LLM Guided Search-and-Rescue with Reinforcement Learning

Farhan Mohammed
Raisulhaq Mohammed Rizwan
Muhammad Qasim

School of Computing, Wichita State University

Abstract

We introduce a hierarchical method that combines a Deep Q-Network (DQN) agent with Large Language Models (LLMs) to speed up search-and-rescue (SAR) operations in urban environments. The system uses an autonomous agent that has RGB vision, LiDAR measurements, and a direction vector. The DQN agent uses these sensor inputs to accomplish low-level navigation, whereas the LLM functions as a high-level planner that provides waypoints to guide the agent towards victim locations more efficiently than unguided search strategies. The combined architecture is designed to address sparse-reward issues, reduce unnecessary exploration, and improve route selection in crowded locations. We assess our system in a 3D environment with obstacles simulated by PyBullet and demonstrate that LLM-guided exploration performs better than pure RL baselines.

1. Introduction

Search-and-rescue (SAR) in cluttered urban or forest-like environments is difficult for autonomous UAVs because they must search large spaces, avoid obstacles, and handle sparse or delayed rewards. Pure reinforcement learning (RL) can learn effective local navigation, but it often wastes time in irrelevant regions, requires heavy reward shaping, and struggles with long-horizon missions where the victim's location is uncertain.

Our work proposes a hybrid SAR framework that combines a LiDAR-based Deep Q-Network (DQN) for low-level obstacle avoidance with a large language model (LLM) that performs high-level waypoint planning from natural-language mission descriptions. The LLM biases the search toward promising directions and intervenes when the agent is stuck, while the DQN remains responsible for continuous control. The system is evaluated in a 3D PyBullet environment with dense obstacles and human targets, showing that LLM-guided navigation can achieve more reliable and efficient missions than purely RL-based baselines.

2. Problem Formulation

2.1. Task Definition and Markov Decision Process (MDP)

We define our search-and-rescue (SAR) task as a finite MDP

$$M = (S, A, P, R, \gamma, G),$$

where S is the state space,
A is the action space of the agent,
$P(s_{t+1} \mid s_t, a_t)$ is the state transition matrix,
$R(s_t, a_t, s_{t+1})$ is the reward function,
γ is a discount factor $\gamma \in [0, 1]$,
G is set of valid goal locations

A mission defines a single goal position $g \in G$. The LLM acts as a high level planner that produces a list of waypoints,

$$w = (w_1, w_2, w_3, \ldots, w_k), \qquad w_k \in R^3,$$

given the mission details and environment information. At any time t, the RL agent is given the current waypoint w_t, and the objective of the agent is to learn the policy $\pi(a_t \mid s_t)$ that maximizes discounted return.

2.2. State Space

At time t, the system's state $s_t \in S$ captures all the information related to the agent's current position and orientation, sensor data, waypoint provided by the LLM, and direction vector. The state of the agent at time t can be defined by,

$$s_t = (p_t, o_t, I_t, L_t, D_t, w_t, d_t),$$

where $p_t \in R^3$ is the agent's position as co-ordinates (x_t, y_t, z_t),
$o_t \in$ is the orientation of the agent,
$I_t \in R^{H \times W \times 3}$ is the RGB image from the camera,

$L_t \in R^n$ is the LiDAR measurements,

$D_t \in R^3$ is the direction vector towards signal or victim location,

$w_t \in R^3$ is the current waypoint provided by the LLM,

$d_t \in R$ is the distance from agent to waypoint w_t

The distance from agent to the current waypoint is represented as,

$$d_t = || p_t - w_t ||$$

2.3. Action Space

Action space A consists of navigation commands executed by the agent at each step.

$$A = \{a^{fwd}, a^{back}, a^{right}, a^{left}, a^{up}, a^{down}, a^{yaw+}, a^{yaw-}, a^{hover}\}$$

Each $a_t \in A$ transforms the environment by updating the agent's position and orientation,

$$(p_{t+1}, o_{t+1}) = f_{env}(p_t, o_t, a_t),$$

where f_{env} is the environment dynamics defined by PyBullet

2.4. Reward Function

The reward function r_t is represented as a combination of progress reward that encourages movement toward the goal, proximity penalty that discourages flight too close to obstacles or going out-of-bounds based on LiDAR measurements, collision penalty for contact with obstacles, and success reward given upon reaching the goal.

$$r_t = r_t^{progress} + r_t^{proximity} + r_t^{collision} + r_t^{success}$$
$$r_t = (d_{t-1} - d_t) + P(l_t) + Cc_t + R_{success}s_t$$

where d_t is the distance from the agent to the goal at time t,

l_t is the LiDAR's measurements,

$c_t \in \{0, 1\}$ is the collision flag,

$s_t \in \{0, 1\}$ is the success flag,

$C < 0$ is the collision penalty,

$R_{success} > 0$ is the success reward

2.5. Termination Conditions

Each search-and-rescue mission gets a single goal position $g \in G$, where G represents all valid victim locations in the environment. We define three termination conditions:

- Success: The agent reaches the goal g before timeout or collision ($s_t = 1$). The termination flag for success is,

$$s_t = \begin{cases} 1, & \text{if } d_t < \varepsilon_{\text{goal}} \text{ (goal reached)} \\ 0 \end{cases}$$

- Collision: The agent collides with obstacle before reaching the goal or timeout ($c_t = 1$). The termination flag for collision is,

$$c_t = \begin{cases} 1, & \text{(if collision detected)} \\ 0 \end{cases}$$

- Timeout: The episode ends without success or collision when maximum time limit is reached ($T = T_{max}$)

