runtime role: At runtime, the hierarchical controller and meta-policy only need to arbitrate between a navigation mode and the combat mode described above; finer distinctions such as LOS-oriented versus pickup-oriented navigation are handled internally by the learned navigation sub-policy rather than exposed as separate options to the LLM.

D. LLM Meta-Policy and Switching

In the initial design, the long-term goal of the system was to use an LLM as a high-level meta-policy that would operate at a low frequency (nominally 5 Hz) and select among the trained RL sub-behaviors (`NAV_LOS`, `NAV_TARGET`, `COMBAT`).

The LLM would receive a compact state summary (visibility status, distances, health, ammo, last-seen direction, etc.) and output the next high-level intent. This design follows the hierarchical structure described in Section 3, where slow, strategic reasoning is separated from fast low-level control.

Final implementation: In the final implementation, this meta-policy is realized by an external LLM-based planner that is exposed as a local HTTP service on `localhost`. A Unity component `PlannerClient` sends JSON-encoded observations to a `/plan` endpoint, and the service forwards these requests to an instruction-tuned open-weight model managed by the LLM agents runtime. The planner is contacted at a nominal interval of 0.5 s and only when a coarse state signature changes, while planner-provided time-to-live values defer subsequent queries; the effective meta-decision frequency is therefore on the order of 1–2 Hz, substantially slower than the 10–20 Hz control loop executed by the navigation and combat sub-policies.

Current prototype (rule-based meta-controller).: The current prototype integrates a locally hosted Large Language Model served through an HTTP API to handle the high-level decision making for the agent. The system relied on a simlpe traditional hierarchical controller before the LLM was introduced. In the previous system mode switching was determined by some predefined game variables such as

- EnemyDetected
- EnemyVisiable
- Distance to enemy
- Player health
- Enemy health

In order for the machine to decide whether the agent should remain in **Navigation mode** or switch to **Combat mode**, these variables were fed into a rule-based state machine. With the LLM integrated, this decision logic is now replaced with a new high-level reasoning loop. The controller creates a prompt every 0,3 seconds that contains the current state of the environment, such as enemy detected, distance, line of sight, health, etc. and sends it to the OpenAI API. After receiving this piece of information, the LLM responds with either "**NAVIGATE** or **COMBAT**", then the controller activates or deactivates the sub-agent accordingly (NavigationAgentRuntime or CombatAgentRuntime). The rest, such as combat movement and aiming is still handled by the low-level rl-agents.

Planner observations and responses: In the deployed system, the information passed from Unity to the planner is encoded as a compact observation vector $\tilde{o}_t$ represented in code by a `PlannerObs` structure. This structure aggregates the agent–target distance, a binary visibility flag, the target and self health values, the number of bullets in the current clip, an estimate of the current enemy count, the time elapsed since the target was last seen, a Boolean indicator of whether the agent is presently in cover, an estimated effective weapon range, and the time since the agent last took damage. The planner response is a small JSON object mapped to a `PlannerResp` record containing (i) a categorical mode label (typically "navigate" or "combat"), (ii) a free-text rationale string that can be logged and inspected offline, and (iii) a scalar `expires_in` field that specifies a time-to-live for the current decision and is used to throttle subsequent planner queries.

Planner client loop and throttling: The `PlannerClient` component implements an asynchronous loop that executes every `queryInterval` seconds. At each iteration it constructs the current observation $\tilde{o}_t$, computes a coarse discrete signature summarizing key aspects of the state (distance bin, visibility, whether the agent was seen recently, a low-health flag, ammunition availability, and enemy count), and only issues an HTTP request if this signature differs from the last transmitted one and if the next allowed query time (determined by `expires_in`) has elapsed. Requests are sent as JSON-encoded `PlannerObs` instances to a local `/plan` endpoint on `localhost`, subject to a configurable timeout `requestTimeout`. On success, the client parses the JSON reply into a `PlannerResp` object, updates its internal deferral timer via `DeferRequestsFor(expires_in)`, and forwards the decision to downstream components; on failure or timeout it logs a warning and leaves the current mode unchanged. A Boolean flag `enableLLM`, which can be toggled at runtime, short-circuits the loop when LLM guidance is disabled so that the controller operates purely under local policies.

LLM switcher and controller integration: Planner decisions are consumed by an `LLMSwitcher` component that subscribes to the `PlannerClient`'s decision callback. Upon receiving a new `PlannerResp`, the switcher maps the textual mode label to a Boolean intent indicating whether combat is desired (e.g., any label containing "combat" is interpreted as a combat intent). To avoid erratic behaviour, the switcher maintains the last requested intent and a counter of how many consecutive planner decisions agree with it; a mode change is only enacted once a configurable number of consistent decisions have been observed. In addition, a minimum dwell time `minDecisionHold` is enforced via a time stamp, so that mode switches cannot occur more frequently than this lower bound even if the planner changes its recommendation. When these conditions are satisfied, the switcher calls `ForceMode` on the `HierarchicalControllerRuntime`, passing the chosen mode together with a time horizon derived from the planner's `expires_in` value, and instructs the `PlannerClient` to defer further requests for the same duration. For evaluation and debugging, the switcher also exposes the latest mode and rationale strings, which are recorded by the logging infrastructure described below.

