

Deep Reinforcement Learning for Precise Multirotor Landing on Moving Targets

Kshitij Prasad

Matriculation Number: 00130477

First Examiner: Prof. Dr. Klaus Kefferpütz

Second Examiner: Prof. Dr. Bernd Martin Ebert

Company Supervisor (Akkodis): Mr. Bruno Simon

Ingolstadt, March 2026

Contents

1	Introduction	5
2	Introduction	5
2.1	Motivation	5
2.2	Problem Statement	5
2.3	Objectives and Contributions	5
3	Simulation Environment and Architecture	6
3.1	PX4 SITL and Gazebo Integration	6
3.2	Communication Bridge: uXRCE-DDS	7
3.3	The ROS2-Gazebo Bridge	8
3.4	Coordinate Frames: NED vs. ENU	8
3.5	Simulation Entities and Custom Environments	9
3.5.1	Custom Environment Generation: Gaimersheim Scenario	9
3.5.2	Vehicle Configuration: x500 with Downward Camera	10
3.5.3	The Dynamic Landing Platform	11
3.5.4	Real-Time ArUco Marker Detection and Pose Estimation	13
4	Classical Control Approach: Reactive Controllers	14
4.1	Problem Foundation – Selection of Control Architecture	14
4.2	Drone Model Dynamics	15
4.2.1	Idealized Model (Pure Integrator)	15
4.2.2	First-Order Lag Model (PT_1)	15
4.3	Coordinate Frame Selection	16
4.3.1	Frame Selection: NED vs ENU	16
4.4	Classical Control Design	16
4.4.1	Initial Design Thinking Process	16
4.5	Reactive Controllers	17
4.5.1	NavSat Based (GPS Only)	17
4.5.2	ArUco Vision-Based	18
4.6	Safety Mechanism Native to Both Controllers	19
4.6.1	Velocity Clamping	19
4.6.2	Soft Landing Zone	19
4.6.3	Timeouts and Fallbacks	19
4.7	Conclusion and Motivation	19
5	Fundamentals of Reinforcement Learning	19
5.1	Agent and Environment Loop	19
5.2	Markov Decision Processes	20
5.3	Policy	20
5.4	Return and Discounting	20
5.5	Value Functions	21

5.6	Bellman Equations	21
5.7	Policy Gradient Methods	21
5.7.1	Policy Gradient Theorem	22
5.7.2	REINFORCE	22
5.7.3	Actor and Critic Architecture	22
5.8	Algorithms Employed	22
5.8.1	Proximal Policy Optimization (PPO)	23
5.8.2	Twin Delayed Deep Deterministic Policy Gradient (TD3)	24
5.8.3	Soft Actor-Critic (SAC)	25
6	Environment Design	27
6.1	State Space	28
6.2	Action Space	28
6.2.1	Action Smoothing	29
6.2.2	Height-Dependent Velocity Limiting	29
6.3	Reward Function	29
6.3.1	Dense Shaping Terms	30
6.4	The Mathematics of the Hovering Exploit	30
6.5	Applied Fixes	32
6.6	New Mathematical Outlook	32
6.7	Analysis: Updated Reward Function	33
6.7.1	Descent and Hovering Rewards	33
6.7.2	Proximity Bonuses	34
6.7.3	Terminal Rewards	34
6.8	Episode Termination	34
6.9	Reset Procedure	35
6.10	ROS 2 and PX4 Integration	35
7	Control Barrier Functions for Safe Reinforcement Learning	36
7.1	Theoretical Background	37
7.1.1	Safe Sets	37
7.1.2	The CBF Condition	37
7.1.3	Integration with RL	37
7.2	Safety Mechanisms for the Landing Task	38
7.2.1	Platform Boundary CBF (h_1)	38
7.2.2	Ground Collision CBF (h_2)	38
7.3	Algorithm-Specific CBF Variants	39
7.3.1	PPO-CBF Configuration	39
7.3.2	SAC-CBF Configuration	39
7.4	Extended Observation Space	40
7.5	CBF Reward Modifications	40
7.6	Intervention Rate as a Training Diagnostic	41

8	Reward Comparison Across Environment Variants	41
8.1	Per-Step Shaping Reward Differences	41
8.2	Terminal Reward Differences	42
8.3	Implications for Cross-Environment Reward Comparison	43
8.4	Metrics That Permit Fair Comparison	43
9	Training Procedures	44
9.1	Stable-Baselines3	44
9.2	Software Dependencies	44
9.3	Launching the Training Pipeline	44
9.4	Hyperparameter Selection and Config	45
9.4.1	PPO Configuration and Selection Rationale	45
9.4.2	SAC Configuration and Selection Rationale	48
9.4.3	TD3 Configuration	50
10	Monitoring Training with TensorBoard	50
11	Comparative Evaluation of Landing Methods	53
11.1	Evaluation Methodology	53
11.1.1	Simulation Setup	53
11.1.2	Performance Metrics	53
11.1.3	Evaluation Protocol	54
11.2	Quantitative Results	54
11.2.1	Summary of Key Metrics	54
11.2.2	Success Rate Analysis	54
11.2.3	Landing Precision	55
11.2.4	Landing Time and Efficiency	55
11.3	Trajectory Analysis	56
11.3.1	Descent Profiles	56
11.3.2	Horizontal Distance Over Time	56
11.3.3	2D Landing Trajectories	57
11.3.4	3D Flight Paths	58
11.3.5	Mean Trajectory with Confidence Bands	59
11.4	Statistical Comparison	60
11.5	Combined Dashboard	61
11.6	Reactive vs. Reinforcement Learning Methods	62
11.7	Comparison Among RL Methods	62
11.7.1	PPO vs. SAC	63
11.8	Comparison Among RL Methods	63
11.8.1	PPO vs. SAC	63
11.8.2	TD3 Performance	64
11.8.3	Effect of Network Architecture	64
11.9	Control Barrier Function Analysis	64

11.9.1	SAC + CBF Performance	64
11.9.2	PPO + CBF: The Hovering Problem	64
11.10	Discussion	65
11.10.1	Overall Ranking	65
11.10.2	Reward Function Design	65
11.10.3	Limitations of This Evaluation	66
12	Conclusion and Future Work	66
12.1	Summary of Contributions	66
12.2	Key Findings	67
12.3	Future Work	67
12.4	Final Remarks	68

1 Introduction

2 Introduction

2.1 Motivation

The global drone delivery market in 2024 was valued at approximately USD 251 million, which is projected to reach USD 65 billion by 2034, this translates to a Compound Annual Growth Rate (CAGR) of 76.1%. Companies like Amazon Prime Air, Wing (Alphabet) and Zipline have already setup the infrastructure and resources to conduct real-world operations.

Beyond commercial delivery, autonomous Unmanned Aerial Vehicles (UAVs) are increasingly critical in domains where human access is dangerous, slow, or infeasible, these include emergency services (landing on moving ambulances), military logistics (resupply to mobile ground units), and maritime operations with UAV recovery on ships in motion.

Despite this momentum a fundamental bottleneck persists across all these applications. Conventional Global Navigation Satellite System (GNSS) provide 2.5-meter accuracy in open areas, a dense city topology – an urban canyon, interference, and multipath effects can degrade this to 10+ meters, this becomes unacceptable for landing on bound targets that are confined to a sub meter radius such as vehicle rooftops, ship decks or designated landing pads.

To compensate for the GNSS limitations, visual fiducial markers like ArUco tags can be thought of as an inexpensive and lightweight solution that are easily detectable by a monocular camera, allowing for relative pose estimate of the target with respect to the camera on the UAV. This can provide centimeter level accuracy, and using this proposition the UAV can align vertically and horizontally to the marker and guide the vehicle to a precise touchdown.

2.2 Problem Statement

This thesis proposes a more complex scenario where the target is no longer stationary. A moving target includes additional challenges such as continuous state estimation of the platform, predictive trajectory planning, robustness and safety constraints during platform velocity changes. Traditional approaches to precision landing rely on sensor feedback (GPS, visual markers) and reactive controllers. These are computationally efficient but struggle with the non-linear dynamics during final approach and adapting to varying environments. Learning based methods have shown promise in complex control tasks but face challenges of safety concern during exploration and sim-to-real transfer gaps. Traditional controllers need hand tuning of components which can be exponentially complex for difficult tasks and is hard to scale. Learning based approach handle these parametric calculations internally.