$$T = min\{\, t \geq 0 \mid s_t = 1 \; or \; c_t = 1 \; or \; t = T_{max} \,\}$$

3. System Architecture

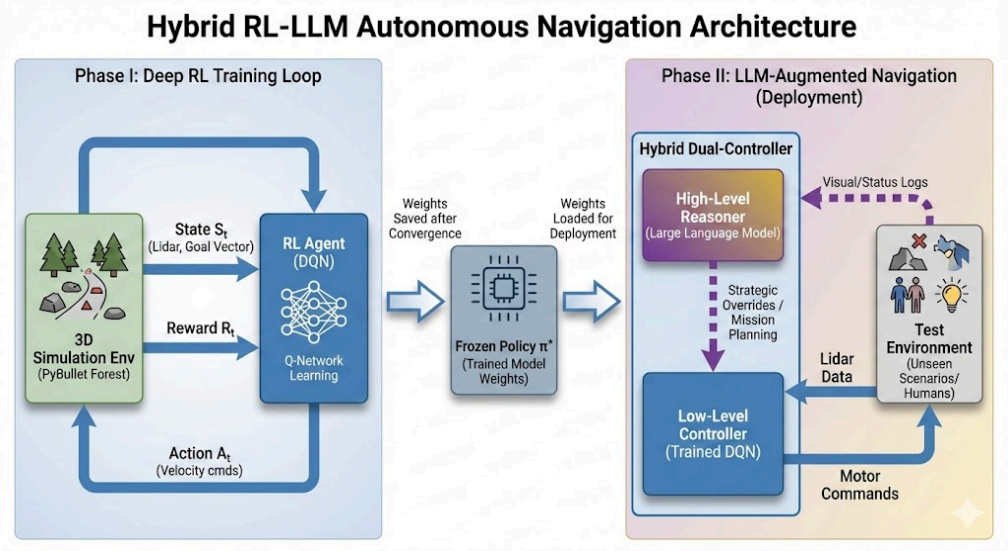


Figure 1: Architecture of the SAR system

The LLM functions as a high-level planner, reasoning over mission descriptions and progress, while a frozen DQN policy works as the low-level navigation controller. The LLM receives a mission objective with an approximate distress direction at the beginning of each episode, and it provides an initial waypoint sequence in world coordinates. The DQN-based controller uses these waypoints as its "goal position" after passing them via an intermediary translation layer that chooses the waypoint that is currently active. At each stage of the simulation, the low-level loop is executed. As the state input, the environment supplies LiDAR data, normalized distance to the objective, and distance to the waypoint.

The agent proceeds toward the waypoint by applying the discrete actions that the DQN outputs. The LLM does not participate in reward computation or alter the reward function. Instead, the environment uses the reward function to compute the reward from this transition and updates the episode termination flags.

The LLM is only activated during certain high-level occurrences. When the agent is detected to be stuck, or whenever the current list of waypoints is exhausted and a new segment is required, it is called at episode start to generate an initial path segment.

The LLM returns a high-level decision, such as skipping the waypoint, continuing to attempt it, or aborting the mission after receiving the current position, target waypoint, stuck duration, and surrounding obstacle information when it is stuck. The controller then implements this choice without changing the underlying DQN policy. Reaching the goal gives a large terminal reward and concludes the episode, which is considered episode-level success.

Failure occurs when the drone runs out of bounds, collides with an obstruction, or due to timeout. These situations are recognized solely in the environment and sent back to both the DQN and the LLM as terminal outcomes, but only the DQN uses them for learning.All LLM requests and responses, selected waypoints, DQN actions, reward values, and termination reasons are all recorded by the system during deployment. A validation layer that verifies JSON schema, boundary restrictions, altitude, uniqueness, and directional consistency is always applied to LLM outputs.

If a response fails validation, the system retries the same prompt a limited number of times. If all retries fail, it returns to the last valid waypoint or terminates the mission. This integration design places the RL agent in charge of continuous control and learning, while the LLM is limited to high-level waypoint planning and acts as a supervisor within strict safety and validation constraints.

4. Methodology

4.1. Deep Q-Network (DQN)

We use a Deep Q-Network (DQN) as the low-level controller that decides how the agent should move at each step. DQN learns a mapping from observations (LiDAR readings, distance to goal, relative angle) to discrete actions by estimating how good each action is in a given state.

The agent learns an action-value function $Q(s, a)$ that estimates the future reward obtained by taking action a in state s and following the optimal strategy. The optimal Q-function satisfies the Bellman equation:

$$Q^*(s, a) = E\left[R(s, a) + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right]$$

where $\gamma = [0, 1)$ is the discount factor which controls the importance of future rewards.

Since we have a continuous state, we approximate the Q-function using a neural network with parameter θ,

$$Q_\theta(s, a) \approx Q^*(s, a)$$

For each state s_t, the network outputs a Q-value for every possible action.

4.2. Loss Function and Target Network

The network is trained by comparing its predicted Q-value with a target Q-value computed from the next state:

$$y_t = R(s, a) + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a')$$

where θ^- are the parameters of the target network that is updated less frequently for stable training.

The loss function during training is defined by,

$$L(\theta) = E\left[\left(y_t - Q_\theta(s_t, a_t) \right)^2 \right]$$

4.3. Parameter Updates

The main network's parameters are updated using gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\theta)$$

The target network is updated after every N steps,

$$\theta^{-} \leftarrow \theta$$

This keeps the learning process stable and prevents feedback loops from rapidly changing Q-values.

4.4. Exploration Strategy

During training, the agent uses ϵ-greedy exploration strategy,

$$a_t = \begin{cases} \text{random action}, & \text{with probability } \varepsilon_t, \\ \arg\max_a Q(s_t, a; \theta), & \text{otherwise.} \end{cases}$$

The exploration rate decays over time according to

$$\varepsilon_t = max(\varepsilon_{min}, \varepsilon_0 e^{-\lambda t})$$

4.5. Replay Buffer

A replay buffer is used to hold each transition $(s_t, a_t, r_t s_{t+1})$. The network is updated and targets are calculated using random batches drawn from the buffer during training. By utilizing prior knowledge, this procedure improves data efficiency and disrupts temporal correlations between successive samples.