Baseline rule-based meta-controller and alternate backend.: Before introducing the LLM planner, mode switching was governed by a purely rule-based state machine that inspected a small set of game variables (enemy detection and visibility, distance to the enemy, and health values) to decide between navigation and combat. In parallel with the LLM agents implementation described above, a second variant of the meta-policy was developed by another group member using a remote OpenAI API as the planner backend, plugged into the same HTTP interface as the local `/plan` service.

V. Implementations of different LLMs

This section describes how the original group project was extended with local large language models (LLMs) and how these models are connected to the existing reinforcement learning (RL) agents at run time. The emphasis is on the structure and behaviour of the system rather than on code details.

1) Baseline project: The starting point is a Unity project based on the Unity ML-Agents framework and a first–person shooter (FPS) training environment. The environment contains a player character and one or more enemies placed in an arena. The original group implementation provides three main types of RL agents.

The first type is a navigation agent that controls the locomotion of the player. During training this agent observes quantities such as the direction and distance to a goal, the current velocity and orientation of the player, and a grounded flag. It produces continuous actions for rotation, forward motion and optional jumping. The training script resets the episode by randomising the player and enemy positions and terminates the episode when the goal is reached, when the player falls below a height threshold, or when a time limit is exceeded. At test time a simplified runtime version of the navigation agent is used. This runtime agent delegates all movement to the shared `CharacterController`, no longer performs resets, and can be enabled or disabled by other components.

The second type consists of combat agents that handle aiming and shooting. Their observations include the enemy position in the player's local coordinate frame, the approximate distance to the enemy, the player's current health and a binary indicator of whether the enemy is alive. The actions correspond to yaw rotation and a firing trigger. Reward shaping encourages accurate aiming and damage dealt to the enemy, while penalising damage received, blind firing and long episodes. The training scripts are responsible for resetting both the player and the enemy at the start of each episode and for terminating the episode when either side dies or a maximum step count is reached. As with navigation, a runtime variant is used in the game scenes, which only runs inference, keeps a reference to the current target and exposes a method for higher-level controllers to update this target.

The third type is an end-to-end agent that combines navigation and combat into a single policy. In addition to enemy position and player velocity, this agent observes information about a health-pickup object and whether the player is grounded. It is trained to approach and defeat the enemy while optionally visiting the health pickup when health is low. The corresponding runtime script assumes that enemies and pickups already exist in the scene and focuses on resetting the player position at the start of each run.

The extensions introduced in this work do not alter the training procedures of these RL agents. Instead, they add LLM-based components on top of the existing runtime agents.

2) OpenAI-style LLM client: To support multiple local LLM backends, a generic client class was introduced. This client communicates with an HTTP endpoint that follows the OpenAI `/v1/chat/completions` API, which is provided by tools such as GPT4All, LM Studio and Ollama when configured in "OpenAI-compatible" mode.

The client exposes a single asynchronous method that accepts a textual prompt describing the current game state and returns a discrete high-level mode. Only two modes are used: `Navigation` and `Combat`, represented by an enumeration. Internally the client constructs a JSON payload with a fixed system message explaining the task and a user message containing the state description. The JSON response is parsed and reduced to a single token; if the token contains the word `COMBAT` the combat mode is selected, otherwise the client falls back to navigation. In case of HTTP errors or parsing failures the client also defaults to navigation, so that the game remains playable even if the LLM process is not available.

Because the client depends only on a base URL and model identifier, it can be pointed at different local engines simply by changing these parameters. No further changes are required in the Unity logic when swapping models.

3) LLM-assisted hierarchical controller: The main integration effort is an LLM-assisted hierarchical controller attached

to the player object. This controller corresponds to an extended runtime version of the group's hierarchical controller and coordinates the navigation and combat agents using high-level decisions from the LLM.

The controller holds references to the navigation and combat runtime agents, the player's health component and tag-based detectors for enemies and health pickups. Internal state includes a flag indicating whether the system is currently in navigation or combat mode, the current target transform and its health component, as well as timers that measure how long an enemy has been visible or hidden.

On each frame the controller first checks terminal conditions such as the player reaching zero health or the current target dying. If a terminal condition is detected, both sub-agents are disabled, their episodes are ended via ML-Agents, and the current run is marked as finished.

If the episode is still active, the update logic depends on the current mode. In navigation mode the controller first evaluates whether the player's health has dropped below a configured threshold. If health is low, it searches the scene for objects with the health-pickup tag and selects the closest active pickup within a search radius. When such a pickup is found, it becomes the navigation goal and the combat target is cleared, causing the navigation agent to move towards the pickup.

If the player is not in a low-health state, the controller searches for enemies. It selects the closest alive enemy based on distance and stores references to its transform and health component. The enemy position is passed to the navigation agent as its goal, so that the navigation policy moves towards it. In addition, the controller computes the distance to the enemy and evaluates line of sight using a raycast from an "eyes" transform, updating timers that track how long the enemy has been seen or hidden.