2.3 Objectives and Contributions

This thesis proposes a comparative study of reactive and learning-based control strategies for UAV landing on a moving platform, evaluated within a high-fidelity Gazebo-PX4 Software-In-The-Loop (SITL) simulation environment interfaced through ROS 2.

The initial objective is to develop a traditional reactive control baseline using sensor data (GPS and ArUco) which envelop the tracking of the platform that allows for a precise landing.

The second objective is to train and evaluate three state of the art deep reinforcement learning algorithms for the moving platform landing tasks, these are: Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), and Twin Delayed Deep Deterministic Policy Gradient (TD3). As learning approaches are not very stable in the beginning and the output of the agent is not deterministic, we integrate Control Barrier Functions (CBF) with the policies to ensure safety, specifically bounded position error and safe descend velocity, both during training and deployment. Finally, this thesis aims at a quantitative and qualitative comparison of all the methods in terms of success rate and landing accuracy.

3 Simulation Environment and Architecture

3.1 PX4 SITL and Gazebo Integration

Software in the Loop (SITL) simulation enables the PX4 flight stack to run on a host computer (Linux/Ubuntu), controlling the software-modeled vehicle in a simulated environment. Contrary to Hardware in the Loop (HITL), where the flight stack runs on the actual flight controller hardware, the SITL interacts with a simulator software (Gazebo) from which it receives sensor data and sends actuator commands just as it normally would.

One of the major advantages of utilizing the SITL architecture is mitigating the risks and costs associated with physical flight testing. It also allows for rapid development cycles for complex autonomous behaviors such as landing on a moving target—which is undertaken in this thesis. As illustrated in Figure 1, the SITL architecture connects various components via UDP and TCP network protocols. Such a structure allows the flight stack to communicate with ground control stations, offboard APIs (ROS 2 Nodes), and the physics engine simultaneously. This modular nature allows the isolation of individual components, making it easier to debug specific issues related to guidance, control logic, or navigation before deploying to the field.

The modern Gazebo simulator (formerly Gazebo Ignition) was the first choice as our primary physics and rendering engine for the development and testing process of the autonomous landing. Apart from Gazebo, one could have used Isaac Sim or jMAVSim. However, Isaac Sim requires high GPU resources, and for MAV simulation, the ROS 2 integration is not as well-established as it is for Gazebo.

For autonomous landing tasks relying on computer vision, the simulator must provide photorealistic rendering and accurate lighting. Gazebo utilizes the Ogre 2.x rendering engine, which supports Physically Based Rendering (PBR). This ensures that visual data fed into the drone’s simulated camera closely approximates real-world video feeds, including shadows and texture details essential for detecting landing targets. Gazebo has a vast, mature ecosystem of pre-built sensor plugins. In our case, the downward-facing camera for the x500 is more of a plug-and-play experience rather than requiring us to model it and set up a custom-designed perception stack from scratch. Furthermore, Gazebo supports the uXRCE-DDS bridge, facilitating high-throughput data exchange between the PX4 autopilot and ROS 2 nodes running reinforcement learning agents.

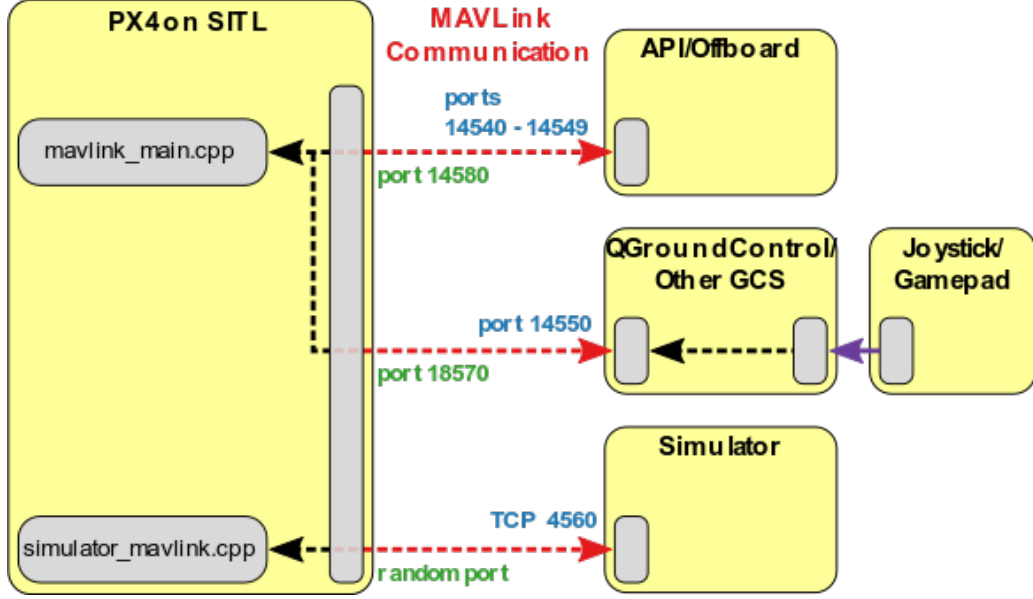


Figure 1: PX4 SITL communication architecture. The diagram depicts the interactions between the simulator backend, main MAVLink module, QGroundControl, and external APIs.

3.2 Communication Bridge: uXRCE-DDS

To link the PX4 flight stack with an external reinforcement learning agent(s) requires a robust middleware. We employ the **uXRCE-DDS** (eXtremely Resource Constrained Environments Data Distribution Service) bridge.

This architecture replaces the legacy FastRTPS bridge. It consists of a client running on the PX4 autopilot and an agent running on the companion computer (host machine). The client publishes uORB topics (internal PX4 messages) to the DDS global data space, making them accessible to ROS 2 nodes.

PX4 uses uORB messages, while the ROS 2 ecosystem uses standard `geometry_msgs` or `sensor_msgs`. A translation node is required to convert between these formats. For instance, a `VehicleCommand` sent by our RL agent to arm the drone is translated into the specific uORB topic structure required by the PX4 kernel.

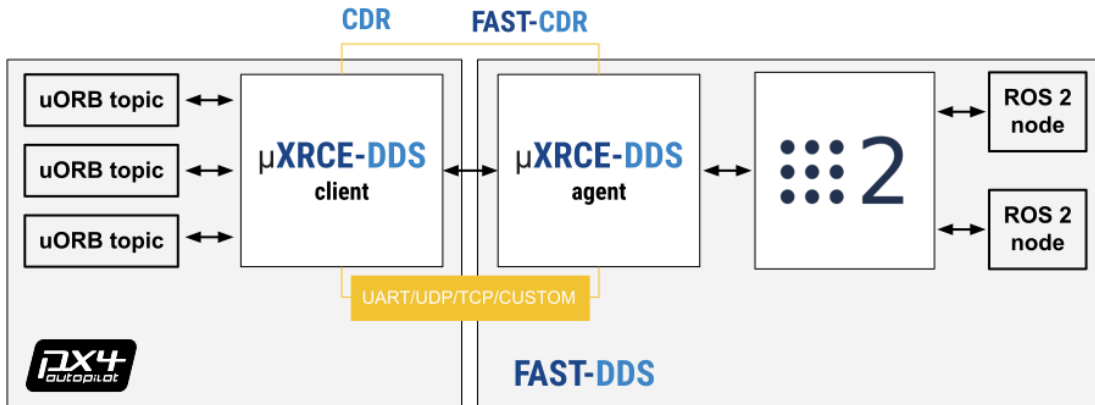


Figure 2: The uXRCE-DDS architecture bridging PX4 uORB topics with ROS 2 nodes via the Fast-DDS middleware.