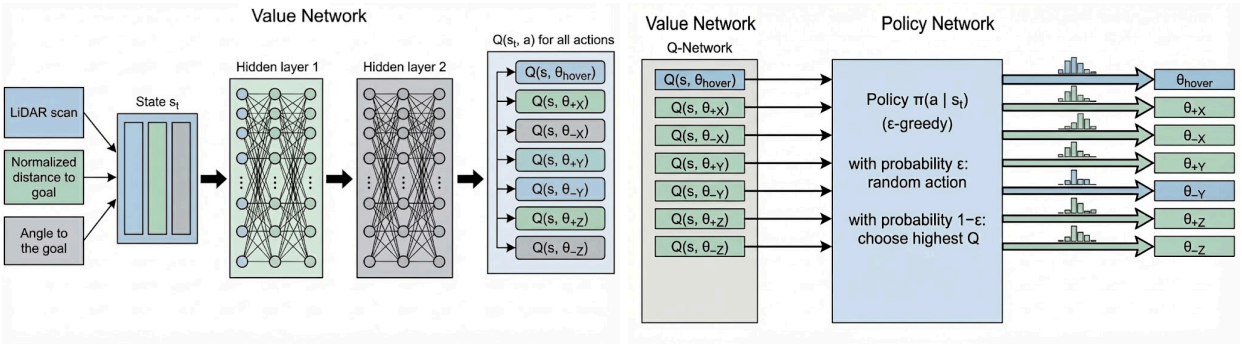


Figure 2: Schematics of value and policy networks for the SAR system

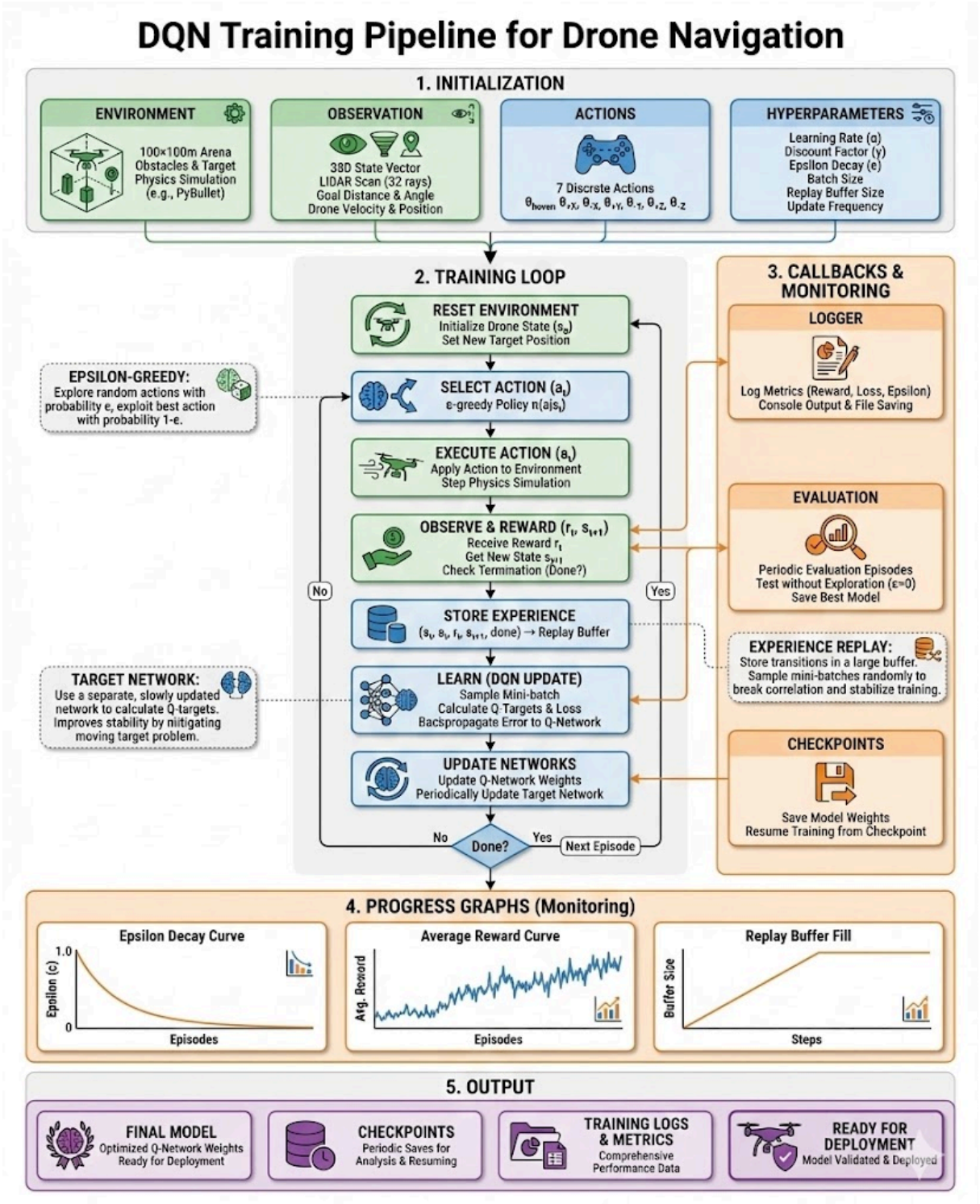


Figure 3: DQN Training Pipeline

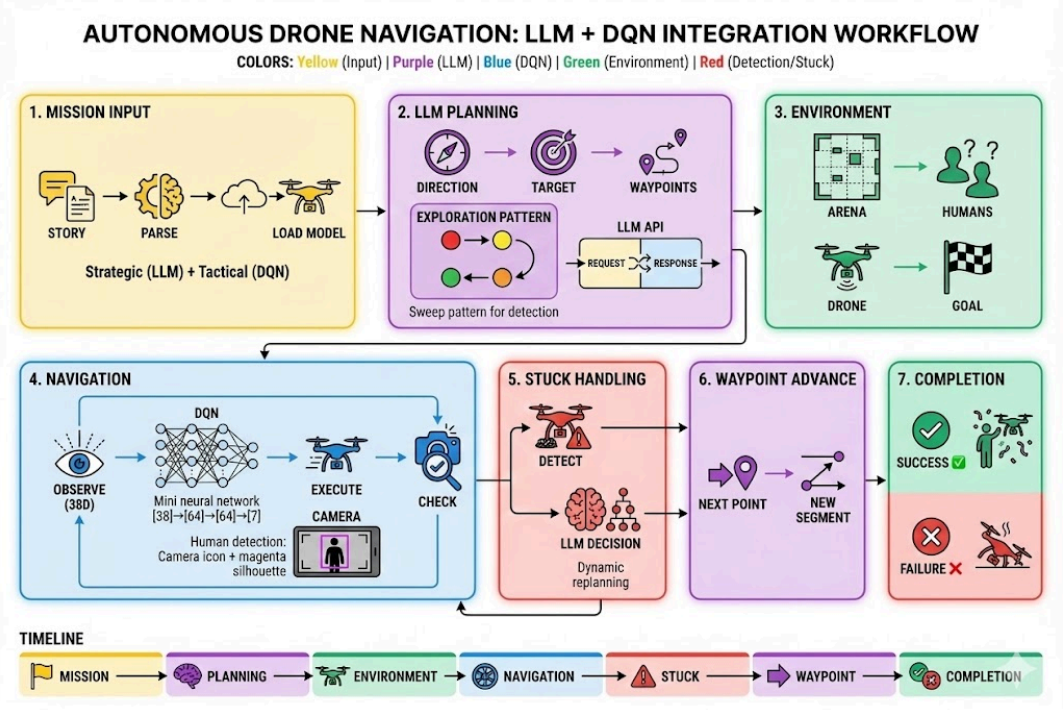


Figure 4: Workflow of RL Agent and LLM to navigate drone

5. LLM Implementation

As a high-level planner, the Large Language Model (LLM) is in charge of creating waypoints, understanding mission objectives, and creating plans in the event that the agent encounters obstacles or becomes trapped. The LLM uses valid JSON instructions to communicate and functions independently of the agent.

5.1. Google Gemma 3 4B

Google Gemma 3 4B was chosen as the preferred model because of its low latency and strong spatial-reasoning performance for waypoint generation. Its long context window helps retain mission details, environmental constraints, and planning history without losing critical information (exact context length depends on the provider and configuration). To reduce hallucinations and improve the reliability of waypoint generation, the model is run at a low temperature with strict JSON-only output formatting.

5.2. Prompt Engineering

The LLM is guided through explicit constraints and strict output formatting. This section details the main prompt categories used in the system.

