Polytech Nice Sophia

Master 2 – Robotics and Autonomous Systems

Autonomous AI-Driven Surveillance Drone

Project Report

Authors:
Youssef Miri
Nicolas Consalvi
Adrien Waeles-Devaux

Supervisors:
Frédéric Juan
Frédéric Rallo
Sébastien Rothhut
Matthias Vermot-Desroches

January 25, 2026

Abstract

This report details the development of an autonomous drone platform that integrates robust systems algorithms with advanced Artificial Intelligence. We established a high-fidelity simulation environment using ROS 2 and Gazebo to validate a custom hardware architecture combining Pixhawk and Nvidia Jetson control. To ensure precise navigation, we implemented a 6-DOF Extended Kalman Filter (EKF) capable of handling non-linear flight dynamics through asynchronous sensor fusion. Building on this foundation, we deployed high-level autonomy features: LLM-based voice control, real-time visual human tracking, and Deep Reinforcement Learning for precision landing on moving targets.

Contents

1. System Design, Assembly, and Simulation

Author: Nicolas Consalvi

1.1 Introduction

As part of the Autonomous Surveillance Drone project, my primary responsibility was to ensure the viability of the platform, both physically (assembly and real flight) and virtually (development of a high-fidelity simulation environment). This dual approach is critical: it allows for the validation of SLAM and navigation algorithms in a safe environment (SITL - Software In The Loop) before risking expensive hardware during real flights. This chapter details the system engineering implemented, from hardware assembly to 3D mapping algorithms under ROS 2.

1.2 Hardware Architecture and Assembly

The drone's architecture is based on a trade-off between payload capacity (to carry the AI computer) and maneuverability. We opted for a Quadcopter "X" configuration.

1.2.1 Component Selection

The core of the system is split into two levels, separating low-level tasks (stabilization) from high-level tasks (AI/SLAM):

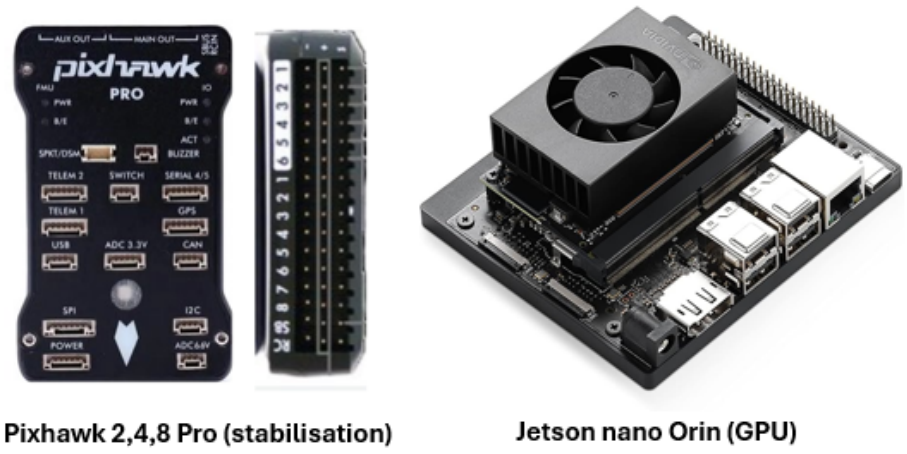


Figure 1.1: Hardware Overview: Pixhawk Pro (Flight Control) and Jetson Nano Orin (Compute).

- **Flight Controller (Pixhawk 2.4.8 Pro):** This microcontroller handles the real-time control loop (500Hz+). It is imperative for safety, as the Jetson (companion computer) is not a hard real-time system.
- **Companion Computer (NVIDIA Jetson Nano Orin):** Connected via serial (MAVLink) to the flight controller, it processes heavy data streams (Depth camera, Lidar, SLAM).
- **Propulsion:** 1300KV motors coupled with a 4-in-1 ESC (Electronic Speed Controller), providing significant space and weight savings compared to individual ESCs.

1.2.2 Assembly Logic and Rotation Direction

The critical step in assembly concerns the orientation of the motors. To ensure yaw stability and torque balance, motors must rotate in opposite directions diagonally.

I configured the drone according to the "Quad X" standard where propellers spin inwards towards the chassis to maximize aerodynamic authority during fast movements.

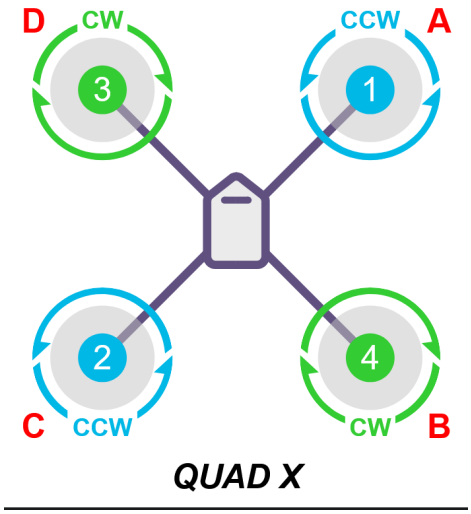


Figure 1.2: Motor configuration and rotation direction scheme (Top View). The verified command order is ABCD: Top-Right, Bottom-Right, Bottom-Left, Top-Left.

The propulsion system is driven by a 4-in-1 ESC (Electronic Speed Controller) powered by a 3S to 6S LiPo battery. This unit routes the PWM signals (channels 1-4) to the Pixhawk's Main Ports.
Hardware Safety Warning: A critical implementation detail concerns the power distribution. Since the ESC's voltage output is unregulated (V_BAT), we integrated an external **5V Buck converter** to power the Pixhawk's motor rail. Connecting the unregulated ESC output directly would have exposed the flight controller to full battery voltage, resulting in immediate hardware failure.
During motor testing, I sequentially validated the command order (A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D) on main ports 1-2-3-4 of the Pixhawk to match the PX4 software mapping. An error here would result in the drone immediately flipping upon takeoff.

1.3 Software Configuration and Tuning (QGroundControl)

Once the hardware was assembled, configuration via QGroundControl (QGC) was necessary to bridge raw sensor data with flight commands. QGroundControl is an open-source ground control station used to configure, monitor, and fly drones powered by MAVLink-supported flight stacks like PX4 or ArduPilot. It provides a comprehensive interface for flight instrument display, real-time telemetry, and advanced autonomous mission planning across multiple platforms.