At a fixed cooldown interval the controller summarises the current situation in a short textual description. This description contains Boolean indicators for enemy detection, visibility, low-health status and availability of a health pickup, as well as numerical values such as distance and the current health of the player and target. The summary is sent to the LLM client, which returns the recommended high-level mode. The returned mode is cached until the next query.

If the LLM recommends combat and the local conditions also satisfy a set of safety checks (enemy visible, within an engagement distance, visible for at least a minimum time, and player not in a low-health state), the controller switches to combat mode. Switching mode disables the navigation agent, enables the combat agent and passes the current target to the combat agent via its target-setting method. The combat agent then takes over aiming and shooting.

While in combat mode the controller continues to monitor distance and line of sight. If the enemy has been hidden for longer than a specified timeout and is farther away than a disengagement distance, the controller exits combat mode, re-enables the navigation agent and clears the combat target. If the player's health falls below the threshold during combat and a health pickup is available, the controller also leaves combat mode and redirects navigation towards the pickup. In each direction, mode switches are achieved purely by enabling or disabling the relevant runtime agents and updating their goals; the underlying RL policies remain unchanged.

The controller logs the generated prompts and the decisions returned by the LLM to the Unity console. These logs provide a convenient way to verify that the local LLM is active and that Unity is applying the resulting high-level actions.

4) Simple LLM-based combat controller: In addition to the hierarchical setup, a simpler controller was created in which the LLM determines only whether the player should fire the weapon. This component uses the standard Unity movement and weapon systems without involving the navigation agent.

Each frame the controller identifies the nearest alive enemy using a sphere query around the player and checks line of sight. At a small interval it builds a compact prompt that encodes whether an enemy is detected, whether the enemy is visible, the distance to the enemy and the health of both the player and target. The system message instructs the LLM to output exactly one of the tokens `NAVIGATE` or `COMBAT` according to a strict rule: shoot only if the enemy is visible and within a fixed engagement range, otherwise navigate.

When the cached mode is `COMBAT` and a valid target is in range, the controller rotates the player smoothly towards the target and repeatedly triggers the shooting method on the weapon manager. When the mode is `NAVIGATE` or no valid target exists, shooting is disabled. This setup serves as a minimal, interpretable example of how a language model can gate actions in an FPS context.

5) Local LLM backends: All LLM components are designed to run entirely on the local machine, without external cloud services. Three tools are used as backends: GPT4All, LM Studio and Ollama. Each tool can expose an HTTP endpoint that emulates the OpenAI chat API; typical default ports are `localhost:4891` for GPT4All, `localhost:1234` for LM Studio and `localhost:11434` for Ollama. The LLM client is configured with the appropriate base URL and model name for the chosen tool.

To check that a backend is running and reachable, short `curl` commands are issued from a PowerShell terminal to list models or send a minimal test query. During Unity execution, the debug logs produced by the controllers confirm that prompts are sent to the selected model and that valid decisions are received, demonstrating that the local LLM is actively influencing the agent behaviour.

VI. Experimental Evaluation and Results

We evaluate both (i) the learning dynamics of the three trained RL sub-behaviors (Navigation-to-LOS, Navigation-to-Target, and Combat) and (ii) the end-to-end hierarchical controller with and without LLM guidance. For the LLM-guided condition, the external planner is instantiated as the locally hosted Ollama service described in Section IV, queried via the `PlannerClient` and `LLMSwitcher` components. We first analyze training dynamics using TensorBoard logs exported during PPO training, and then compare online episodes with

the planner enabled versus disabled (toggling the `enableLLM` flag) using metrics derived from the `StatsLogger` CSV logs.

A. Learning Curves from TensorBoard

Figures 1 and 2 illustrate the training dynamics for our three trained low-level behaviors: Navigation-to-LOS, Navigation-to-Target, and Combat. These curves reflect how quickly each policy converges, how stable its final performance is, and whether the reward shaping produces the intended behaviors.

Navigation-to-LOS: The episode length for Navigation-to-LOS decreases with training, dropping quickly, then stabilizing around 105 steps. This is higher than other sub-behaviors because it was trained in an environment with more occluding walls. The greater arena complexity also increases variance, as random spawns significantly affect the episode's length. The cumulative reward rises rapidly and steadily, converging by 3M steps. Its final value exceeds other sub-behaviors, reflecting more generous reward shaping rather than better performance.

Navigation-to-Target: The Navigation-to-Target agent begins with long and highly variable episode lengths (up to 140 steps), reflecting inefficient wandering and occasional collisions early in training. As learning progresses, episode length decreases steadily and stabilizes around $\sim$ 30 steps, indicating that the agent reaches the target efficiently and with minimal oscillation. The cumulative reward curve climbs monotonically and converges around 0.83 by 7.3M training steps. This stable plateau indicates that the agent consistently executes a smooth, direct trajectory toward the pickup.

Combat: The Combat agent improves quickly, with cumulative reward rising sharply during the first 0.5–1.0M steps and briefly peaking near 1.0 as effective aiming behavior emerges. As entropy decreases, the policy settles to a lower plateau around 0.82, indicating a more conservative and somewhat suboptimal strategy. Episode length stabilizes near 90 steps, showing that the agent wins reliably but slowly. Overall, the behavior is stable but not fully optimized, and likely requires adjusted reward shaping or a slower entropy schedule rather than simply more training.