Segment Generation Prompt

This is the primary prompt used for generating segment-based navigation paths

```
You are an emergency drone navigator. Current mission segment {segment_count + 1}.

Story: "{story}"

Current position: {current_pos}
Direction: {temp_direction}
Direction vector: {DIRECTION_VECTORS[temp_direction]}

To advance {self.spacing}m in direction {temp_direction}, add approximately
{self.spacing / (2**0.5):.4f} to both X and Y coordinates (for north-east).

Generate EXACTLY {num_wp} waypoints for the next {segment_dist:.1f}m segment.

First waypoint must be: [{current_pos[0] + DIRECTION_VECTORS[temp_direction][0] *
self.spacing:.4f}, {current_pos[1] + DIRECTION_VECTORS[temp_direction][1] *
self.spacing:.4f}, {current_pos[2]:.1f}]
Each subsequent waypoint advances another {self.spacing}m in the same direction.

Last waypoint advances {segment_dist:.1f}m total from current position.

Maintain altitude Z = {start_pos[2]}m.
Stay within ±{self.bounds}m boundaries.

Return ONLY valid JSON: {{"waypoints": [[x1,y1,z1], [x2,y2,z2], ...]}}

RULES:
- No markdown ``` markers
- No extra text
- Waypoints must progress exactly in the direction
- Use the exact coordinates provided for the first waypoint
```

Waypoint Generation Prompt

This is the primary prompt used for generating waypoints

```
You are an emergency drone navigator. Full remaining path.

Story: "{self.story}"

Current position: {current_pos}
```

```
Direction: {self.temp_direction}
Direction vector: {vec}

Generate EXACTLY {num_wp} DISTINCT waypoints for the full {segment_dist:.1f}m to
the boundary.

The direction vector {vec} means for each step, add {vec[0]} to x, {vec[1]} to y,
{vec[2]} to z.

First waypoint must be: [{current_pos[0] + vec[0] * self.spacing:.4f},
{current_pos[1] + vec[1] * self.spacing:.4f}, 1.5]

Each subsequent waypoint advances exactly {self.spacing}m in the same direction:
add {vec[0] * self.spacing} to x, {vec[1] * self.spacing} to y, {vec[2] *
self.spacing} to z.

Last waypoint advances {segment_dist:.1f}m total from current position.

Maintain altitude Z = 1.5m.
Stay within ±{self.bounds}m boundaries.

Return ONLY valid JSON: {{"waypoints": [[x1,y1,z1], [x2,y2,z2], ...]}}

RULES:
- No markdown ``` markers
- No extra text
- Waypoints must progress exactly in the direction vector
- Use the exact coordinates provided for the first waypoint
- All waypoints must be unique and distinct
- Do not repeat any waypoint
```

Stuck-Resolution Prompt

This prompt is triggered when the agent makes no progress for *N* steps, indicating that the agent is in a local minimum or its path is obstructed.

```
Drone emergency: STUCK near obstacle.

Current: {current_pos}
Target: {target_pos}
Stuck: {stuck_steps} steps
Obstacles: {len(close_obstacles)} within 2m (LIDAR: {close_obstacles[:3]})

Cannot progress. Decision?

Options:
```

```
- "skip": Skip this waypoint (recommended if obstacle blocks path)
- "abort": End mission
- "continue": Keep trying

Return ONLY JSON: {{"decision": "skip"}}
```

Fallback Strategies

We implement multiple fallback strategies to handle LLM failures:

- Retry Logic: The system attempts up to three times with the same prompt if waypoint generation is unsuccessful or generates invalid JSON.
- Validation Layer: Generated waypoints are immediately validated:
  - Boundary checking: All coordinates must satisfy $x, y \leq 50$ meters
  - Duplicate removal: Consecutive identical waypoints are filtered
- Waypoint Validation Fallback: If all LLM attempts fail, the system can continue toward the last valid waypoint rather than aborting the mission
- Default Direction: If no directional keyword is detected in the mission information, the system defaults to "north" direction

6. Experimental Setup

The experiments evaluate a hybrid UAV navigation system that combines a LiDAR-based DQN policy for local obstacle avoidance with an LLM-driven planner for story-based mission guidance in a cluttered simulated environment. The goal is to test whether this hierarchy can navigate reliably through dense obstacles, follow natural-language missions, and detect human targets.

6.1. Simulation environment and UAV model

All experiments use PyBullet in a 100 m × 100 m arena with procedurally generated trees, rocks, logs, buildings, and small artificial obstacles, plus boundary walls with cardinal labels and a collision-enabled ground plane. The UAV is modeled as a kinematic sphere (radius 0.3 m, mass 1.0 kg) constrained to 1.5 m altitude, equipped with a 36-ray 2 m-range planar LiDAR and a 320×240 forward-facing RGB camera, and controlled via 3.0 m/s velocity commands in the world frame. This abstraction removes attitude control and focuses the learning problem on navigation and obstacle avoidance.

6.2. RL formulation and training

The RL state is a 38-dimensional vector with 36 normalized LiDAR ranges, normalized distance to the current goal, and relative bearing to that goal. The action space has seven discrete velocity primitives (hover, ±X, ±Y, ±Z), repeated for several physics steps, and the reward combines distance progress, obstacle proximity penalties, a collision penalty, and a success bonus when the drone reaches a 0.5 m goal radius. A DQN agent from Stable-Baselines3 with a 2×64 MLP policy is trained for 1 M timesteps (buffer 200k, learning rate 2×10^{-4}, $\gamma = 0.99$, target update

every 500 steps) on the SDSC Expanse GPU-shared partition, with evaluation rollouts every 50k steps and automatic saving of the best and final models.

6.3. LLM-based mission planner

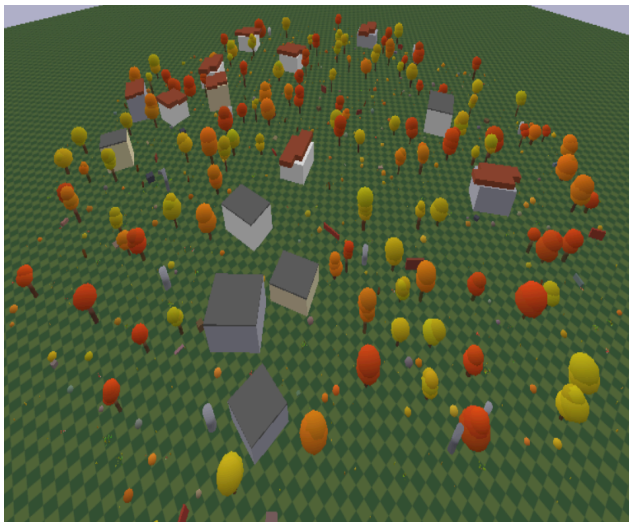
High-level missions are provided as text stories (for example, "search north-east for a missing hiker") and processed by a Navigator module that uses the Gemma 3 4B. The system extracts a dominant direction, computes a boundary target within ±50 m, and asks the LLM for a sequence of waypoints along that direction, under strict JSON and geometric constraints; all waypoints are validated before being passed to the RL controller. Each waypoint is expanded into a small lateral exploration pattern (left–center–right–center, ±3 m), and when the drone is detected as stuck, a compact context (pose, target, LiDAR summary) is sent to the LLM to decide whether to continue, skip the waypoint, or abort.

6.4. Human detection and evaluation protocol

Humans are rendered as magenta cylinders and detected by a simple color-based vision routine that thresholds magenta pixels, checks aspect ratio, and triggers when enough contiguous pixels are present, after which the system logs the position, saves a snapshot, and ends the episode as a successful rescue. For each mission story and direction, the best DQN checkpoint is tested in 20 episodes with randomized start–goal pairs and human placements along the mission direction; the LLM provides the waypoint path once, the DQN handles local navigation, and episodes terminate on human detection, boundary completion, LLM abort, or a step limit, while navigation, detection, LLM usage, and 2D trajectories are logged for later analysis.