1.3.1 Sensor and Radio Calibration

The first step involves calibrating the IMU (Inertial Measurement Unit). This is a delicate procedure where the drone must be placed on its 6 faces while remaining perfectly still. I also performed the magnetometer (compass) calibration, ensuring no electromagnetic interference was present (away from speakers or screens).

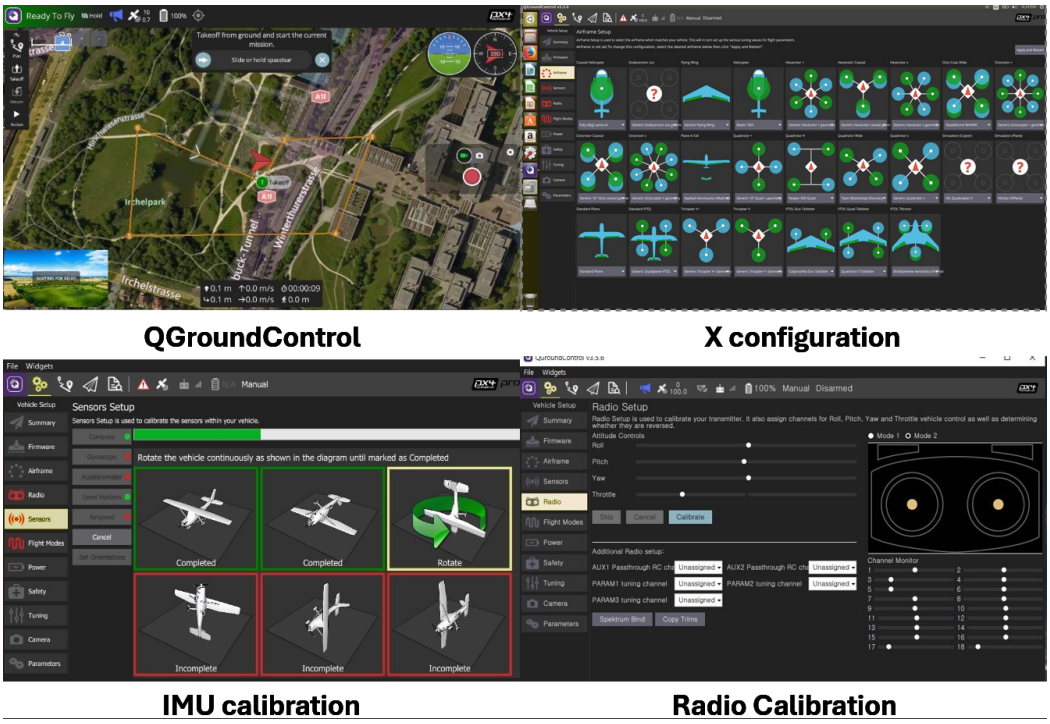


Figure 1.3: Calibration interface in QGroundControl and X-frame selection.

The radio control assignment was configured in **Mode 2** (Throttle on the left), which is the most common standard. I also mapped safety switches (Arm Switch on RC10) and flight modes (Stabilized, Position, Offboard) to the radio.

1.3.2 PID Tuning

The most technical part of flight preparation was tuning the PID (Proportional, Integral, Derivative) control loops. Initially, the drone exhibited slight oscillations. I had to adjust the gains:

- **P Gain (Proportional):** Increased progressively to improve reactivity until rapid oscillations appeared, then reduced by 10%.
- **D Gain (Derivative):** Adjusted to dampen sudden movements and prevent setpoint overshoot. The default parameters were way too high, so we lowered them.
- **I Gain (Integral):** Used to correct static errors, we lowered them. (e.g., drifting against a constant wind).

Despite the challenging testing environment involving high winds and the added weight of power and safety tether cables, the drone successfully maintained stable flight, however, the PID parameters still require further optimization to achieve peak performance.

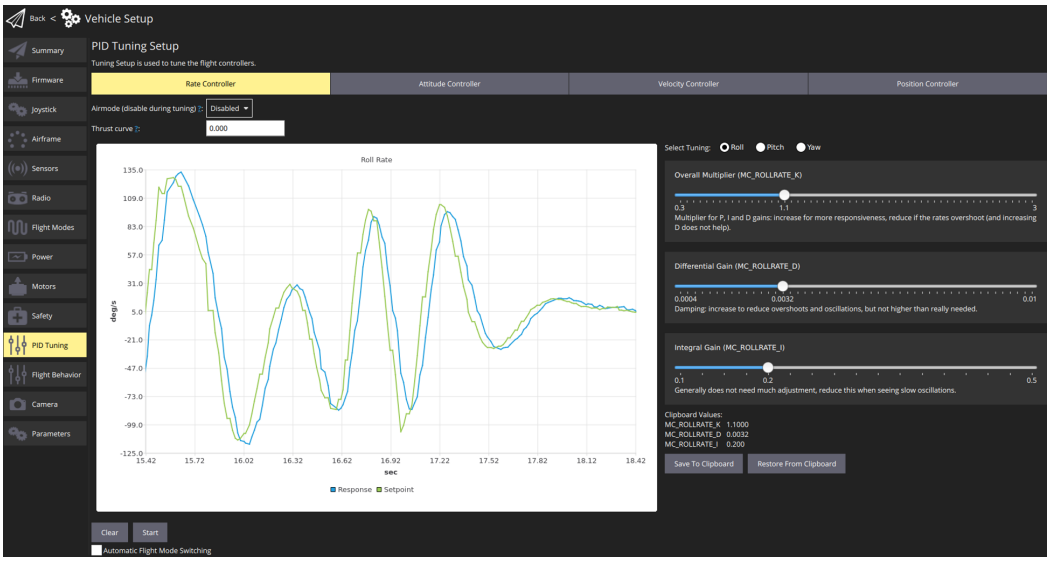


Figure 1.4: PID Tuning Interface: Adjusting Roll/Pitch/Yaw response curves.

1.4 Simulation Environment Development (SITL)

To test the SLAM and autonomous navigation algorithms developed by the team, moving directly to real flight was too risky. Therefore, I established a complete simulation chain.

1.4.1 Technology Stack and Complexity

The environment relies on the interaction between three major software blocks:

1. **PX4 Autopilot (SITL):** The same firmware as the real drone, but running on the host computer.
2. **Gazebo (Harmonic/Garden Version):** The physics simulator generating the world, gravity, and collisions.
3. **ROS 2 Humble:** The robotics middleware for high-level data communication.