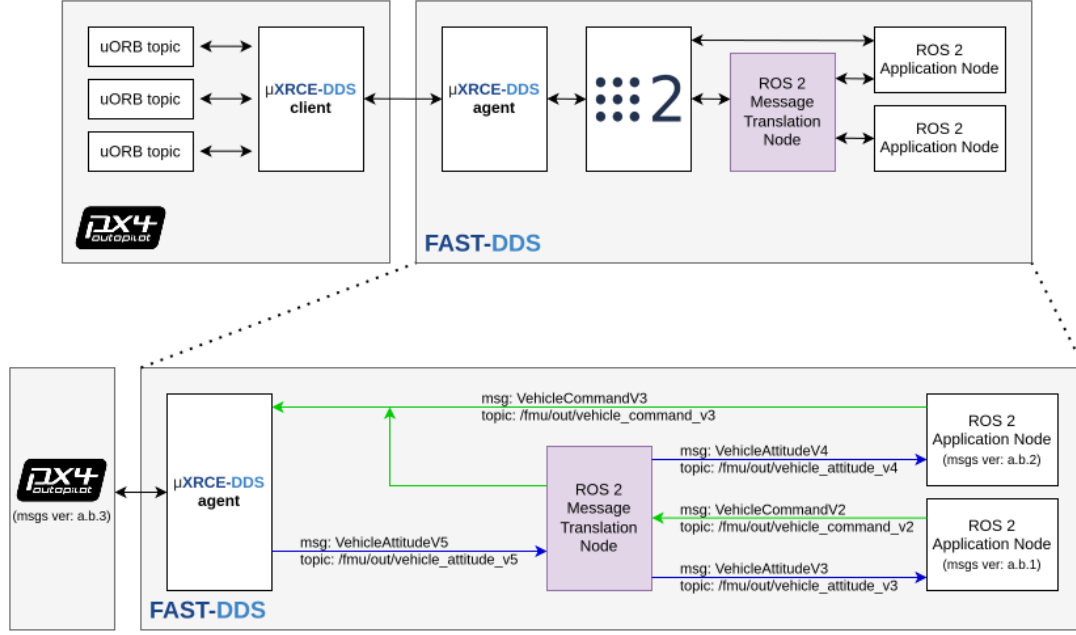


Figure 3: Data flow diagram including the ROS 2 Message Translation Node, which converts standard ROS 2 messages into PX4-specific uORB formats and vice versa.

3.3 The ROS2-Gazebo Bridge

To have our data from the Gazebo Simulation Environment into the ROS2 Framework such that the autonomous landing tasks via the **ArUco** and **Landing Node** are to be executed successfully, we established parameter bridges via utilizing the `ros_gz_bridge` package. The middleware ensures that the data generated at the Simulation end is accessible to the ROS2 Nodes for processing.

The moving platform’s GPS data is transferred by bridging the Gazebo topic:

`/world/moving_platform/.../navsat`

(carrying `gz.msgs.NavSat` messages) to the ROS 2 topic `/platform/navsat` (converted to `sensor_msgs/msg/NavSatFix`). This **NavSat** data allows our `landing_node` / or the `reinforcement_landing` node to receive precise geolocation data about the platform’s current position.

Simultaneously, to facilitate the operation of the `pl_aruco` node, we bridged the downward-facing camera data. The bridge was configured to forward raw imagery and camera parameters from the simulated `x500_mono_cam_down_0` model by converting `gz.msgs.Image` and `gz.msgs.CameraInfo` types into their corresponding ROS `sensor_msgs` equivalents. This ensured that the OpenCV algorithm has the necessary Image data fed for real time to pose estimation.

3.4 Coordinate Frames: NED vs. ENU

A significant challenge faced in this simulation setup is the discrepancy between the coordinate conventions. PX4 and ROS 2 utilize different frames of reference, requiring careful handling within the training environment code.

PX4 operates natively in the **NED** (North-East-Down) frame. In this system, the X axis points North, the Y axis points East, and the Z axis points Down (meaning altitude is represented

as a negative Z value). The body frame of the drone (x^B, y^B, z^B) aligns with this convention, as shown in Figure 4. Conversely, Gazebo and ROS 2 utilize the **ENU** (East-North-Up) frame. In this system, the X axis points East, the Y axis points North, and the Z axis points Up.

In the training environment, this conversion is handled explicitly. As we discuss in the later sections, when the RL agent calculates position error based on GPS coordinates, it calculates a North/South difference (corresponding to X in NED) and an East/West difference (corresponding to Y in NED). These values are then fed into the observation space consistent with the PX4 guidance logic. This ensures that a positive velocity command in the X-axis moves the drone North, regardless of the underlying simulator’s coordinate system.

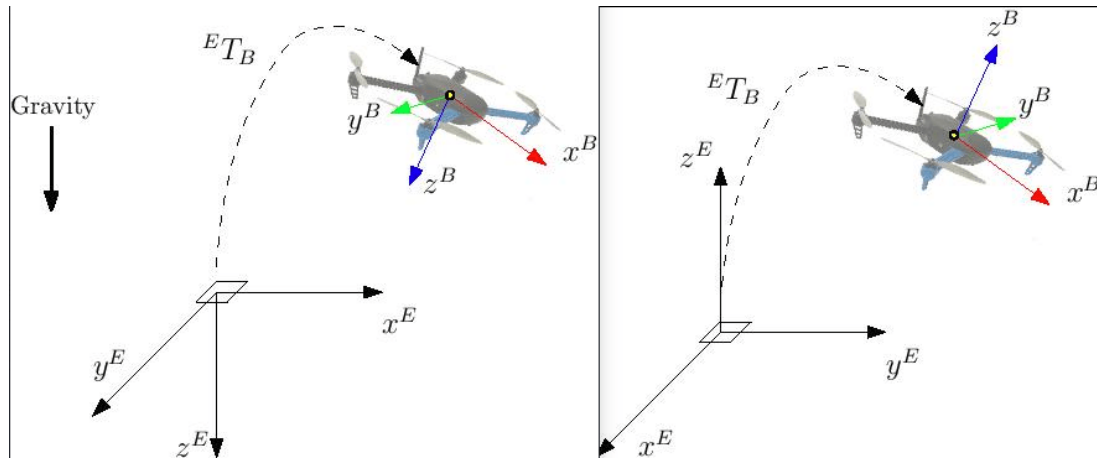


Figure 4: Visual representation of the coordinate frames. The diagram illustrates the relationship between the Earth-fixed frame (x^E, y^E, z^E) and the Body-fixed frame (x^B, y^B, z^B).

3.5 Simulation Entities and Custom Environments

3.5.1 Custom Environment Generation: Gaimersheim Scenario

Initially, the goal was to build a custom environment of the Gaimersheim Akkodis building area to validate and test the operation of the RL in relevant environments beyond generic empty worlds. Using tools like Blender OSM, we generate custom simulation worlds mirroring real-world locations. While the real-world execution of such a scenario was out of scope, the workflow to develop and import a custom environment into Gazebo for testing, training, and validation is briefly discussed below.

This workflow involves importing OpenStreetMap data for buildings and terrain into Blender, applying textures, and exporting the scene as a Simulation Description Format (SDF) file. This file is then loaded into Gazebo, providing a digital twin of the target deployment zone (Figure 6). By testing in this environment, we can verify that the visual navigation system is robust against visual clutter and occlusions typical of urban or industrial settings.

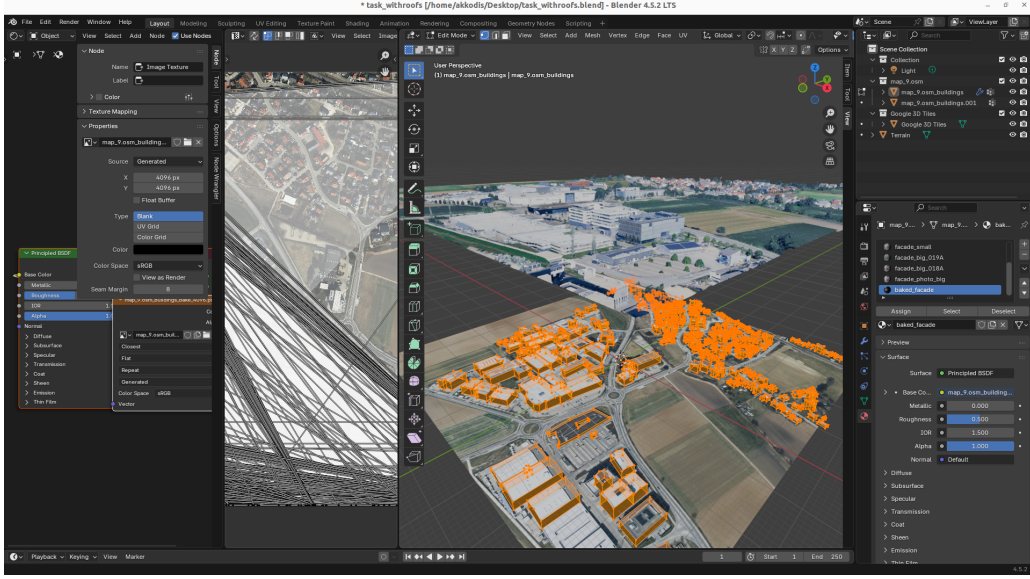


Figure 5: Workflow for creating a custom simulation world. Real-world building and terrain data are imported into Blender using the Blender OSM plugin, textured, and then exported as SDF models for Gazebo.