6.5. Simulation Environment:

Figures given below illustrate the simulated search-and-rescue environment and UAV setup used in our experiments, showing the drone navigating at low altitude toward a human-shaped target and a top-down view of the procedurally generated forest–village scene with dense trees, buildings, and scattered obstacles that the agent must traverse.



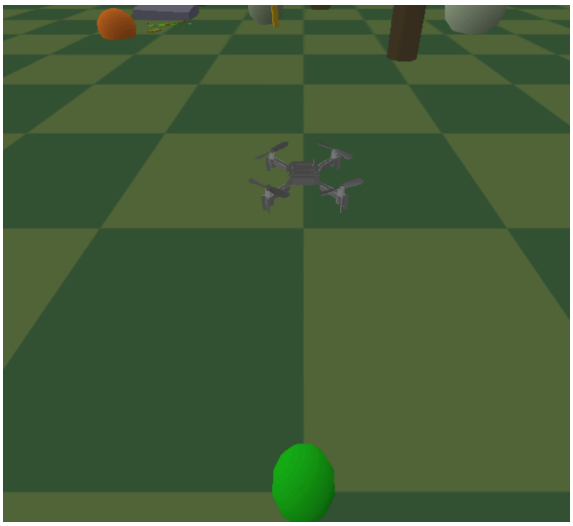


Figure 5: Aerial view of the simulation environment.

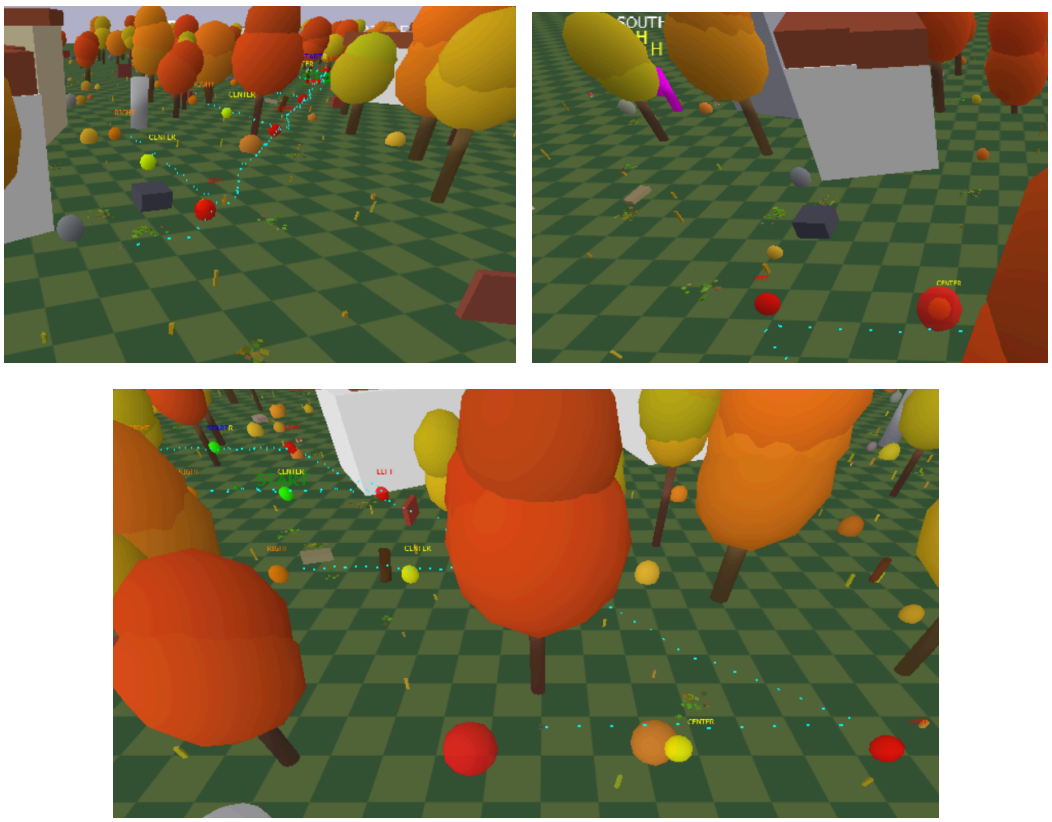


Figure 6: Visualization of the drone's navigation path

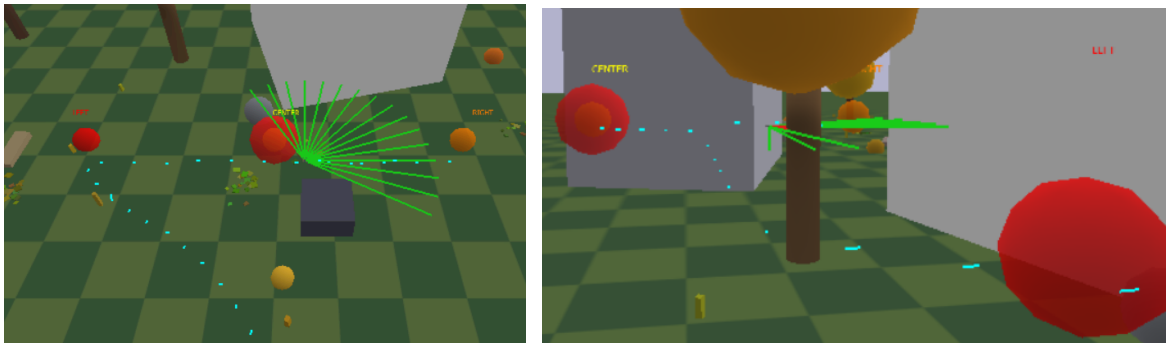


Figure 7: LiDAR scanning to avoid obstacles

```
📍 Target: CENTER → [-30.0, -30.0, 1.5]
→ Next Target: [-30.0, -30.0, 1.5]
Lidar overlay: OFF
  ✓ Human detected through camera at pixels (475, 128) (mask=1284)
  ✓ Saved snapshot: detected_human_snaps\human_detected_ep1_step4075.png
I got human at this xyz location: [-29.55510885938542, -27.806434063346803, 1.4812781019460783]
```

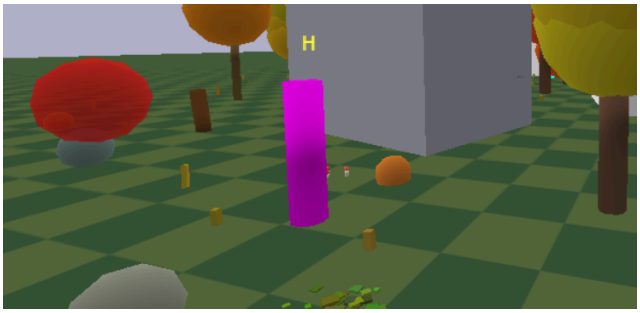


Figure 8: Human detection through the drone's RGB camera

7. Results and Analysis

We trained the LiDAR-based DQN agent with distance and relative angle inputs for 1,000,000 timesteps and periodically evaluated the policy on a fixed validation set of episodes. The eval/mean_reward curve shows that, after an initial drop around 150k steps, the agent steadily improves, reaching a mean evaluation reward of approximately 23–24 by 500–600k steps.