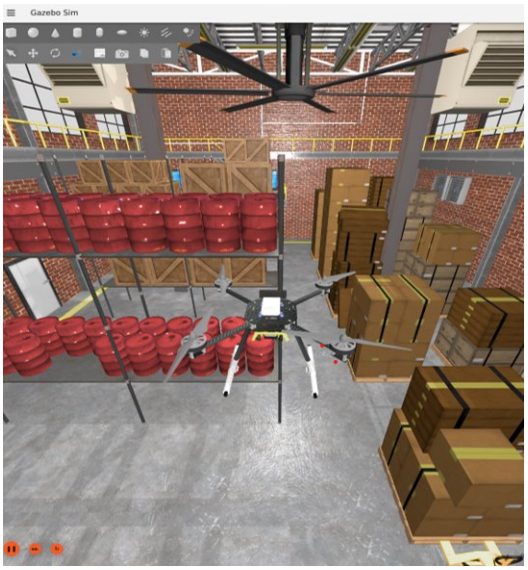


Figure 1.5: Simulation Environment: Integration of ROS 2 Humble, and PX4 on Gazebo Harmonic with a gz x500 depth drone model.

1.4.2 Documentation Issues and ROS 2 Bridge

A major challenge in this project was the rapid obsolescence of online documentation. Most PX4 tutorials refer to ROS 1 (Noetic) and Gazebo Classic. However, we are using ROS 2 Humble.
I had to manually configure the communication bridge. Unlike ROS 1 which used MAVROS, modern architecture uses a **Micro-XRCE-DDS** bridge.

- The Micro-XRCE-DDS client runs on the PX4 firmware.
- The agent runs on the companion computer (or the simulation).

I wrote launch scripts to automate the connection of the agent to the simulator, allowing ROS 2 topics (such as ‘/fmu/in/trajectory/setpoint‘) to be visible and usable by our control nodes.

1.4.3 Firmware Divergence within the Team

It is important to note a particularity in our organization. While I configured the simulation under **PX4** to benefit from its advanced integration with ROS 2 for control, my colleague Youssef developed his AI tests under the **ArduPilot** firmware. ArduPilot offered superior simplicity for his Python scripts, whereas PX4 was necessary for the SLAM and Adrien's EKF.

1.5 3D SLAM Implementation and Transformations (TF)

The final goal of the simulation was to validate the SLAM (Simultaneous Localization and Mapping). For this, I used the `rtabmap_ros` package under ROS 2.

1.5.1 Transform Management (TF Tree)

For SLAM to function, the robot must know the position of its sensors relative to its center. In ROS 2, this is managed via the Transform Frames (TF). TF is a specialized display module used to visualize the coordinate frames of a robot, showing the spatial relationships and orientations between different parts in 3D space. It allows users to track how these frames move and interact in real-time, which is essential for debugging kinematic chains and sensor alignments. The raw data from the simulator did not provide these links correctly.
I developed a broadcaster node publishing static TFs:

- **base_link:** The geometric center of the drone.
- **camera_link:** The position of the RGB-D camera, offset by a few centimeters forward and up relative to ‘base_link‘.

Without this step, point clouds generated by the camera appeared at incoherent locations in the 3D visualization space (Rviz).

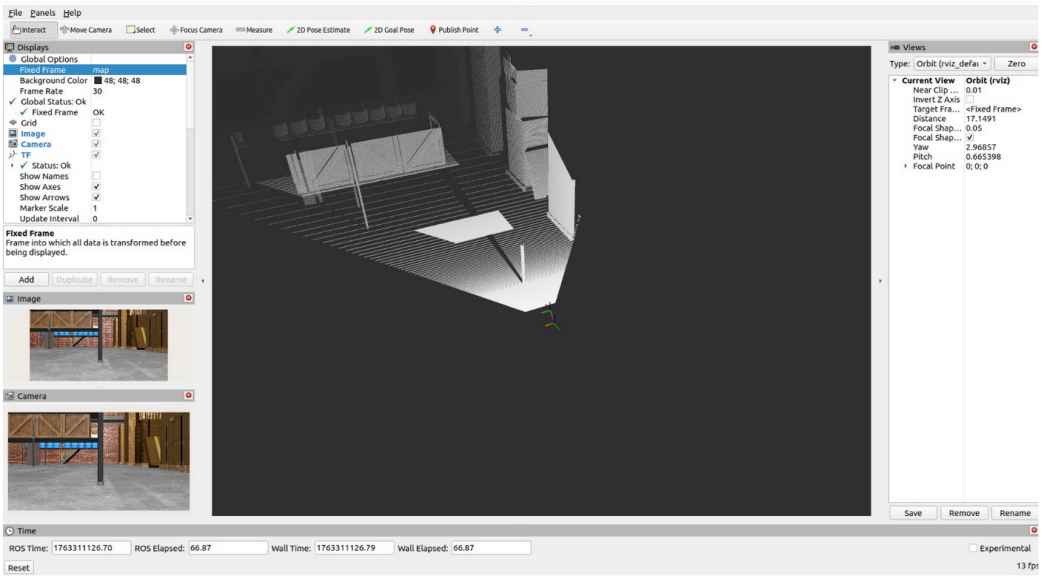


Figure 1.6: Image From RVIZ2 where we can see Transformations Frames (TFs) of the drone.

1.5.2 RTAB-Map Integration

I configured a specific *launch file* to run RTAB-Map in simulation mode. This file remaps the visual odometry topics coming from the PX4 simulation to the inputs expected by the mapping algorithm.