Interpretation: Together, these results confirm:

- The Navigation-to-Target agent quickly learns efficient point-to-point movement and stabilizes with short, consistent trajectories.
- The Navigation-to-LOS agent learns to position for clear sightlines, steadily reducing episode length and converging, though at more variable steps due to occluding walls.
- The Combat agent achieves reliable long-horizon engagement behavior.
- All policies converge smoothly and maintain performance for millions of steps, making them suitable for integration under a high-level meta-controller.

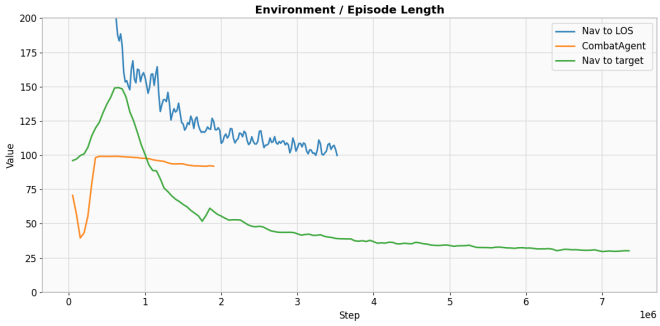

Fig. 1. **Episode length during training.** The Navigation-to-Target agent (green) rapidly reduces its episode length from $\sim$ 140 to $\sim$ 30 steps as it learns to reach the pickup efficiently. The Combat agent (orange) stabilizes around $\sim$ 90 steps, reflecting longer multi-action engagements. The Navigation-to-Los agent (blue) stabilizes around $\sim$ 105 steps, it appears to take longer compared to Navigation-to-target, but it's due they uses a different reward system and environment.

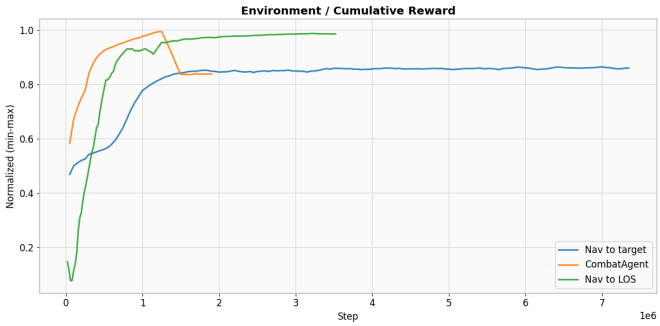

Fig. 2. **Cumulative reward during training.** The Navigation-to-Target agent (blue) and Navigation-to-Los(green), both exhibits a smooth monotonic rise, converging around .83 and 0.99 respectively. The Combat agent (orange) improves rapidly, briefly peaks near 1.0, and stabilizes near 0.82.

B. Results of different LLMs

This section compares the behaviour of the three LLM-driven agents – LMStudio (deepseek-r1-qwen3-8b), ollama (gemma-3-4B), and GPT4ALL (Mistral Instruct) – in the end-to-end scenario. The analysis focuses on how distance to the enemy relates to damage dealt (distance vs. targetHP) and how ammunition is consumed over the course of each episode (ammo vs. time).

C. LMStudio (deepseek-r1-0528-qwen3-8b)

The LMStudio agent exhibits a relatively cautious, mid-range engagement style. In the distance–targetHP scatter plot for LMStudio (Figure 3), most points cluster at high targetHP values (around 100) across a wide range of distances. This indicates a long exploratory and approach phase in which the agent is closing the distance but not yet dealing damage. Once the agent reaches mid-range (approximately 30–40 units), targetHP begins to decline, and further reductions occur as distance decreases into the close-range band. The colour-coding of visibility shows that almost all damaging interactions occur while the enemy is visible, consistent with a policy that only fires when it has a clear line of sight.

The ammo–time curve in Figure 4 supports this interpretation. Ammunition remains close to its maximum value for much of the early episode, confirming that the agent spends significant time manoeuvring without firing. After the first engagement, there is a sharp drop in ammo, followed by a sustained period at or near zero. This suggests that once LMStudio commits to combat, it tends to expend a large portion of its ammunition in a relatively concentrated time window, often leaving a long interval with no remaining ammunition. Overall, LMStudio behaves as a mid-range "poker": it approaches cautiously, begins damaging the target from moderate distance, and then commits a burst of fire that rapidly consumes its resources.

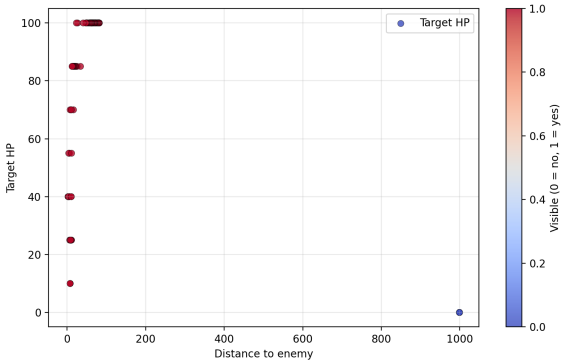

Fig. 3. Distance to enemy versus target HP for the LMStudio (deepseek-r1-0528-qwen3-8b) agent.