Around 700–800k steps, there are two clear performance drops, followed by recovery and another upward trend that reaches about 22.5–23.0 near 1M steps, indicating that the policy is able to relearn and stabilize after exploration-induced fluctuations.

The behavior of eval/mean_ep_length is consistent with this reward profile. At the beginning of training, the mean episode length is high and very noisy, which is expected because the agent explores inefficient and sometimes looping trajectories. As training progresses towards 400–600k timesteps, the average episode length decreases and becomes more stable, which aligns with the improved mean reward and suggests that the agent is learning to reach the goal in fewer steps instead of wandering. In the later part of training, around the same region where the mean reward dips (700–800k steps), the episode length temporarily increases again, which matches the degradation in policy quality, but it then stabilizes back in the low-to-mid 90s as the reward curve recovers near the end of training.

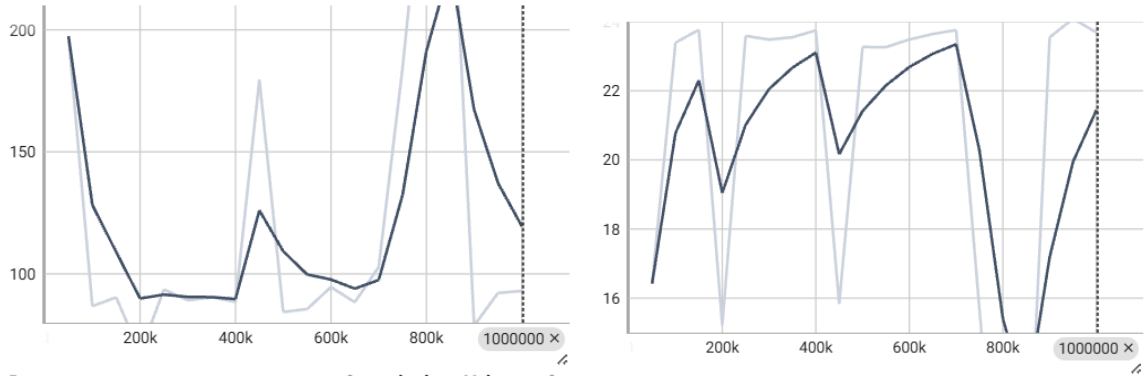


Figure 9: Mean episode length and mean reward across all steps

Overall, these results confirm that the final observation design (LiDAR scan, normalized distance-to-goal, and relative angle) with a DQN-based policy is capable of learning a reasonably stable navigation strategy in the cluttered environment. The mixed pattern of local drops and recoveries in both reward and episode length also supports our earlier observations that using too many parallel environments and overly complex hybrid inputs (RGB + kinematics + depth) made it harder for DQN to converge, whereas the more compact LiDAR-centric state representation led to more consistent performance gains over time.

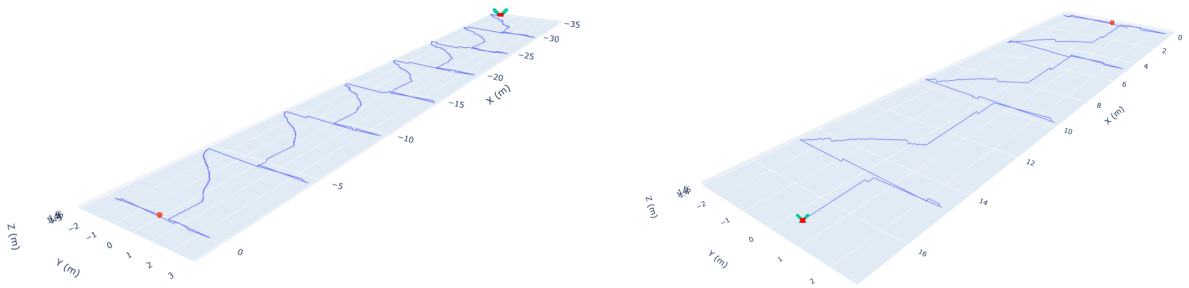


Figure 10: 3D Trajectory plots of the agent's navigation path

Evaluation Results

The SAR metrics show that the system achieves a high overall success rate, suggesting that the LLM is effective in guiding the drone from initial position to the target location. Search efficiency is decent due to detours, obstacle avoidance maneuvers, and occasional deviations by LLM planning. Coverage ratio remains low, as successful missions generally follow direct waypoint corridors rather than wide-area sweeps. Collision rate is moderate, indicating that while the policy handles most obstacle interactions, certain edge cases lead to failures. Time-to-rescue values reflect mission length and environmental complexity, showing that even successful runs may require substantial step counts when navigating around dense obstacles or recovering from misaligned waypoints.

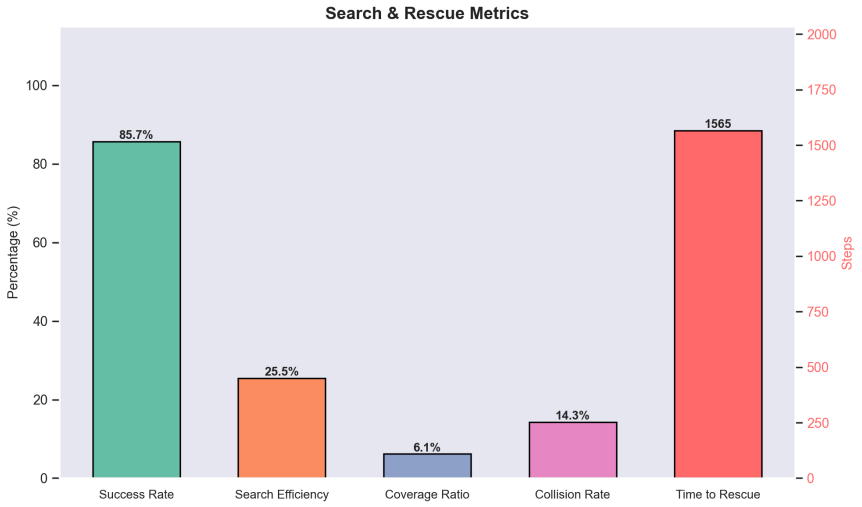


Figure 11: Evaluation results of the search and rescue operation

The latency distribution indicates that the LLM responds in approximately one second, which is sufficient for real-time waypoint generation. Low latency indicates that high-level planning can operate together with a low-level controller in real-time. The hallucination rate shows the number of generated waypoints in which the model gave wrong or semantically incorrect output. This highlights that hallucinations can lead to inefficient trajectories, reinforcing the need for validation layers and fallback behaviors in LLM-guided navigation.