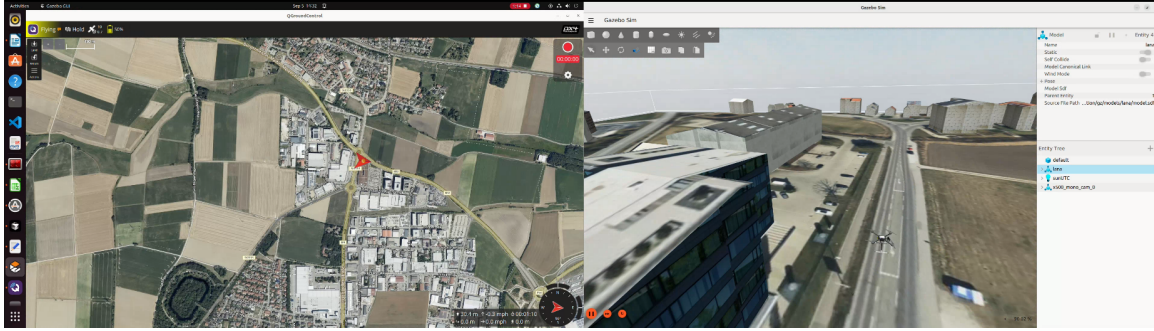


Figure 6: The SITL simulation running in the custom Gaimersheim Akkodis environment. Left: QGroundControl displaying the drone’s position on the satellite map. Right: Gazebo visualization of the drone flying over the 3D-reconstructed buildings.

3.5.2 Vehicle Configuration: x500 with Downward Camera

The primary vehicle model used in our simulations is the **x500**, a standard quadrotor airframe supported by PX4. To facilitate the autonomous landing task, we utilize the specific configuration `x500_mono_cam_down`.

The `x500_mono_cam_down` is equipped with a vertically downward-facing monocular camera mounted on the underside of the drone’s frame. In the simulation, the camera captures a stream that includes the landing platform and the ArUco target marker. This allows for visual odometry and platform pose detection—specifically the relative position error (e_x, e_y) , which we discuss in later sections.

Figure 7 visualizes the x500 model hovering directly above the landing target. The simulation provides the raw image data via ROS 2 topics, which our reinforcement learning agent consumes to make control decisions.

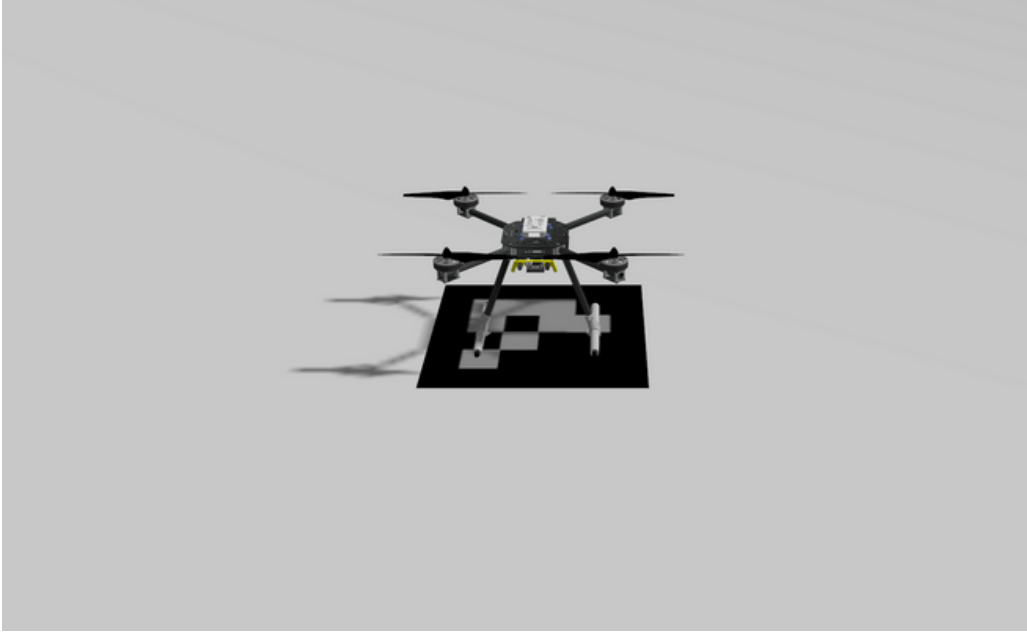


Figure 7: The `x500_mono_cam_down` vehicle model in Gazebo. The drone rests on the custom-designed platform with a fiducial marker to assist in precise localization while landing.

3.5.3 The Dynamic Landing Platform

The PX4-Autopilot repository comes with simulation tools, and that package comprises various models that one can choose to set up in the Gazebo environment for training, testing, or validating. One such model is the moving platform model. By default, the platform in this world moves randomly in the X, Y, and Z directions. Unlike static targets, this platform emulates the behavior of ships or trucks, introducing velocity and heading variables that the drone must track. Figure 8 shows the platform in the simulation environment, illustrating the stark contrast between the marker and the ground plane, which simplifies the initial phases of computer vision training.

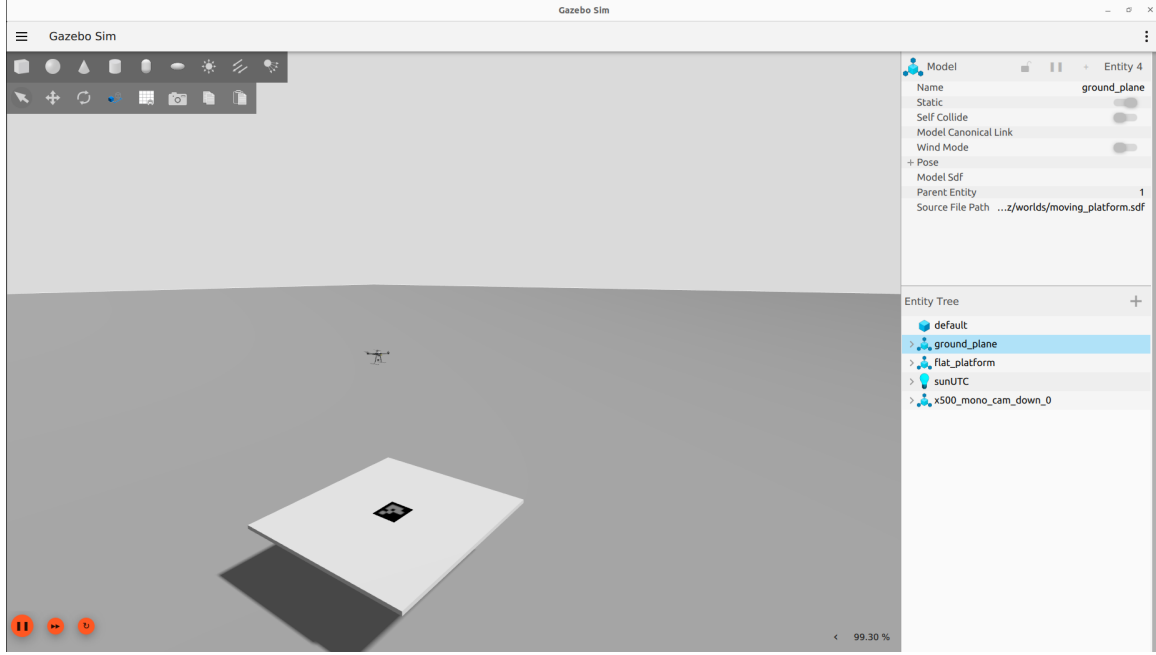


Figure 8: The simulation environment featuring the moving platform. The high contrast of the ArUco marker against the white platform facilitates robust detection during training exercises.