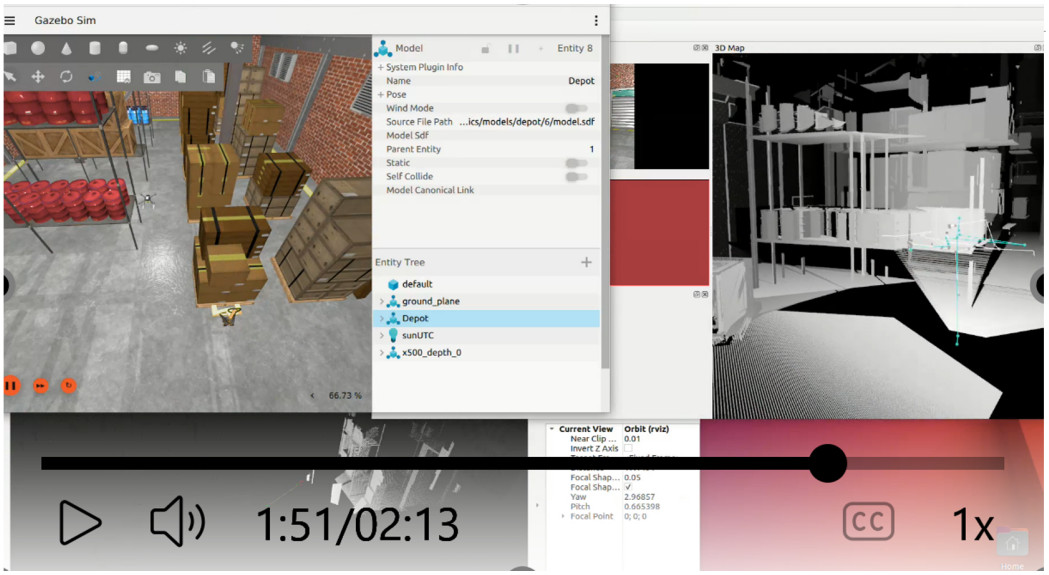


Figure 1.7: 3D SLAM Result in Simulation: At Left, the Gazebo environment. At the bottom left, the camera view with RVIZ2. At Right, the generated 3D map (Octomap) in real-time.

As shown in Figure 1.7, we successfully generated a coherent 3D map of the Gazebo "Warehouse" environment, thus validating the complete acquisition and processing chain prior to any real flight.

2. State Estimation via Extended Kalman Filter

Author: Adrien Waeles-Devaux

2.1 Introduction and Objectives

In aerial robotics, the control loop's performance is limited by the quality of the state estimation. The quadrotor platform requires a precise, high-frequency estimate of its full state vector, specifically attitude (Roll, Pitch, Yaw) and angular rates (p, q, r) to maintain stable flight.
While low-cost sensors like IMUs (Inertial Measurement Units) provide high-rate data, they suffer from significant noise and drift. Conversely, magnetometers provide absolute heading but are susceptible to magnetic disturbances and have slower update rates. To address these limitations, I implemented a 6-DOF Extended Kalman Filter (EKF).
The choice of an EKF over a standard Kalman Filter or a Complementary Filter is driven by the physics of the drone. The rotational dynamics of a rigid body, given by Euler's equations, are non-linear due to gyroscopic coupling effects (cross-products of inertia). A linear filter cannot capture these dynamics during aggressive maneuvers. The implemented EKF solves this by linearly approximating the system dynamics around the current operating point at each time step.

2.2 Mathematical Model

2.2.1 State Vector Definition

I define the state vector $\mathbf{x} \in \mathbb{R}^6$ as:

$$\mathbf{x} = [\phi, \theta, \psi, p, q, r]^T \tag{2.1}$$

Where ϕ, θ, ψ represent Euler angles (Roll, Pitch, Yaw) in radians, and p, q, r represent the body-frame angular velocities in rad/s.

2.2.2 Process Model and Prediction

The prediction step projects the state forward in time using the non-linear transition function $f(x, u)$. The model I adopted uses the control inputs torques generated by the motors.
The continuous-time dynamics are derived from Newton-Euler equations for a rigid body and were taken from the paper [1]:

$$\dot{p} = \frac{1}{J_x}\left[(J_y - J_z)qr + L \cdot U_2\right] \tag{2.2}$$

$$\dot{q} = \frac{1}{J_y}\left[(J_z - J_x)pr + L \cdot U_3\right] \tag{2.3}$$

$$\dot{r} = \frac{1}{J_z}\left[(J_x - J_y)pq + f \cdot U_4\right] \tag{2.4}$$

Here, J_x, J_y, J_z are the moments of inertia, L is the arm length, and U_2, U_3, U_4 are the control torques.
To propagate the error covariance matrix P, I then compute the Jacobian matrix $\mathbf{F} = \frac{\partial f}{\partial \mathbf{x}}$ at each step. This matrix captures how uncertainties in one state propagate to others.

2.3 Software Implementation

I implemented the system using ROS 2 nodes written in C++ using the Eigen library for efficient linear algebra operations.

2.3.1 Control Inputs Calculation

A crucial component of the prediction step is accurate knowledge of the forces applied to the drone. I developed a specific node, `ControlInputsNode`, which subscribes to the motor speeds (ω_i) published by the simulator. It computes the virtual control inputs u using the thrust coefficient (k_f) and drag coefficient (k_m). Example for U_2 computation:

$$U_2 = k_f(-\omega_0^2 + \omega_1^2 + \omega_2^2 - \omega_3^2) \quad \text{(Roll Torque)} \tag{2.5}$$

This allows the EKF to anticipate movement before the sensors even detect it, reducing phase lag.

2.3.2 Asynchronous Sensor Fusion Architecture

The fusion architecture is designed to be event-driven, handling sensors with different refresh rates without blocking.

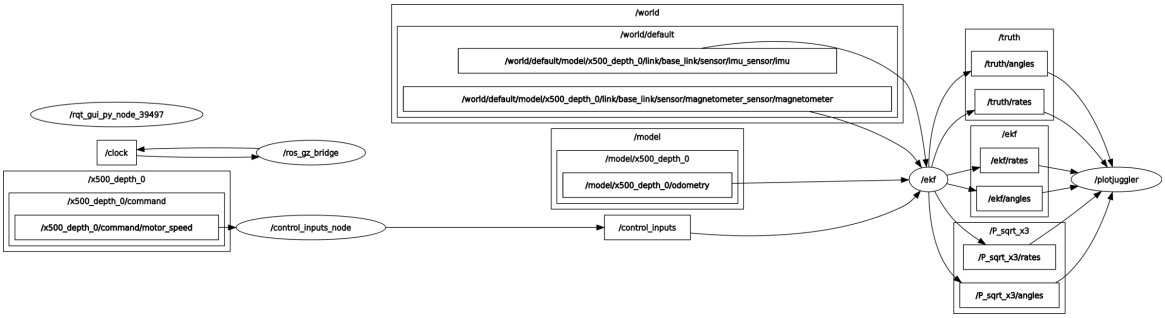


Figure 2.1: ROS 2 Node Graph from RqtGraph.