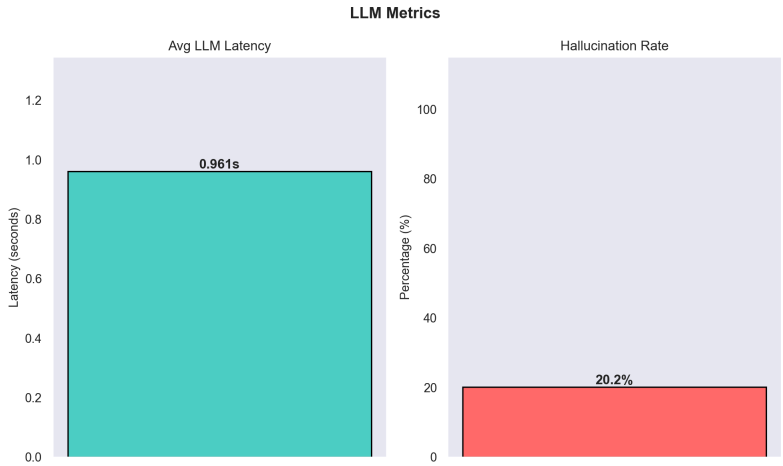


Figure 12: Evaluation results of the LLM metrics

Challenges

We started by working with the gym-pybullet-drones environment, where we initially learned how to navigate the drone with a reinforcement learning (RL) agent from point A to B in an empty environment. We experimented with both PPO and DQN, but since DQN was preferable in our setting, we decided to stick with it. Although DQN can give good results when tuned

properly, a major issue is that standard DQN in Stable-Baselines3 only supports discrete action spaces, while in reality we have a continuous-control drone environment. To address this mismatch, we started with a basic discrete action space consisting of seven actions: hover (0), ±x, ±y, and ±z velocity commands.

At first, we tried to implement DQN from scratch, but this did not yield stable results; there were large fluctuations in the mean reward curve. We then switched to using the DQN implementation from Stable-Baselines3, which is considered an industry-standard library and significantly reduces boilerplate code, since we only need to configure and call the provided DQN class. In this experiment, we used an MLP policy with DQN and provided only a low-dimensional distance vector as input, allowing the agent to learn Q-values directly from this kinematic information. This setup performed reasonably well and started to work properly with only about 50k timesteps of training. However, this scenario was still unrealistic because the environment was empty, while in the real world we have obstacles at different locations.

To move towards realism, we started working with a combination of vision and distance metrics for model training. In gym-pybullet-drones, by default, observations can be either kinematics or RGB, but only one modality can be selected at a time. In our case, we also needed the drone to be capable of flying from a start position to an uncertain target position, so distance measurements had to be considered explicitly. To achieve this, we tried using the MultiInputPolicy of DQN in Stable-Baselines3, where we defined the observation as a dictionary of the form {“rgb”: ..., “state”: ...}, combining images with state vectors. However, this configuration did not work as expected, because the model was not able to learn a meaningful reward-to-state mapping and failed to interpret the hybrid observation structure properly in our setup.

Following official guidance and examples, we realized that it is also possible to work with RGB-only observations if integrating hybrid inputs is problematic. Therefore, we switched back to pure RGB and attempted to inject distance information in a different way. By default, the environment provides four channels (RGBA). We extended this to eleven channels by adding seven extra channels encoding distance-like vectors, aiming to pass this augmented image tensor to the CNN policy so that it could learn directional cues. However, this approach still did not work as intended. We then removed this modification and reverted to using only the original four RGBA channels, while trying to encode distance information purely through the reward function. In this configuration, when the drone reached the target location and received a large success bonus, the agent effectively learned that “this specific building” or region was heavily rewarded, because the distance information was only available inside the reward function and not exposed in the observable state. As a result, instead of learning generalizable navigation, the agent exploited this bias and kept accumulating largely uninformative transitions in the replay buffer.
In another approach, we extended the discrete action space to include more than seven actions, adding diagonal motion commands as well. However, this larger action space made the DQN policy harder to train: the agent became more confused, training took longer, and the performance still did not improve. We tried several other RGB-based approaches, including parameter tuning and hyperparameter adjustments, but the results remained unsatisfactory. At this point, we moved to a hybrid observation design again, using both vision and kinematics simultaneously via a MultiInputPolicy DQN configuration. In this design, the agent still

struggled to learn stable path-following behavior, which we believe was partly due to varying colors and textures in the RGB images, making it difficult to extract robust geometric cues for navigation.

Given these repeated failures, we decided to replace RGB with a more structured visual representation. Motivated by prior work showing that RGB images lack explicit range information and that depth provides more direct geometric structure for obstacle avoidance, we investigated depth images as an alternative. We then fed depth images together with a distance vector into a MultiInputPolicy DQN (Stable-Baselines3), again using a dictionary observation that combines a depth image branch and a low-dimensional state branch. This configuration started to show promising behavior after around 400k timesteps, so we slightly adjusted some hyperparameters and attempted to perform transfer learning on top of these partially trained models but it started to fail again even after training it to 1M+.

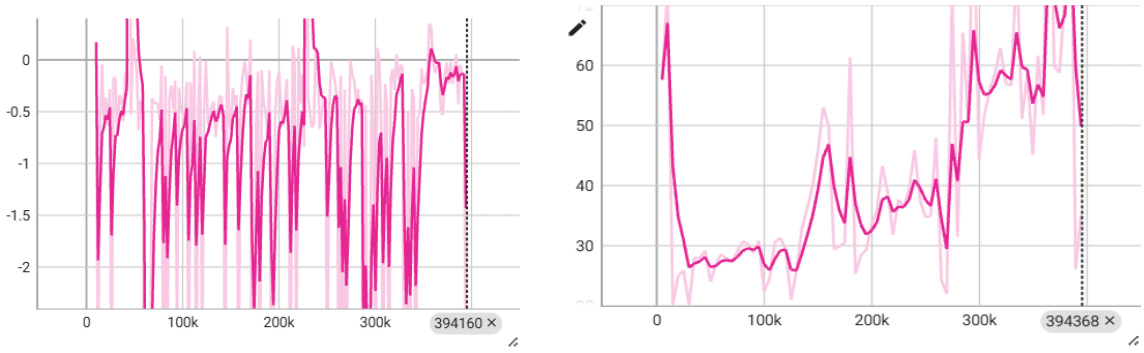


Figure 11: Mean Step reward and Mean episode length

At this point, we were left with only one practical option: using a LiDAR sensor, and the next question was what additional information should be provided along with LiDAR measurements for training the RL agent. Initially, we again tried to use a MultiInputPolicy with RGB images, LiDAR information, and a distance vector to the goal, but this configuration did not produce good results. We then tried a final approach that used only LiDAR information, a normalized distance-to-goal vector, and the relative angle to the goal, and this setup gave us good performance.