ArUco Marker Integration: To enable precision landing, we modified the platform’s SDF definition (`model.sdf`) to include a visual fiducial marker. We define a `visual` element named `aruco_visual`, which is a flat plane positioned 0.051 meters above the platform surface to prevent Z-fighting artifacts during rendering. The material script applies the `arucotag.png` texture to this plane.

Custom Moving Platform Controller Plugin: To control the movement of the platform deterministically, we utilize a custom Gazebo system plugin: `libMovingPlatformController.so`. This C++ plugin allows us to modify the platform’s velocity, direction, and oscillation behavior directly through the simulation parameters, without recompiling code.

Kinematic Formulation of the Moving Platform:

The behavior of the platform is defined by the custom controller, which prescribes a deterministic trajectory. We need this flexibility as in later sections we discuss how this allows us to create a curriculum of training scenarios, from completely static to complex multi-coordinate motion. Instead of directly setting waypoints, the platform’s position $(x(t), y(t))$ at any given time t is governed by sinusoidal kinematic equations:

$$x(t) = x_0 + A_x \sin\left(\frac{2\pi}{T}t\right) \quad (1)$$

$$y(t) = y_0 + A_y \sin\left(\frac{2\pi}{T}t + \phi_y\right) \quad (2)$$

where A_x and A_y are the respective spatial amplitudes along the X and Y axes, T is the oscillation period, and ϕ_y introduces a phase shift.

Controller Logic Analysis:

The logic resides in the `MovingPlatformController.cpp` file. At every simulation update, the

controller operates by calculating the trajectory setpoint defined by the equations above. For example, circular tracking or a figure-eight (Lissajous) pattern can be introduced by activating amplitudes in both the x and y directions.

The controller applies a wrench to the platform link to drive it once the setpoint is determined. This utilizes a Proportional-Derivative (PD) controller logic to minimize the error between the current platform state and the target trajectory. This method ensures that the platform moves physically within the simulation, interacting with gravity and other forces, rather than simply teleporting to a new coordinate.

3.5.4 Real-Time ArUco Marker Detection and Pose Estimation

A major self-discussion whilst developing the approach to create a package that aimed at autonomously landing a drone on a platform investigated the pose calculation of the platform relative to the drone.

To do the above mentioned we had two ways:

1. One approach aimed at using the GPS Data from the Platform data published as `/platform_navsat` that we had transported from Gazebo using the ROS-Gazebo bridge methodology as discussed in Section 1.
2. The other approach aimed at using the x500 variant with a downward facing camera that can identify a marker on the platform and using the OpenCV pipeline.

Implementation of the ArUco Pose Detection: To be able to achieve the high precision terminal control we developed a ROS2 Package aimed at having a ROS2 Node that would publish the platforms pose. The `PoseStamped` message published to `/p1/pose_cam` represents the position and orientation of the **ArUco Marker** expressed relative to the **Camera Optical Center**.

We built the perception system on OpenCV, specifically using the `detectMarkers` and `estimatePoseSingleMarkers` functions to resolve the PnP (Perspective-n-Point) problem. This calculates the marker's pose relative to the camera's optical frame. To handle images, we skipped the standard `cv_bridge` package in favor of a **custom decoder** that manipulates the memory buffer directly for NumPy array generation.

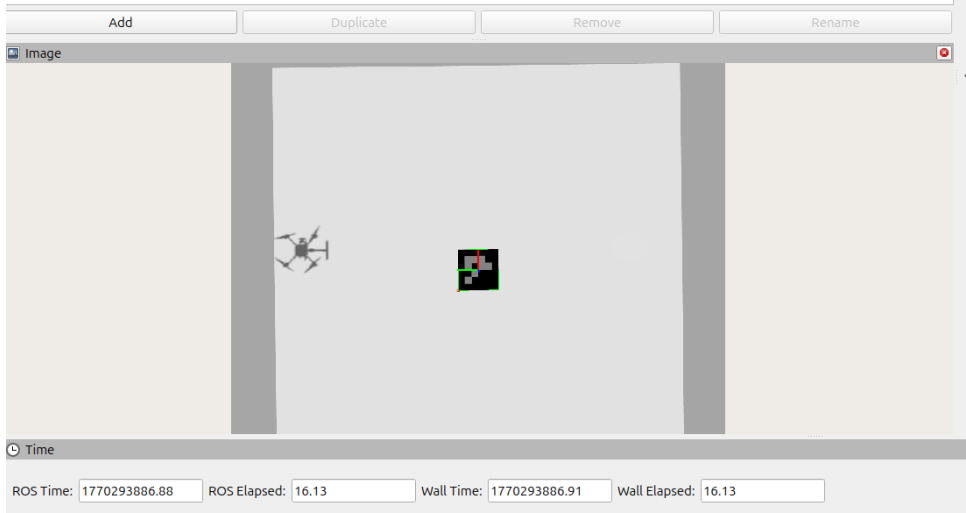


Figure 9: The figure visualizes the pose detection of the platform in RViz2

4 Classical Control Approach: Reactive Controllers

The idea behind developing the reactive controllers was to benchmark the baseline performance with the Reinforcement Learning methods. The Reactive Control approaches discussed in this work represent the traditional robotics approach based on domain knowledge and principles from control theory.

4.1 Problem Foundation – Selection of Control Architecture

Modern UAVs running the PX4 autopilot stack are equipped with multiple control interfaces:

- **Position Control Mode:** User Inputs the desired X, Y and Z coordinates and the Autopilot handles everything.
- **Velocity Control Mode:** User Inputs the desired velocities (V_x, V_y, V_z) and the Autopilot stabilizes the altitude.
- **Attitude Control Mode:** User Inputs the roll/pitch/yaw angles and thrusts which leads to manual stabilization.
- **Rate Control Mode:** User Inputs angular rates + thrust which is fully manual or the acrobatic mode.

Based on the above-mentioned control interfaces, the velocity control mode is apt for the following reasons:

1. **Abstraction Level:** A very high abstraction (the position mode) would necessarily mean that there would be no learning needed, and the Autopilot does everything. A very low abstraction would mean that the agent must learn aerodynamics, rotor physics and stabilization which probably would have led to extremely complex state space and a very slow convergence during the training for the RL Models.

2. **Safety:** As the velocity commands can be clamped, one can bound the maximum speeds to prevent crashes. In the same manner, attitude commands, if misconfigured, can cause flips or tumbles. Another important reason for this was to allow soft limits on approach (critical for landing safely).
3. **Generalization:** Velocity control works across different drone models without retraining, whereas attitude control is very specific to the hardware due to different inertias and motor configurations.

The drone model in velocity mode acts as an integrating plant which leads to the velocity commands being integrated to position by the flight controller, thus it doesn't involve the controller to deal with the low-level motor mixing or attitude stabilization.

4.2 Drone Model Dynamics

To properly design the controller, the dynamic relationship between the commanded velocity v_{com} and the resulting drone position x must be understood. Two models are considered:

4.2.1 Idealized Model (Pure Integrator)

Under the assumption that the drone instantly tracks the commanded velocity ($v = v_{\text{com}}$), the dynamics reduce to a pure integrator:

$$\dot{x} = v_{\text{com}} \implies X(s) = \frac{1}{s} V_c(s) \quad (3)$$

Stability with P-control ($u = -K_p \cdot e_x$): The closed-loop system has a single pole at $s = -K_p$. The root locus shows that this pole moves left along the real axis for increasing $K_p > 0$, so the system is **stable for any** $K_p > 0$ with **no oscillations**. Under this model, a P-controller alone would suffice.

4.2.2 First-Order Lag Model (PT₁)

In practice, the drone's velocity v does not instantly match v_{com} due to motor dynamics, inertia, and the internal PX4 velocity loop. A more realistic model introduces a first-order lag with time constant T :

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -\frac{1}{T} v + \frac{1}{T} v_{\text{com}} \end{aligned} \quad (4)$$

In the Laplace domain:

$$X(s) = \frac{1}{s(1 + sT)} V_c(s) \quad (5)$$

Stability with P-control: The open-loop transfer function now has poles at $s = 0$ and $s = -1/T$. The root locus shows that for increasing K_p , the poles move toward each other, enter the complex plane, and while the system remains **stable for all** $K_p > 0$, **oscillations become**

possible for large gains. This explains the oscillatory behavior observed with P-control in practice.