The `/ekf` node centralizes data from `/control_inputs`, IMU, and Magnetometer.
The `/ekf` node structure is as follows:

- **IMU Callback (100 Hz):** This sets the tempo of the filter.
  1. **Predict:** State propagation using the latest control inputs u and time delta dt.
  2. **Update (Gyro):** Correction of angular rates p, q, r.
  3. **Update (Accelerometer):** Correction of Roll and Pitch angles using the gravity vector.
- **Magnetometer Callback (10 Hz):**
  1. **Update:** Asynchronous correction of the heading (Yaw). I implemented a Tilt-Compensated calculation to project the magnetic field onto the horizontal plane using the estimated Roll/Pitch, ensuring accurate heading even when the drone is inclined.

I utilized SensorDataQoS (Quality of Service) policies for all sensor subscriptions to ensure low-latency, prioritizing up-to-date data over reliability (dropping old packets if necessary).

2.3.3 Robustness: Accelerometer Gating

A simple implementation of attitude estimation could assume the accelerometer direction always points down (gravity). However, during flight, the sensor also measures other specific force: $\mathbf{a}_{meas} = \mathbf{g} + \mathbf{a}_{motion}$. To prevent the filter from confusing drone acceleration with a change in orientation, I implemented a Gating mechanism.

- **Detection:** The system monitors the norm $||\mathbf{a}_{meas}||$. If it deviates from g by more than 20% (not included in $[0.8g, 1.2g]$), the measurement is rejected.
- **Consequence:** The update step is skipped. In addition, I increase the covariance P to reflect this increased uncertainty. The filter temporarily relies on Dead Reckoning until the drone stabilizes.

2.4 Experimental Results and Analysis

2.4.1 Tuning Methodology

Tuning the noise covariance matrices **Q** (Process Noise) and **R** (Measurement Noise) is critical.

- **Measurement Noise (R):** I extracted the sensor noise characteristics from the Gazebo SDF sensor definition files.
- **Process Noise (Q)** This represents the trust in the physical model.
  I utilized the ROS 2 `OnSetParameters` callback to tune these values from the terminal in real-time during flight. By adjusting **Q** of each state, I could instantly observe the trade-off between tracking responsiveness and jitter rejection.

2.4.2 Validation via Statistical Consistency

I validated the filter performance by comparing the estimated state against the Ground Truth from the simulator using PlotJuggler, a real-time visualization tool for ROS data streams that can be executed as a standalone node [5].

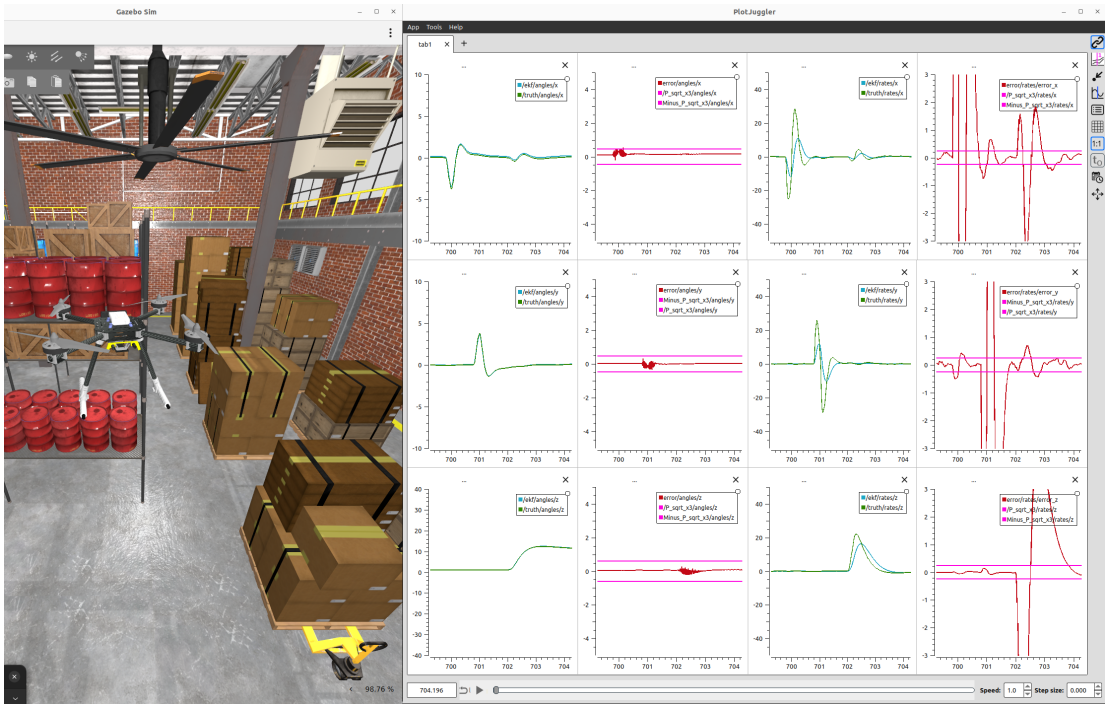


Figure 2.2: Experimental results visualized in PlotJuggler. Left: Gazebo simulation. Right: Real-time plots of estimation error (Red) vs 3σ confidence bounds (Pink).

As depicted on the figure 2.2, i used PlotJuggler to create a structured interface composed of four columns and three rows, each row corresponding to one of the three rotation axes (roll, pitch, yaw). The columns provide different levels of analysis:

- **Column 1**: comparison between the estimated angles and the ground truth;
- **Column 2**: angle estimation error together with the $\pm 3\sqrt{P}$ confidence bounds;
- **Column 3**: comparison between the estimated angular rates and their ground truth;
- **Column 4**: angular-rate estimation error together with the $\pm 3\sqrt{P}$ confidence bounds.

The terms $3\sqrt{P(i,i)}$ depicted by the pink envelope correspond to the three-sigma confidence bounds obtained from the EKF covariance matrix P. Each diagonal element P_{ii} represents the variance of the state x_i, whose square root defines the standard deviation. Scaling this quantity by a factor of three gives an interval that, due to the Gaussian noise assumption, should contain approximately 99.7% of the estimation uncertainty.
By understanding this, the plots in Figure 2.2 demonstrate two key behaviors:

1. **Convergence:** The estimation error (Red) remains centered around zero, indicating no systematic bias.
2. **Consistency:** The error remains confined within the theoretical 3σ envelope ($\pm 3\sqrt{P_{ii}}$) for approximately 99% of the time. This confirms that my tuning of P correctly captures the actual uncertainty of the system.

2.5 Future Work and Improvements

While the current EKF implementation provides stable flight, I have identified several key points for improvement to further enhance precision and robustness.