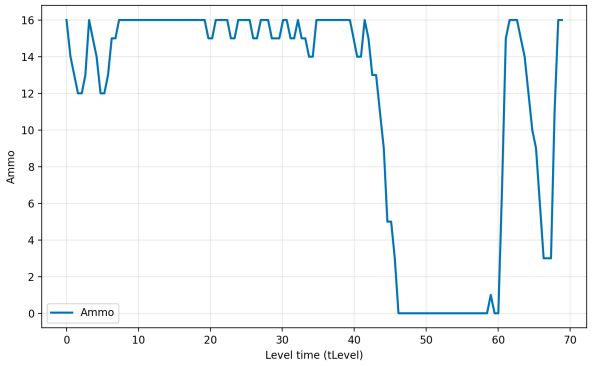

Fig. 4. Ammo over level time for the LMStudio (deepseek-r1-0528-qwen3-8b) agent.

D. ollama (gemma-3-4B)

The ollama/gemma agent shows a distinctly different pattern, more consistent with a close-range, high-commitment style. In the gemma distance–targetHP scatter plot (Figure 5), the majority of points with targetHP equal to 100 are spread across distances from the starting point down towards the low-teens, indicating that the agent continues to close the gap for quite some time without inflicting damage. The first substantial reduction in targetHP typically occurs only once the agent is very close to the enemy (roughly 8–10 units). After that point, targetHP drops sharply through several discrete values (for example, 55, 40, 25, 10), all at very short distances. This pattern reflects a "dive-in" engagement: the agent delays firing until it is almost on top of the target, then deals most of its damage in a compact close-range burst.

The gemma ammo–time plot in Figure 6 is consistent with this behaviour. Ammo remains high and nearly flat during the entire approach phase, then decreases rapidly over a relatively short time interval once close-range combat begins. In several sequences, the agent runs its ammunition down to zero exactly when the targetHP is already substantially reduced. This tight coupling between late-phase ammo usage and rapid HP drop reinforces the picture of gemma as a close-range brawler: it commits fully once it decides to fight, willingly trading HP and ammo in an intense, short engagement rather than poking from distance.

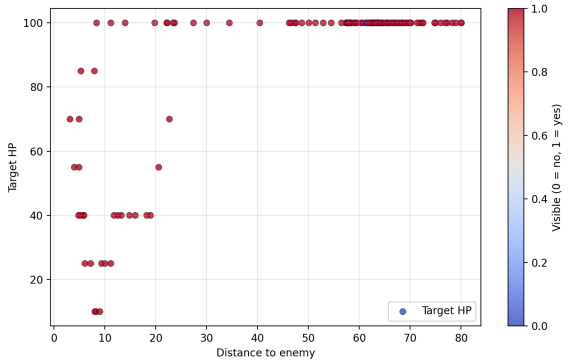

Fig. 5. Distance to enemy versus target HP for the ollama (gemma-3-4B) agent.

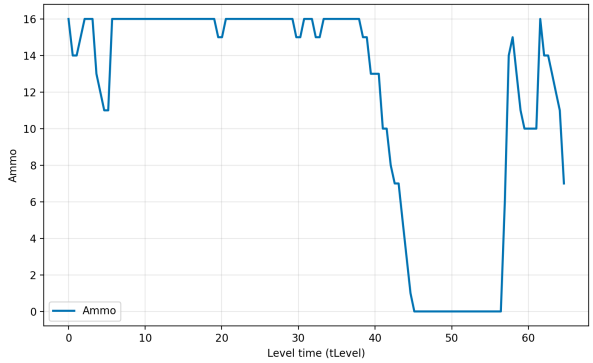

Fig. 6. Ammo over level time for the ollama (gemma-3-4B) agent.

E. GPT4ALL (Mistral Instruct)

The GPT4ALL/Mistral agent demonstrates a more opportunistic and temporally earlier engagement pattern than the other two models. In the GPT4ALL distance–targetHP plots (Figure 7), reductions in targetHP appear noticeably earlier in the episode than for LMStudio or gemma. In one of the runs, the first drop from 100 HP occurs around 18–19 seconds at a mid-range distance of roughly 20–25 units, rather than waiting until

very short range. Subsequent points show continued damage at comparable or slightly shorter distances, indicating that GPT4ALL is willing to begin fighting as soon as it reaches a workable firing distance rather than pushing all the way into melee range.

The ammo–time plot in Figure 8 further highlights this behaviour. Ammunition starts decreasing significantly earlier in the episode than for the other agents, suggesting that GPT4ALL begins firing sooner and distributes its shots over a longer period. Compared with gemma, there is less of a single steep "ammo cliff" and more of a gradual decline, implying a more measured rate of fire. Unlike LMStudio, GPT4ALL does not spend as long in a pure navigation phase before engaging, nor does it concentrate all of its ammo expenditure into a single late burst. Together, these patterns characterise GPT4ALL as a mid-range skirmisher: it engages relatively early, maintains pressure over time, and does not rely solely on a final all-in close-range attack.