Identification: The time constant T could be identified experimentally by means of a step response in velocity. A step command v_{com} applied to the drone and the measured velocity response would reveal the lag dynamics.

4.3 Coordinate Frame Selection

NavSat (GPS) data from the ROS2-Gazebo bridge provides the raw data in latitude, longitude, and altitude, whereas the velocity commands use local Cartesian coordinates.

4.3.1 Frame Selection: NED vs ENU

PX4 natively uses the NED (North-East-Down) frame, hence needing no conversion for the velocity commands.

Altitude in NED: $z = 0$ at takeoff and $z > 0$ means below takeoff level. The logic here is that when z tends to a large positive value, the drone is descending toward the ground.

4.4 Classical Control Design

4.4.1 Initial Design Thinking Process

Control Objective: Minimize:

$$e(t) = [x_{\text{drone}}(t) - x_{\text{platform}}(t), \quad y_{\text{drone}}(t) - y_{\text{platform}}(t)] \quad (6)$$

Subject to $z_{\text{drone}}(t) \rightarrow z_{\text{platform}}(t)$ monotonically (descent constraint).

Standard Control Architecture: The classical feedback control structure is:

$$w \longrightarrow \sum \longrightarrow G_R(s) \longrightarrow G_S(s) \longrightarrow x \quad (7)$$

where w is the reference (desired platform position), $G_R(s)$ is the controller transfer function, and $G_S(s)$ is the plant (drone dynamics).

System Identification: First order lag (velocity commands due to motor dynamics), Decoupled (at low speeds horizontal xy motion doesn't affect vertical motion z), each velocity command (v_x, v_y, v_z) affects one position coordinate.

Error Dynamics: Defining the position error $e_x = x_{\text{drone}} - x_{\text{platform}}$ and $e_y = y_{\text{drone}} - y_{\text{platform}}$, the error dynamics are:

$$\begin{bmatrix} \dot{e}_x \\ \dot{e}_y \end{bmatrix} = \begin{bmatrix} v_{x,c} - \dot{x}_p \\ v_{y,c} - \dot{y}_p \end{bmatrix} = \begin{bmatrix} -K_p e_x \\ -K_p e_y \end{bmatrix} - \begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix} \quad (8)$$

where $\dot{x}_p, \dot{y}_p$ are the platform velocities. Technically, any $K_p > 0$ stabilizes the error dynamics under the idealized model (Eq. 3).

Oscillation Analysis: Under the idealized pure-integrator model, a P-controller yields a first-order closed-loop system with no oscillations. However, the drone's internal velocity loop

introduces a first-order lag (PT₁), making the effective plant second-order. With P-control on a second-order plant, damping is insufficient and oscillations arise for higher gains.

Velocity-Feedback Control: To address this, drone velocity feedback is added to provide damping:

$$v_{x,\text{cmd}} = -K_p \cdot e_x - K_D \cdot v_{x,\text{drone}} \quad (9)$$

Note: this is not a true PD controller, which would be $u = -K_p e_x - K_D \dot{e}_x$. Here, $v_{x,\text{drone}}$ is the drone's absolute velocity, not the velocity error $\dot{e}_x = v_{x,\text{drone}} - \dot{x}_p$. However, since the platform velocity $\dot{x}_p$ is small relative to the drone's velocity during approach, the effect is similar.

State-Space Interpretation: This control law can be interpreted as a state-feedback controller. Defining the state vector $\mathbf{x} = [x_{\text{pos}}, v]^\top$, the control input is:

$$u = -\mathbf{K} \begin{bmatrix} x_{\text{pos}} \\ v \end{bmatrix} + K_p \cdot x_{\text{des}} \quad (10)$$

where $\mathbf{K} = [K_p, K_D]$.

The drone dynamics in state-space form with the PT₁ velocity model are:

$$\begin{bmatrix} \dot{x}_{\text{pos}} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{T} \end{bmatrix} \begin{bmatrix} x_{\text{pos}} \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{T} \end{bmatrix} v_{\text{com}} \quad (11)$$

The velocity-feedback term ($-K_D \cdot v$) effectively places both closed-loop poles in the left-half plane, providing the damping that P-control alone cannot achieve under the PT₁ model.

4.5 Reactive Controllers

Overview: Two controller variants were developed to handle different sensor availability scenarios. Both use the same underlying velocity-feedback structure.

4.5.1 NavSat Based (GPS Only)

Sensor: GPS receiver from the platform (10 Hz update rate, low positional accuracy).

Control Law: The code computes the error as $\tilde{e}_x = x_{\text{platform}} - x_{\text{drone}}$ (i.e. the vector pointing from drone toward platform), giving a *positive* proportional gain convention. The full control law, including the feedforward term, is:

$$\begin{aligned} \tilde{e}_x &= x_{\text{platform}} - x_{\text{drone}} \\ v_{x,\text{cmd}} &= \underbrace{-K_D \cdot v_{x,\text{drone}}}_{\text{damping}} + \underbrace{K_p \cdot \tilde{e}_x}_{\text{position error}} + \underbrace{K_{ff} \cdot \dot{x}_{\text{platform}}}_{\text{feedforward}} \end{aligned} \quad (12)$$

where: $K_D = 0.5$ (velocity damping baseline), $K_p = 0.6$ (proportional gain on GPS error), and $K_{ff} = 1.0$ (feedforward gain on estimated platform velocity)

The platform velocity $\dot{x}_{\text{platform}}$ is estimated by finite-differencing consecutive GPS fixes:

$$\dot{x}_{\text{platform}} \approx \frac{\Delta x_{\text{platform}}}{\Delta t} \quad (13)$$

The vertical descent uses proportional height control clamped to $[0.08, 0.8]$ m/s.

Gain Tuning: The deployed gains are $K_p = 0.6$, $K_D = 0.5$, $K_{ff} = 1.0$. These were tuned experimentally. The rule of thumb $K_D \approx 2\sqrt{K_p}$ for critical damping would suggest $K_D \approx 1.55$; the value chosen for K_D was found to give faster convergence.

Limitations: Latency from GPS detection causes lag in the error correction mechanism; near trees or buildings there could be multipath interference leading to noisy position estimates, making high-level precision difficult.

4.5.2 ArUco Vision-Based

Design Philosophy: “Use camera for precision when close, ignore GPS noise.”

Sensor: Downward-facing camera detecting ArUco marker on platform (30 Hz, < 0.1 m accuracy when visible).

Perception Pipeline: The ArUco detection node subscribes to the downward camera image, and publishes a `PoseStamped` containing the full 3D pose from PnP (Perspective-n-Point). The controller operates entirely in 3D space.

Control Law (single-stage, 3D pose with normalized coordinates): The 3D marker position in the camera frame (p_x, p_y, p_z) is first transformed to the body frame via a static rotation $\mathbf{R}_{\text{cam}}^{\text{body}}$ and offset $\mathbf{t}_{\text{cam}}^{\text{body}}$:

$$\mathbf{b} = \mathbf{R}_{\text{cam}}^{\text{body}} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \mathbf{t}_{\text{cam}}^{\text{body}} \quad (14)$$

The error is then depth-normalized (dividing by b_z) to get a perspective-corrected signal. This effectively produces a quantity similar to pixel coordinates but in the body frame:

$$n_x = \frac{b_x}{b_z}, \quad n_y = \frac{b_y}{b_z} \quad (15)$$

The proportional control in body frame plus a marker-velocity feedforward term is:

$$\begin{aligned} v_{x,\text{body}} &= K_p \cdot n_x + K_{ff} \cdot \dot{n}_x \\ v_{y,\text{body}} &= K_p \cdot n_y + K_{ff} \cdot \dot{n}_y \end{aligned} \quad (16)$$

where $K_p = 0.45$, $K_{ff} = 0.8$, and $\dot{n}_x$ is estimated by finite-differencing consecutive marker positions.