2.5.1 Performance Metrics

Currently, validation is primarily visual. I plan to integrate automated metric calculation:

- **RMSE (Root Mean Square Error):** To quantify the absolute accuracy of the attitude estimation over a full flight trajectory.
- **NEES (Normalized Estimation Error Squared):** A rigorous statistical test to detect filter over-confidence or under-confidence.

2.5.2 Evaluation framework using Rosbags

To optimize parameters rigorously, I aim to implement a replay workflow:

1. **Data Collection:** Record raw sensor data from challenging flights using 'ros2 bag record'.
2. **Offline Tuning:** Replay these bags into the EKF node while varying Q (process noise) parameters to minimize the RMSE.
3. **Benchmarking:** Compare my custom EKF performance against the industry-standard PX4 EKF2.

3. Alexa Voice Control

Author: Youssef Miri

3.1 Simulation and Testing Environment

Before implementing any AI features, I needed a way to test flight commands without the risk of a physical crash. I used the **ArduPilot SITL (Software In The Loop)** simulation, which allows the flight code to run on a local machine as if it were on a real flight controller.
For the ground station interface, I used **QGroundControl (QGC)**. This setup was essential for verifying MAVLink packets, visualizing real-time telemetry, and performing safe failure testing of the safety watchdog in a virtual environment.

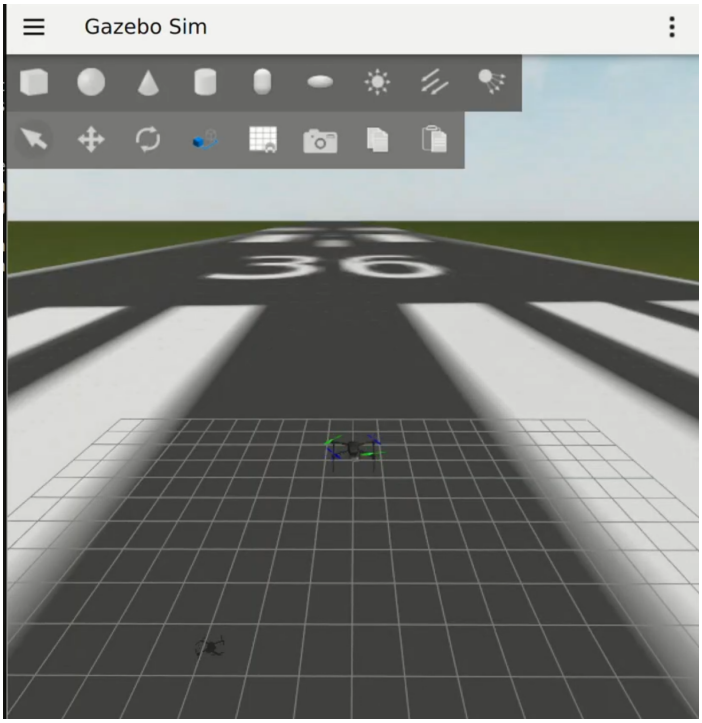


Figure 3.1: A screenshot of the ArduPilot simulation and QGroundControl interface.

3.2 The Concept

The goal was to move away from complex joysticks and instead interact with the drone using natural language. By simply saying "Alexa, move forward five meters," the system bridges the gap between a live microphone and the flight controller.

3.3 How the AI Capabilities Work

The "brain" of this system is a stack of different AI models working together to process audio into physical motion.

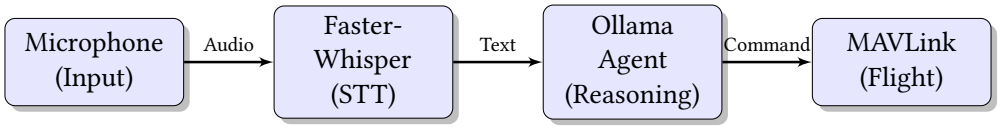


Figure 3.2: System Logic Flow: Converting voice to physical movement.

3.3.1 Listening and Transcribing

We used `PvPorcupine` for low-power wake-word detection. Once "Alexa" is recognized, the system records the command and passes it to `Faster-Whisper` for high-accuracy transcription.

3.3.2 The Reasoning Engine

We utilized a `LangChain` agent powered by `ChatOllama`. The model interprets natural language (e.g. "climb a bit") and maps it to specific tools like `control_drone_movement` with the appropriate parameters.

3.4 The Technical Backbone

The drone utilizes the MAVLink protocol via `DroneKit`. When a command is received, the script calculates coordinates in a Local NED (North, East, Down) frame.

3.4.1 Precise Movement

The script captures the starting position and monitors the distance moved in a continuous loop. The drone maintains its path until it is within a half-meter tolerance of the target coordinate.

3.5 Safety: The "Watchdog" Logic

To prevent unpredictable AI behavior, a safety watchdog overrides the agent if a human pilot takes control.

```
def mode_callback(self, attr_name, value):
    if value.name not in ['GUIDED', 'AUTO', 'LAND', 'BRAKE']:
        manual_override = True
        speak("Manual control detected.")
```

4. Human Tracking and Following

Author: Youssef Miri
This section details the second major AI feature: fully autonomous person tracking. This implementation utilizes a unified vision-and-depth approach to achieve higher reliability and smoother flight dynamics compared to traditional methods.

4.1 Hardware Architecture

The hardware was selected to maximize the onboard processing speed, which is critical to minimize latency in the control loop.

4.1.1 Compute Hardware: Jetson Orin Nano Upgrade

The tracking pipeline is powered by the **NVIDIA Jetson Orin Nano**. This module delivers significantly higher AI performance than standard modules, allowing us to run **YOLOv11** while simultaneously managing flight telemetry.

4.1.2 Unified Depth Sensing: Intel RealSense

We replaced traditional single-point Lidar with an **Intel RealSense Depth Camera**.

- **Full Depth Mapping:** The RealSense provides a complete depth map, preventing the system from losing the target during sharp turns.
- **Pitch Dynamics:** Distance data is smoothed using moving average filters and processed by a PID controller to maintain a constant 2.5-meter distance.