During these different experiments, we also tried using multiple parallel environments during training to speed up the process, but increasing the number of environments did not consistently improve the results. We ran several trials with 16 parallel environments and gradient steps of 4–8 to absorb the larger data throughput, but the RL agent struggled to extract good information from such settings. In our experiments, the training runs that used only 4 environments (the default setting) produced relatively better and more stable results compared to configurations with a higher number of parallel environments.

Once we had a working navigation solution with LiDAR, normalized distance, and relative angle, we started to explore how to integrate a large language model (LLM) into our system. We initially considered an open model such as Qwen 2.5 4B from Hugging Face, but it required downloading large tokenized model files, which seemed heavier in practice than using a hosted API. For this reason, we chose Google Gemma 3 4B (instruction-tuned, google/gemma-3-4b-it)

as the lightweight model for the LLM component of our framework from **OpenRouter**. As a ~4B-parameter class model, it provides a good balance between latency and capability for our waypoint-generation prompts, while remaining substantially cheaper and faster than larger models. We access it through OpenRouter, which simplifies provider integration and lets us swap models via configuration if availability or performance changes. Exact per-token pricing varies by provider and account settings on OpenRouter, so we treat cost as a configurable deployment concern rather than hard-coding a single fixed price in the design.

In the LLM component, we initially used the free tier, so our main objective was to minimize the number of API calls. The first use of the LLM was waypoint generation towards the goal. Given that our environment covers a 100×100 m area, we first considered calling the API segment-wise (e.g., requesting waypoints for each 25 m segment). However, due to the free-tier limitations and the relatively small environment, we instead requested a direct sequence of waypoints along the rough hint direction, while constraining the solution to stay within the 50×50 m bounds. After receiving the JSON response, we applied our own validation checks before passing the waypoints to the RL agent.

Next, we had to address how to explore the surroundings of the straight-line path, because the goal could lie near but not exactly on the direct line. Initially, we planned to call the LLM for 3 m surrounding waypoints around each main waypoint, but we realized this would lead to too many API calls and higher cost, especially under the free tier. Instead, we implemented this local exploration directly in code, generating 3 m lateral search offsets around each waypoint, which allowed us to avoid extra LLM calls while still searching the neighborhood of the straight path efficiently. As a result, after reaching each waypoint, the agent systematically scans 3 m on both sides to reduce the risk of missing the goal while moving toward the boundary.

Finally, we encountered another issue: some waypoints could lie inside obstacles or in regions that are effectively unreachable. To handle such cases, we implemented a helper mechanism using the LLM. When the drone had difficulty reaching a waypoint, we sent the LiDAR readings, the current drone position, and the waypoint coordinates to the LLM, and asked it whether the agent should continue attempting to reach the waypoint, abort the episode, or skip that waypoint (especially in cases where the drone is already close). This decision-support step allowed us to recover from problematic waypoints, and in this way we were able to implement the overall system successfully so that it can identify and reach the goal reliably in cluttered environments.

Contribution:

Qasim: Qasim worked on different experiments, mentioned in the challenges section and got the RL and LLM part working out of it. Initially, in our first deadline he collaborated with Ateesh for the RL part and implemented the LLM but after feedback in class he again wrote code for RL training and transitioned our approach to small model Gemma 3 4B as required. In the report he worked on the challenges section and after our first deadline he also updated the report as per our new approach in our final code.

Farhan: Farhan worked mainly on writing the report. He also tried some methods in the last week of 1st deadline as he was unavailable before that due to his Capstone project. He tried to get the RL and LLM part working but we didn't get the results in that approach.

Rizwan: Rizwan mainly worked on making the presentation slides.

References:

J. Panerati, H. Zheng, E. Kaufmann, A. Granacher, P. Foehn, D. Scaramuzza, and A. P. Schoellig, "Learning to Fly: A Gym Environment with PyBullet Physics for Reinforcement Learning of Quadcopter Control," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2021.

A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, R. Traoré, and H. Erm, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," J. Mach. Learn. Res., Open Source Software Track, 2021.

Stable-Baselines3 Contributors, "DQN — Deep Q-Networks," Stable-Baselines3 Documentation, 2025. [Online]. Available: https://stable-baselines3.readthedocs.io

Stable-Baselines3 Contributors, "Policy Networks (Custom Policy and MultiInputPolicy)," Stable-Baselines3 Documentation, 2025. [Online]. Available: https://stable-baselines3.readthedocs.io

Stable-Baselines3 Contributors, "Using Custom Environments," Stable-Baselines3 Documentation, 2025. [Online]. Available: https://stable-baselines3.readthedocs.io

Stable-Baselines3 Contributors, "Vectorized Environments," Stable-Baselines3 Documentation, 2025. [Online]. Available: https://stable-baselines3.readthedocs.io

M. Rybicka et al., "Comparative Analysis of DQN and PPO Algorithms in UAV Obstacle Avoidance," in Proc. CEUR Workshop, 2024.

Y. Zeng et al., "Deep Reinforcement Learning-Based End-to-End Control for UAV Autonomous Navigation," Drones, vol. 6, no. 11, 2022.

K. Grolinger et al., "Autonomous Unmanned Aerial Vehicle Navigation Using Deep Reinforcement Learning," M.E.Sc. Thesis and related publications, Western University, 2022.

J. Thomas, O. Olshanskyi, et al., "Interpretable UAV Collision Avoidance Using Deep Reinforcement Learning," 2021. [Online]. Available: arXiv:2105.12254.

Y. Li et al., "E-DQN-Based Path Planning Method for Drones in AirSim Simulator," Sensors, vol. 24, no. 8, 2024.

Z. Ji et al., "VO-Safe Reinforcement Learning for Drone Navigation," Cardiff University, 2024. [Online]. Available: https://orca.cardiff.ac.uk

Y. Lin et al., "Reinforcement Learning-Based Autonomous Robot Navigation and Tracking," 2024.

A. P. Lacerda et al., “Dynamic Reward in DQN for Autonomous Navigation of UAVs Using Depth Camera,” in Proc. Int. Conf. Control, Decision and Information Technologies (CoDIT), 2023.
Z. Yang et al., “Autonomous Quadrotor Obstacle Avoidance Based on Dueling Deep Q-Network,” Neurocomputing, vol. 458, pp. 489–499, 2021.
Y. Chen et al., “Deep Reinforcement Learning for Collision Avoidance in Unmanned Aerial Vehicles,” 2025.
H. Lee and C. Lo, “A UAV Indoor Obstacle Avoidance System Based on Deep Reinforcement Learning Using LiDAR,” 2022.
A. da Silva et al., “Autonomous UAV Navigation Using Reinforcement Learning,” Int. J. Mach. Learn., vol. 9, 2019.
Z. Ji et al., “A New Hybrid Reinforcement Learning with Artificial Potential Field for UAV Path Planning,” 2025.
T. Lin et al., “NavRL: Learning Safe Flight in Dynamic Environments with Language Guidance,” arXiv preprint, 2024.
Google DeepMind and Google Cloud, “Gemini 2.0: A New Generation of Multimodal Models,” Technical and Pricing Documentation, 2025. [Online]. Available: https://cloud.google.com