The body-frame commands are then rotated to NED frame using the current yaw ψ :

$$\begin{bmatrix} v_{x,\text{NED}} \\ v_{y,\text{NED}} \end{bmatrix} = \begin{bmatrix} \cos \psi & -\sin \psi \\ \sin \psi & \cos \psi \end{bmatrix} \begin{bmatrix} v_{x,\text{body}} \\ v_{y,\text{body}} \end{bmatrix} \quad (17)$$

The result is magnitude-clamped to 1.0 m/s and low-pass filtered. The total ArUco velocity command is assembled identically to the NavSat controller (error correction + damping).

Timeout fallback: If no marker pose is received for 0.5 s, the ArUco contribution drops to zero and only the damping term remains (the drone decelerates and hovers).

4.6 Safety Mechanism Native to Both Controllers

4.6.1 Velocity Clamping

All controllers were implemented with hard limits to prevent aggressive maneuvers. Maximum velocities are clamped, and height dependent velocity terms are scaled down as the drone descends.

4.6.2 Soft Landing Zone

Near the platform i.e. when distance remaining is less than < 0.5 the descend speed is progressively reduced by the following

$$v_{z,\text{cmd}} = \begin{cases} v_{z,\text{cmd}} \cdot \frac{h}{h_{\text{soft}}} & \text{if } h < h_{\text{soft}} \\ v_{z,\text{cmd}} & \text{otherwise} \end{cases} \quad (18)$$

where $h_{\text{soft}} = 0.5$ m is the soft-landing zone threshold.

4.6.3 Timeouts and Fallbacks

If there is a communication error and the GPS or the Camera data stops flowing for 2s and 0.5s respectively, their contribution goes to zero. Similarly, offboard priming is used that publishes zero velocity for 1.0s before initial control starts to prevent jerks.

4.7 Conclusion and Motivation

The above-mentioned architecture and design complexity for the reactive controllers forms the motivation for using Reinforcement Learning based Control, which is to be discussed in detail in the next section.

5 Fundamentals of Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique where an agent learns how to behave by continuous interaction with its environment and collecting scalar feedbacks called rewards[1].

Unlike supervised learning, RL does not rely on labeled training data, instead it learns by trial and error. The agent observes its environment, analyzes its current situation, selects an action and then it receives a scalar reward that signals how good or bad that action was. This paradigm of machine learning was researched and engineered in animal psychology and optimal control theory, and since it has been formalized mathematically through the theory of Markov decision processes [2, 3].

5.1 Agent and Environment Loop

At each discrete time step t the environment presents the agent with an observation of the current state s_t . The agent then chooses an action a_t according to its current strategy and applies that action to the environment. In response the environment transitions to a new state s_{t+1} and emits

a numerical reward r_{t+1} . This cycle repeats until the episode terminates or a maximum number of steps is reached [1]. Sutton and Barto describe this loop as the most elementary abstraction of goal-directed interaction, where every action has consequences and the agent must discover which consequences are desirable by interpreting the reward signal over time [1].

5.2 Markov Decision Processes

The mathematical framework underlying most RL algorithms is the Markov Decision Process (MDP). An MDP is specified by the tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ [3], where $\mathcal{S}$ is the state space containing all situations the agent may encounter, $\mathcal{A}$ is the action space containing all possible actions, $P(s_{t+1} | s_t, a_t)$ is the transition function specifying the probability of reaching state s_{t+1} after executing a_t in s_t , $R(s_t, a_t, s_{t+1})$ is the reward function returning a scalar signal that evaluates the quality of a transition, and $\gamma \in [0, 1)$ is a discount factor that controls how much the agent values future rewards relative to immediate ones [1]. For a UAV the state space typically comprises position, velocity, and target-relative measurements and may be continuous ($\mathcal{S} \subset \mathbb{R}^n$). In continuous control tasks such as commanding velocity, the action space is also continuous ($\mathcal{A} \subset \mathbb{R}^m$).

The defining assumption of an MDP is the Markov property, which states that the transition probability depends only on the current state and action, not on any earlier history. Formally this is expressed as

$$P(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1} | s_t, a_t). \quad (19)$$

This memoryless property makes it possible to solve the decision problem using recursive decomposition, as first described by Bellman in the context of dynamic programming [2].

5.3 Policy

A policy π prescribes how the agent selects actions. A deterministic policy is a mapping $\pi: \mathcal{S} \rightarrow \mathcal{A}$ that returns a single action for each state. A stochastic policy instead returns a probability distribution over actions, written $\pi(a | s)$, representing the likelihood of choosing action a when observing state s [1]. Stochastic policies are beneficial because they allow the agent to explore different actions in similar states, which is essential during the learning phase to avoid converging prematurely to a poor strategy.

5.4 Return and Discounting

The objective of the agent is to maximize the expected cumulative discounted reward, referred to as the return, which is defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \quad (20)$$

Discounting ensures that the sum remains finite for infinite-horizon problems and encodes a preference for obtaining rewards sooner rather than later [1]. In episodic tasks such as a single landing attempt the sum naturally terminates when the episode ends.

5.5 Value Functions

A value function measures the long-term desirability of being in a particular state, or of taking a particular action in a particular state, when following a given policy. The state-value function is defined as the expected return starting from state s and thereafter following π , given by [1]

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]. \quad (21)$$

The action-value function extends this concept by also conditioning on the first action taken, and is written as

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]. \quad (22)$$

Whereas the reward signal conveys immediate feedback, the value function captures long-term consequences. An action with high immediate reward may lead to states of low value, and a prudent agent should avoid it. This distinction between reward and value is why value function estimation lies at the heart of most RL algorithms [1].

5.6 Bellman Equations

Both value functions satisfy recursive relationships known as Bellman equations. For the state-value function under policy π the Bellman expectation equation reads [2]

$$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) [R(s, a, s') + \gamma V^\pi(s')]. \quad (23)$$

This equation decomposes the value of a state into the immediate reward plus the discounted value of successor states, weighted by the policy and transition probabilities. The optimal value functions V^* and Q^* satisfy Bellman optimality equations in which the expectation over the policy is replaced by a maximization, giving

$$V^*(s) = \max_a \sum_{s'} P(s' \mid s, a) [R(s, a, s') + \gamma V^*(s')]. \quad (24)$$

Classical dynamic programming methods solve these equations when the transition model is known. In the model-free setting considered here the agent never has access to P and must estimate value functions from sampled experience [1].

5.7 Policy Gradient Methods

Value-based methods such as Q-learning derive a policy indirectly by first estimating Q^* and then acting greedily with respect to it. However, for problems with continuous action spaces, like commanding three-dimensional velocity, enumerating all actions is impossible, so value-based methods are impractical without discretization [4]. Policy gradient methods take a different route and parameterize the policy directly as $\pi_\theta(a \mid s)$ with parameters θ , then optimize θ to maximize expected return.

5.7.1 Policy Gradient Theorem

The policy gradient theorem, proved by Sutton et al. [5], establishes that the gradient of the expected return with respect to the policy parameters can be written as

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(a | s) Q^{\pi_{\theta}}(s, a)] \quad (25)$$

where $J(\theta) = \mathbb{E}_{\pi_{\theta}}[G_0]$. A remarkable property of this result is that it does not require differentiating the environment’s transition dynamics, allowing the gradient to be estimated purely from sampled trajectories.

5.7.2 REINFORCE

The REINFORCE algorithm [6] implements a Monte Carlo estimate of the policy gradient. After collecting a complete trajectory $\tau = (s_0, a_0, r_1, \dots, s_T)$ the gradient is approximated by

$$\nabla_{\theta} J(\theta) \approx \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t \quad (26)$$

using the sampled return G_t in place of $Q^{\pi_{\theta}}$. While unbiased, this estimator has notoriously high variance and requires many samples before convergence [7].

5.7.3 Actor and Critic Architecture

Actor-critic methods reduce variance by introducing a learned value function (the critic) alongside the policy (the actor) [8]. The critic estimates a baseline, usually $V_{\omega}(s)$, using temporal-difference (TD) learning. The TD error is given by

$$\delta_t = r_t + \gamma V_{\omega}(s_{t+1}) - V_{\omega}(s_t) \quad (27)$$

and the policy gradient then uses δ_t as an advantage estimate, so that

$$\nabla_{\theta} J(\theta) \approx \mathbb{E}[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \delta_t]. \quad (28)$$

This substitution introduces a small bias but drastically lowers variance, leading to faster and more stable learning [9]. All three algorithms employed in this work belong to the actor-critic family.