4.2 Control and Vision Pipeline

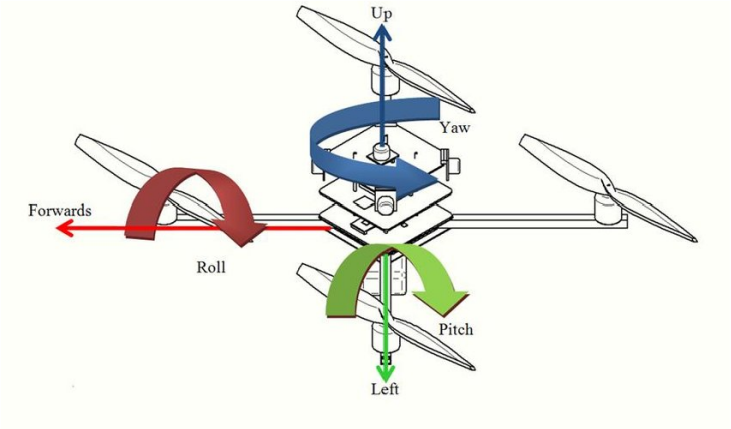


Figure 4.1: Three axes of movement for the quadcopter: Yaw, Pitch, and Height.

The vision system converts pixel-space detections into physical coordinate offsets to drive the motors.

4.2.1 Object Detection and Tracking Logic

The system uses the YOLOv11 architecture for real-time human detection. The model filters detections to ensure a human subject is locked before assigning tracking priority. The distance between the detection's center and the drone's heading creates a delta that drives the Yaw control.

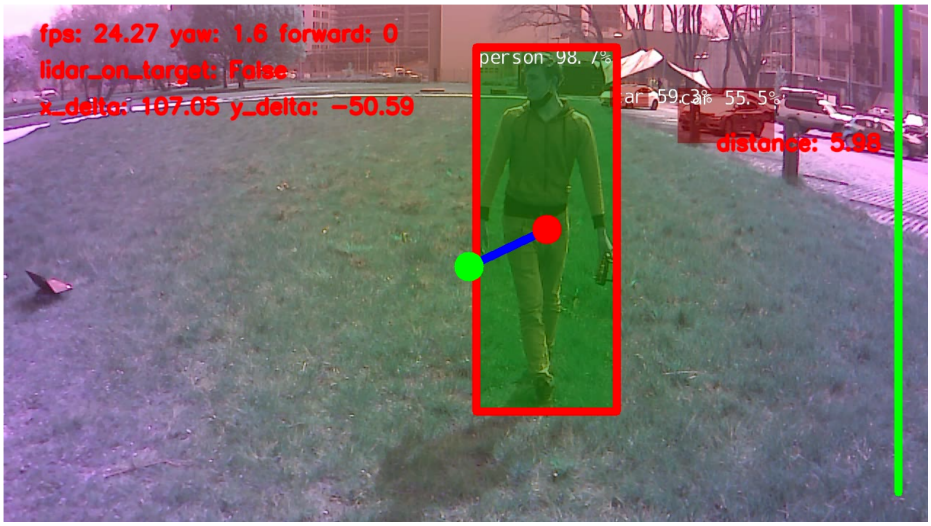


Figure 4.2: Visualization of the vision system showing YOLO detection and center-point delta.

4.2.2 The Autonomous State Machine

The drone operates in four distinct states:

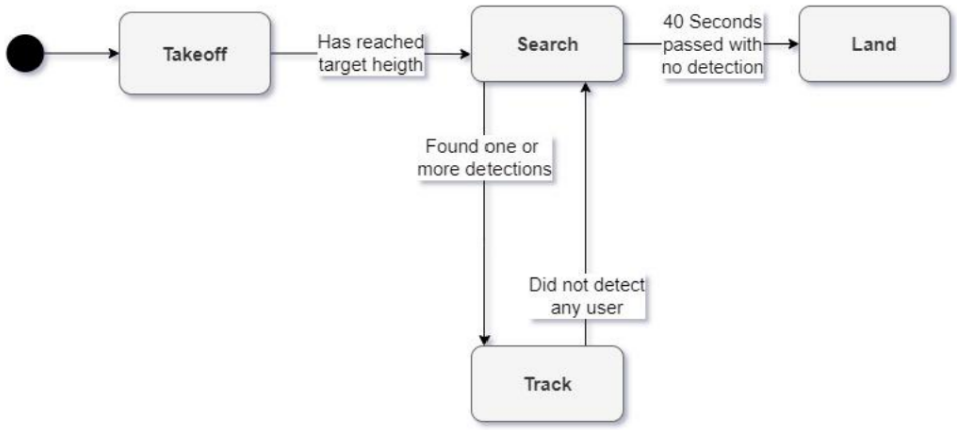


Figure 4.3: Control system state diagram: Transitioning from Search to Track.

1. **Takeoff:** The drone arms and ascends to a safe static height of 3 meters.
2. **Search:** The drone performs a slow Yaw rotation to find a target. If detection fails for 40 seconds, a failsafe landing is triggered.
3. **Track:** This is the active state where PID loops manage Yaw and Pitch to follow the target.
4. **Land:** The system lands and disarms when the mission is complete or the target is permanently lost.

5. Autonomous Drone Landing using Reinforcement Learning

5.1 Introduction

Autonomous landing on moving platforms presents significant challenges due to non-stationary dynamics, distinguishing it from static landing scenarios. This project leverages Deep Reinforcement Learning (DRL) to control a quadrotor landing on a stochastic moving target.
We propose a custom Gymnasium-MuJoCo environment modeling the drone-platform interaction and a "track-then-descend" reward curriculum. Furthermore, we provide an experimental analysis highlighting the superior stability of on-policy methods (PPO) over off-policy methods (SAC) in tasks with strict terminal constraints.

5.2 Problem Formulation

5.2.1 Environment Dynamics

The simulation is built on the MuJoCo physics engine.

- **The Agent:** A quadrotor with 4 continuous actuators (thrusts).
- **The Platform:** A ground vehicle moving with a forward velocity $v_x \in [0.25, 0.29]$ m/s. To increase complexity, the platform performs a lateral sway $v_y = A\sin(\omega t + \phi)$, making the trajectory stochastic.

5.2.2 Observation Space

The state space $\mathcal{S} \in \mathbb{R}^{11}$ is designed to be platform-agnostic, relying on relative coordinates. The observation vector is defined as:

$$s_t = [\mathbf{a}_{t-1}, \theta, \mathbf{p}_{rel}, \mathbf{v}_{rel}] \tag{5.1}$$

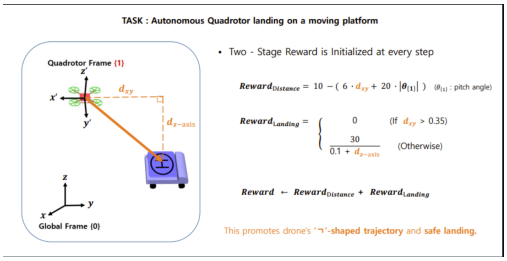
Where $\mathbf{a}_{t-1}$ is the previous action buffer (smoothness), θ is the pitch angle, $\mathbf{p}_{rel}$ is the relative position, and $\mathbf{v}_{rel}$ is the relative velocity.