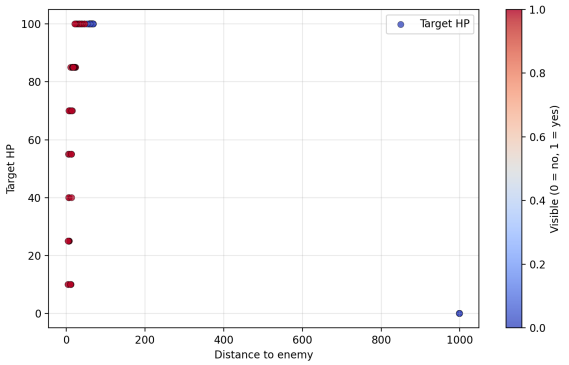

Fig. 7. Distance to enemy versus target HP for the GPT4ALL (Mistral Instruct) agent.

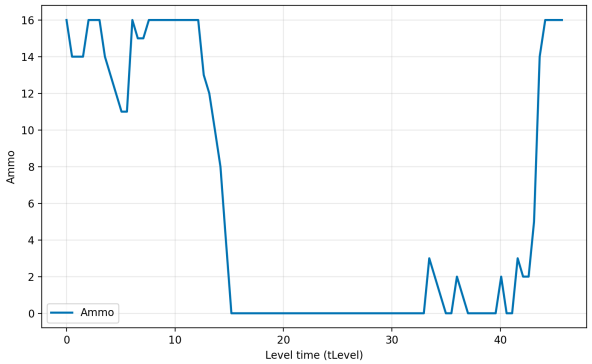

Fig. 8. Ammo over level time for the GPT4ALL (Mistral Instruct) agent.

VII. Summary and Discussion

In this project we implemented two concrete variants of the high-level meta-policy: one using a remote OpenAI API and one using a locally hosted model served through the Ollama runtime. A systematic empirical comparison between the OpenAI and Ollama variants is planned as part of the final evaluation in the coming weeks.

We present a hierarchical controller for FPS NPCs that combines an LLM meta-policy with specialized RL sub-policies for navigation and combat. The design exploits complementary strengths: low-latency motor control from RL (trained with PPO) and higher-level intent selection from the LLM. In Unity, we trained (i) Navigation-to-LOS to position for unobstructed sightlines to the enemy, (ii) Navigation-to-Target to reach pickups, and (iii) Combat to track and fire under continuous yaw/pitch control.

In the current hierarchical runtime, however, only two of these behaviors: Navigation-to-Target and CombatAgent are integrated. The Navigation-to-LOS module, while trained and validated in isolation, is not yet connected to the switching framework. Similarly, the LLM-based meta-policy responsible for high-level intent selection is fully designed and partially implemented, but not yet activated or evaluated in end-to-end control.

As a result, the present evaluation focuses on the individual RL sub-behaviors rather than the full hierarchy. For the final version of the project, we will (i) integrate the Navigation-to-LOS policy into the runtime controller, (ii) activate and evaluate the LLM-based planner for real-time intent switching, and (iii) conduct systematic comparison across RL-only and LLM guided controllers using metrics such as win-rate, time-to-kill, damage, and switch latency.

The work completed so far establish the full set of low-level behaviors and the infrastructure required for hierarchical control, providing a solid foundation for the remaining integration and evaluation tasks.

A. Empirical takeaways on the RL sub-behaviors

Learning curves show stable convergence for all sub-policies. Navigation-to-Target quickly achieves short, consistent trajectories. Navigation-to-LOS converges at longer and more variable episode lengths due to greater maze complexity and spawn variance. The ComabtAgent showed rapid initial improvement, briefly peaking near 1 before settling to a lower plateau around 0.82, suggesting that the policy adopted a more conservative strategy as entropy decreased. Notably, the higher final cumulative reward for LOS vs. Target reflects more generous shaping rather than superior task performance. These properties make the policies suitable as reusable skills under a switcher.

B. Summary of contributions (Ollama-based integration)

The hierarchical controller used in this work is built around two low-level policies trained with Unity ML-Agents: a navigation policy that controls continuous locomotion and a combat policy that controls aiming and firing. The main contribution of the Ollama-based implementation is to wrap these policies in an LLM-guided meta-controller that remains compatible with the real-time constraints of the FPS setting while being fully deployable on a local machine. Concretely, a Unity-side `PlannerClient` constructs a compact observation summary

(PlannerObs) and issues HTTP requests to a planner endpoint on localhost that is backed by the Ollama runtime. The planner returns high-level "navigate" versus "combat" intents together with a time-to-live parameter, which a dedicated LLMSwitcher component smooths and applies as mode overrides on the HierarchicalControllerRuntime.

From a systems perspective, this implementation shows that an open-weight LLM served via Ollama can function as a low-frequency, advisory meta-policy without requiring any changes to the underlying Unity ML-Agents training pipeline. The planner runs at a substantially lower effective frequency than the 10–20 Hz control loop thanks to a send-on-change mechanism and planner-specified time-to-live values, and it remains strictly optional via the enableLLM toggle. In addition, a StatsLogger component records the controller state and planner decisions to CSV, providing a concrete measurement and analysis pipeline for future quantitative comparison of LLM-on versus LLM-off episodes under the same task configuration and, eventually, for direct comparison against the OpenAI-based backend.