5.2.3 Reward Function

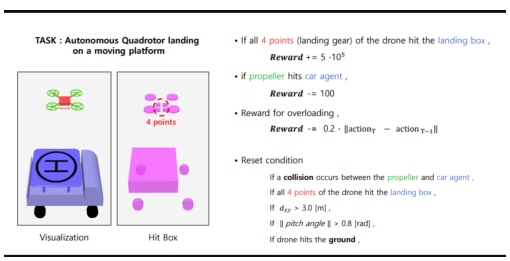
To solve the sparse reward problem, we implemented a two-stage dense reward function (Fig. 5.1a):

1. **Tracking:** Penalizes distance d_{xy} and pitch θ.
2. **Landing:** Bonus for minimizing height d_z when close ($d_{xy} < 0.35$).

Crucially, a massive sparse reward ($+5 \cdot 10^5$) is granted only if **all 4 landing gears** touch the pad simultaneously (Fig. 5.1b).



(a) Reward & Frames



(b) Task & Hitbox Constraints

Figure 5.1: RL Formulation. (a) The shaped reward guides the agent to track before descending. (b) Success requires simultaneous contact of all 4 points.

5.3 Methodology

We compared two distinct RL paradigms using *Stable-Baselines3*:

- **PPO (On-Policy):** Uses a clipped objective function. We hypothesized it would be stable but potentially sample inefficient.
- **SAC (Off-Policy):** Maximizes entropy, we assumed it would learn faster with better exploration.

Training Setup: Both agents were trained for $2 \cdot 10^6$ steps using 16 parallel environments.
Note on constraints: PPO had a max episode length of 1000 steps. SAC was restricted to 265 steps to artificially encourage faster convergence, as early tests showed it struggled to terminate episodes.

5.4 Experimental Results

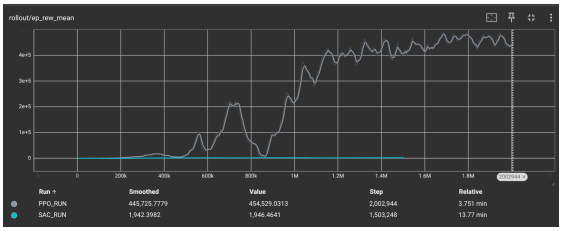
5.4.1 Computational Efficiency

PPO trained much more efficiently in our setup, achieving $\approx 8,800$ FPS versus SAC's $\approx 1,800$ FPS.

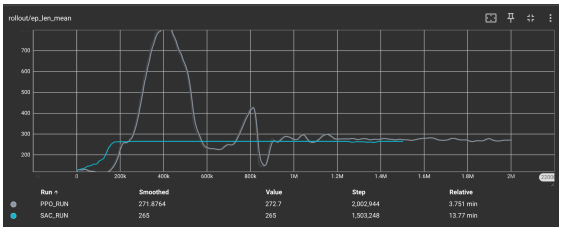
5.4.2 Performance Analysis

The training curves (Fig. 5.2) reveal a critical difference in behavior.

1. **Success vs. Local Optimum:**
   The Reward curve (Fig. 5.2a) shows PPO (gray) reaching the success threshold ($\approx 4.5 \cdot 10^5$). SAC (blue) flatlines near 2000. Qualitative analysis confirms that SAC learns to perfect the *Tracking* stage (following the car) but refuses to land. Landing is risky (potential crash = penalty), so the entropy-maximizing agent settles for the "safe" local optimum of hovering above the target.
2. **Episode Length & Time Limits:**
   Figure 5.2b confirms this diagnosis.
   - **SAC:** The curve saturates at exactly 265 steps (the imposed limit). The agent neither crashes nor lands, it waits for the timeout to maximize the accumulation of small distance rewards.
   - **PPO:** Despite a higher limit of 1000, PPO converges to ≈ 270 steps. It learns to perform the task efficiently rather than stalling.



(a) Mean Episode Reward



(b) Mean Episode Length

Figure 5.2: Training results. (a) PPO learns the task (high reward), while SAC fails. (b) SAC hits the time limit (265) constantly, while PPO lands efficiently.

5.5 Conclusion

We successfully trained a quadrotor to land on a moving platform using PPO. The comparison highlights that while SAC is theoretically sample-efficient, it got stuck in a local optimum of "safe tracking." PPO, with its stable updates, successfully navigated the sparse reward cliff to achieve consistent landings. Future work could involve transferring this policy to a realistic flight controller to bridge the sim-to-real gap.

Bibliography

[1] Nguyen Xuan Mung, Tiep Huu Nguyen, Loan Thi Kim Au, Nguyen Ngoc Anh (2022). *Finite-time control for a UAV system based on finite-time disturbance observer*. Available at: https://www.mdpi.com/2227-7390/13/11/1810.

[2] PX4 Autopilot Developers (2024). *PX4 Simulation (SITL) Documentation*. Available at: https://docs.px4.io/main/en/simulation/.

[3] Open Robotics (2023). *ROS 2 Humble Hawksbill: Documentation and Middleware Architecture*. Available at: https://docs.ros.org/en/humble/.

[4] Open Robotics (2022). *Gazebo Simulator: Physics-based Simulation for Robotics*. Available at: https://gazebosim.org/.

[5] Faconti, D. (2023). *PlotJuggler: Real-Time Data Visualization Tool for Robotics*. Available at: https://plotjuggler.io/.

[6] DeepMind (2024). *MuJoCo: Multi-Joint dynamics with Contact*. Available at: https://mujoco.org/.

[7] Raffin, A., et al. (2021). *Stable-Baselines3: Reliable Reinforcement Learning Implementations*. Available at: https://github.com/DLR-RM/stable-baselines3.

[8] Farama Foundation (2024). *Gymnasium: A standard API for reinforcement learning environments*. Available at: https://gymnasium.farama.org/.

[9] Griaznov, D. (2023). *Custom MuJoCo Environment for RL*. Available at: https://github.com/denisgriaznov/CustomMuJoCoEnviromentForRL.

[10] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347.

[11] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. International Conference on Machine Learning (ICML).