C. Discussion of design choices and relation to the OpenAI variant

The Ollama-based meta-policy is intentionally positioned as an advisory layer rather than a hard dependency: when enableLLM is disabled, the hierarchical controller falls back to its local heuristics and RL sub-policies, and when it is enabled, the planner can only request mode changes at a relatively low rate. This design reflects the latency and robustness concerns highlighted in prior work on LLM-in-the-loop control, and it ensures that occasional planner delays or errors cannot destabilize the fast control loop. The decision smoothing implemented in LLMSwitcher—requiring multiple consistent planner outputs and enforcing a minimum dwell time between switches—further reduces the risk of oscillatory behaviour due to noisy or indecisive LLM responses.

From an integration standpoint, the Ollama-based planner shares the same meta-policy interface as the OpenAI-based variant developed by another group member: both consume the compact numeric summary of the game state produced by PlannerClient and emit discrete mode intents plus textual rationales, while the rest of the system (mode application in HierarchicalControllerRuntime and telemetry in StatsLogger) remains unchanged. The key difference lies in deployment: the OpenAI variant relies on a remote API, whereas the present implementation uses a locally hosted Ollama backend. This common interface will make it possible to compare the two backends under identical conditions by simply switching the planner endpoint, without modifying the low-level control logic or logging pipeline.

D. Limitations and ongoing work

The current Ollama-based integration is subject to several limitations. First, it has so far been exercised only in the 1v1 arena scenario with comparatively simple map geometry and a single enemy type, so it does not yet probe how well the meta-policy scales to more complex tactical situations. Second, the planner operates on a deliberately compact numeric observation (PlannerObs) rather than richer inputs such as images, trajectories, or natural-language descriptions of the map, and its action space is restricted to binary mode choices between navigation and combat. Third, the prompting and model configuration within Ollama are relatively simple and have not been systematically tuned, and the number of logged LLM-on and LLM-off episodes at this stage is insufficient for a rigorous statistical comparison of performance.

These limitations point to several directions for future work. In the short term, we plan to (i) collect larger sets of episodes with the Ollama-based planner enabled and disabled, using the StatsLogger metrics to quantify time-to-kill, time-to-death, distance statistics, and mode-switching patterns; and (ii) run matched experiments where the only difference is whether the meta-policy backend is the OpenAI API or the locally hosted Ollama model, thereby enabling a direct empirical comparison between the two implementations under a shared evaluation protocol. In the longer term, the same integration pattern could support richer intent spaces (e.g., flank, retreat, reload, hold-cover), more expressive planner inputs, or even learned meta-policies that are trained to imitate or augment the decisions of the LLM. In all cases, the existing Ollama-based implementation provides a concrete and extensible starting point for these experiments.

E. Discussion with opposing group

As far as discussion with the opposing group go, there haven't been many. In the beginning, we had decided to build the project in the same environment so we could potentially present our model against theirs, and they even suggested an environment that our group is still using until today. Unfortunately, the other group changed their environment due to some issues and informed us about their decision. No further discussions were made.

References

[1] Y. Du *et al.*, "Guiding pretraining in reinforcement learning with large language models (ellm)," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[2] U. Ruiz-Gonzalez *et al.*, "Words as beacons: Guiding rl agents with high-level language prompts," in *NeurIPS Workshop on Open-World Agents*, 2024.

[3] S. Basavatia *et al.*, "Starling: Self-supervised training of text-based rl agent with large language models," *Preprint*, 2024.

[4] S. Liu *et al.*, "Rl-gpt: Integrating reinforcement learning and code-as-policy," *Preprint*, 2024.

[5] M. Klissarov *et al.*, "Maestromotif: Skill design from artificial intelligence feedback," *Preprint*, 2024.

[6] S. Rho *et al.*, "Language-guided skill discovery (lgsd)," in *International Conference on Learning Representations (ICLR)*, 2025.

[7] M. Yildirim *et al.*, "Highwayllm: Decision-making and navigation in highway driving with rl-informed language model," *Preprint*, 2024.

[8] Y. Zhuang *et al.*, "Yolo-marl: You only llm once for multi-agent reinforcement learning," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, 2025.

[9] X. Wan *et al.*, "Think twice, act once: A co-evolution framework of llm and rl for large-scale decision making (ace)," in *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

APPENDIX A
WEEKLY SUMMARY OF CONDUCTED WORK (PROJECT DIARY)

Week 37 – Project definition and planning
Defined the project topic (hierarchical LLM–RL control for an FPS NPC) and agreed on group roles. Produced the initial project plan with milestones for RL sub-behaviors, LLM integration, and evaluation. Sketched the high-level architecture with a slow meta-policy over fast RL sub-policies.

Week 38 – Environment and tooling setup
Set up the development environment: Unity, ML-Agents, Python dependencies, version control, and logging tools. Verified that a simple ML-Agents policy could be trained and run in Unity. Investigated options for LLM access (OpenAI API versus local/open-weight models through HuggingFace/Ollama/GPT4All) and confirmed that a local HTTP planner service would be feasible.

Week 39 – Game rules, RL design, and planner API
Refined the FPS scenario (1v1 arena, time limits, health system, basic weapons). Defined observation, action, and reward structures for navigation and combat sub-behaviors, including continuous yaw/movement and firing actions. Designed a lightweight planner API (JSON over HTTP) with a compact state summary and discrete *mode* output for the LLM-based meta-policy. Sketched a baseline rule-based meta-controller as a fallback.

Week 40 – Navigation environments and first RL training
Implemented the first navigation training environment in Unity ML-Agents and started training with PPO. Iterated on reward shaping (progress to goal, penalties for collisions and getting stuck). Defined and started implementing a second navigation variant focusing on reaching targets (pickups), which later became the Navigation-to-Target behavior described in the report.

Week 41 – Navigation refinement and combat environment design
Stabilized navigation training by tuning hyperparameters and reward terms. Finalized the geometry and observation space for the arena and cover/line-of-sight logic. Designed the combat training environment (moving enemy, continuous yaw/pitch, firing) and specified rewards for aim alignment, accurate shots, and kill events, together with penalties for wasted shots and timeouts.

Week 42 – Combat training and first hierarchical controller
Implemented the Combat sub-behavior in ML-Agents and began PPO training, monitoring learning curves via TensorBoard. Integrated trained navigation and combat policies into a single Unity scene, wrapped by a first version of a hierarchical controller that switched between them using simple rule-based logic (distance, visibility, health, enemy-detected flags). Validated that the "RL-only + rule-based switching" baseline worked end-to-end.

Week 43 – Navigation variants and logging infrastructure
Split navigation into conceptually distinct variants (navigation to line-of-sight versus navigation to pickups) and finalized the Navigation-to-Target and Navigation-to-LOS training setups. Continued training all sub-policies to convergence. Implemented a `StatsLogger` component to log health, distance, visibility, active mode, and other telemetry to CSV for later analysis. Defined evaluation metrics such as win rate, time-to-kill/death proxies, distance statistics, and mode-switch counts.

Week 44 – LLM planner integration (Ollama / OpenAI backend)
Implemented the `PlannerObs` and `PlannerResp` structures and the `PlannerClient` component that sends JSON-encoded observations to a `/plan` HTTP endpoint and parses returned decisions. Integrated a first LLM backend (OpenAI API and a local model served via Ollama) behind this endpoint. Added basic throttling via query intervals, timeouts, and planner-provided `expires_in` to keep the planner frequency much lower than the 10–20 Hz RL control loop.

Week 45 – LLMSwitcher, robustness, and early LLM-on runs
Developed the `LLMSwitcher` logic to smooth planner outputs: requiring multiple consistent decisions, enforcing minimum dwell time, and preventing rapid mode oscillations. Added the `enableLLM` toggle and failover behavior so the controller defaults to RL-only if planner calls fail or are disabled. Ran short batches of episodes with LLM-on versus LLM-off to verify stability, collect initial `StatsLogger` logs, and qualitatively inspect planner rationales and switching patterns.

Week 46 – First draft of report and protocol definition
Consolidated TensorBoard learning curves for Navigation-to-Target, Navigation-to-LOS, and Combat, and documented their convergence behavior. Wrote the first complete draft of the report (introduction, related work, formal problem description, methodology, and initial results), including the description of the hierarchical controller, the LLM-based planner integration, and the planned evaluation protocol for comparing RL-only and RL+LLM conditions.

Week 47 – Additional LLM models (Mistral Instruct via GPT4All)
Extended the meta-policy backend to support an additional local model: Mistral Instruct served through GPT4All. Adapted the `/plan` endpoint to allow switching between backends without changing the Unity-side code. Ran qualitative tests to compare Mistral-based decisions with the original backend in terms of latency, stability, and typical "navigate vs. combat" choices. Logged representative episodes for later analysis.

Week 48 – DeepSeek R1 via LM Studio and RL vs. RL+LLM matches
Integrated DeepSeek R1 via LM Studio as a second alternative planner backend using the same `PlannerObs`/`PlannerResp` interface. Conducted small-scale experiments where the RL-only baseline agent was run

against the hierarchical LLM+RL agent to gain intuition about behavioral differences and failure modes. Collected CSV logs for all three configurations (RL-only, RL+LLM with Mistral, RL+LLM with DeepSeek) to understand how different models affect switching behavior and the combat/navigation balance.

Week 49 – Consolidation of experiments and report refinement
Reviewed and cleaned the logged data from the different planner backends. Updated plots and text in the report to better explain the architecture (`PlannerClient`, `LLMSwitcher`, `HierarchicalControllerRuntime`, and `StatsLogger`) and the evaluation protocol. Added a more detailed discussion of limitations, planned ablations, and future work (e.g., richer intent spaces and more complex arenas). Started drafting slides and a demonstration plan for the final presentation.

Week 50 – Finalization and presentation
Polished the written report (language, figures, structure) and ensured that the implementation details matched the final code. Prepared and rehearsed the presentation, including a live or recorded demo of the RL-only versus LLM+RL agent behavior. Incorporated final feedback from the course staff and submitted the final report and code.