5.8 Algorithms Employed

For this thesis, we selected three state of the art algorithms. Proximal Policy Optimization (PPO) provides stable on-policy updates, Twin Delayed Deep Deterministic Policy Gradient (TD3) suppresses overestimation bias in the critic, and Soft Actor-Critic (SAC) incorporates entropy regularization for robust exploration. We chose the three to address different practical weaknesses of vanilla policy gradients.

5.8.1 Proximal Policy Optimization (PPO)

Schulman et al. [10] observed that the performance of policy gradient methods is extremely sensitive to the size of gradient updates. A single excessively large update can destroy a well-performing policy. This was addressed by constraining policy updates with a Kullback-Leibler divergence trust region through Trust Region Policy Optimization (TRPO) [11], this was however computationally expensive second-order optimization. A comparable stability was achieved through PPO which uses a simpler first order approach.

Clipped Surrogate Objective To quantify how much a policy has changed over two consecutive updates, PPO introduces a probability ratio $\rho_t(\theta)$, defined as

$$\rho_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)}. \quad (29)$$

When the resulting ratio is one, this corresponds to the two policies being identical, if one deviates, the ratio is no longer equal to one, indicating a policy change. These updates can at times be destructively large, to prevent this the clipped objective function constraints this ratio, and is given by

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(\rho_t(\theta) \hat{A}_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] \quad (30)$$

ϵ or clip range, here is a hyperparameter, that defines the permissible update size for the policy, in the subsequent section it is discussed how we chose this hyperparameter. $\hat{A}_t$ is the estimated advantage, it decides whether an action should be encouraged or discouraged, and by how much, based on how that action performs. The minima operation ensures that the objective is a pessimistic lower bound, which means that the any policy change that could worsen the performance is included, but improvements beyond the clipping range are ignored. [10]. Figure 10 shows the behavior of the clipped objective for positive and negative advantage values. Engstrom et al. [12] showed that this simple mechanism rivals TRPO's constrained optimization in practice.

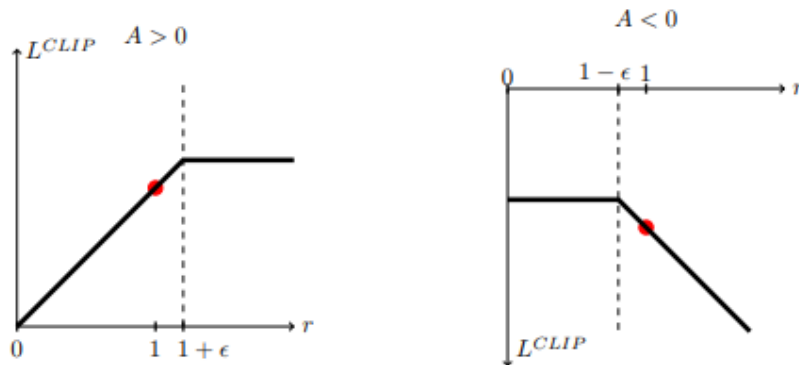


Figure 10: Behavior of the PPO clipped surrogate objective as a function of the probability ratio ρ . When the advantage $\hat{A}$ is positive (left) the objective is clipped at $1 + \epsilon$, preventing excessively optimistic updates. When $\hat{A}$ is negative (right) the objective is clipped at $1 - \epsilon$, preventing the policy from moving too far in the wrong direction. Figure adapted from [10].

Generalized Advantage Estimation PPO uses Generalized Advantage Estimation (GAE) [9] as a means of balancing bias and variance, to calculate how $\hat{A}_t$ is computed. The GAE estimate is defined as

$$\hat{A}_t^{\text{GAE}(\gamma, \lambda)} = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} \quad (31)$$

where $\lambda \in [0, 1]$ interpolates between TD(0) at $\lambda = 0$ (low variance but potentially biased) and Monte Carlo at $\lambda = 1$ (unbiased but high variance).

Training Procedure As PPO is an on policy algorithm, it discards the trained experience after each update. In each iteration a batch of transitions is collected, by executing its current policy, computes the GAE advantages, and then it performs several epochs of minibatch stochastic gradient descent on the clipped objective. Intuitively, as it doesn't have a replay buffer, PPO typically needs to have more environment interactions than off-policy methods, but it is straightforward to implement and tune [10].

5.8.2 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Deep Q-learning with function approximation is known to systematically overestimate action values, a phenomenon analyzed by Thrun and Schwartz [13]. This overestimation bias persists in actor-critic algorithms and can cause divergent training. Fujimoto et al. [15] proposed TD3, which combines three techniques to suppress overestimation. Figure 11 shows how TD3's clipped double Q-learning reduces the gap between estimated and true Q-values compared to standard DDPG.

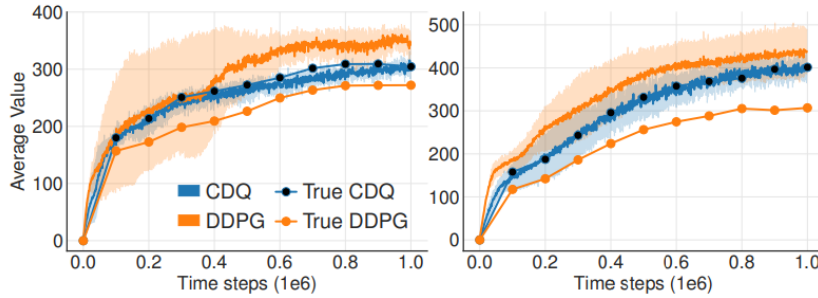


Figure 11: Comparison of estimated Q-values (shaded regions) versus true Q-values (lines with dots) for Clipped Double Q-learning (CDQ, used by TD3) and standard DDPG on two continuous control benchmarks. CDQ consistently produces estimates closer to the true values, confirming that the twin critic mechanism reduces overestimation bias. Figure reproduced from [15].

Clipped Double Q-Learning TD3 maintains two independent Q-networks Q_{θ_1} and Q_{θ_2} , extending the Double Q-learning idea of van Hasselt [14]. For computing the target value the minimum of the two estimates is used, so that the target becomes

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi'}(s')) \quad (32)$$

where θ'_i and ϕ' are target network parameters updated via exponential moving averages. Taking the minimum prevents either network's overestimation from propagating through the target

[15]. The resulting mild underestimation is less harmful because underestimated actions are not preferentially selected by the policy.

Delayed Policy Updates If the critic’s value estimates are noisy, updating the actor at every step amplifies that noise into the policy. TD3 addresses this by updating the policy π_ϕ and target networks only once every d critic updates, with $d = 2$ in the original paper and in this work. This allows the critic to settle toward a more accurate estimate before it informs the actor [15].

Target Policy Smoothing Deterministic policies can exploit narrow peaks in the estimated Q-surface. TD3 regularizes the critic by adding clipped noise to the target action, calculated as

$$\tilde{a} = \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}^2), -c, c) \quad (33)$$

with noise scale $\tilde{\sigma} = 0.2$ and clip bound $c = 0.5$ as used in this implementation. This enforces that nearby actions should have similar Q-values, smoothing out spurious peaks.

Exploration and Reward Clipping Because TD3 outputs deterministic actions, exploration is achieved by adding Gaussian noise $\mathcal{N}(0, \sigma^2)$ with $\sigma = 0.1$ to actions during training. This noise is removed at evaluation time. Additionally, a reward clipping wrapper was applied in this implementation, bounding rewards to $[-5, +5]$ to prevent large gradients from destabilizing the critic networks.

5.8.3 Soft Actor-Critic (SAC)

Sample efficiency and convergence stability remain open challenges for model-free deep RL. Haarnoja et al. [16] proposed Soft Actor-Critic, which augments the standard RL objective with an entropy bonus that encourages the policy to remain stochastic. Figure 12 compares the learning curves of SAC against PPO, DDPG, and TD3 on standard continuous control benchmarks, showing that SAC generally achieves the highest asymptotic return across tasks.

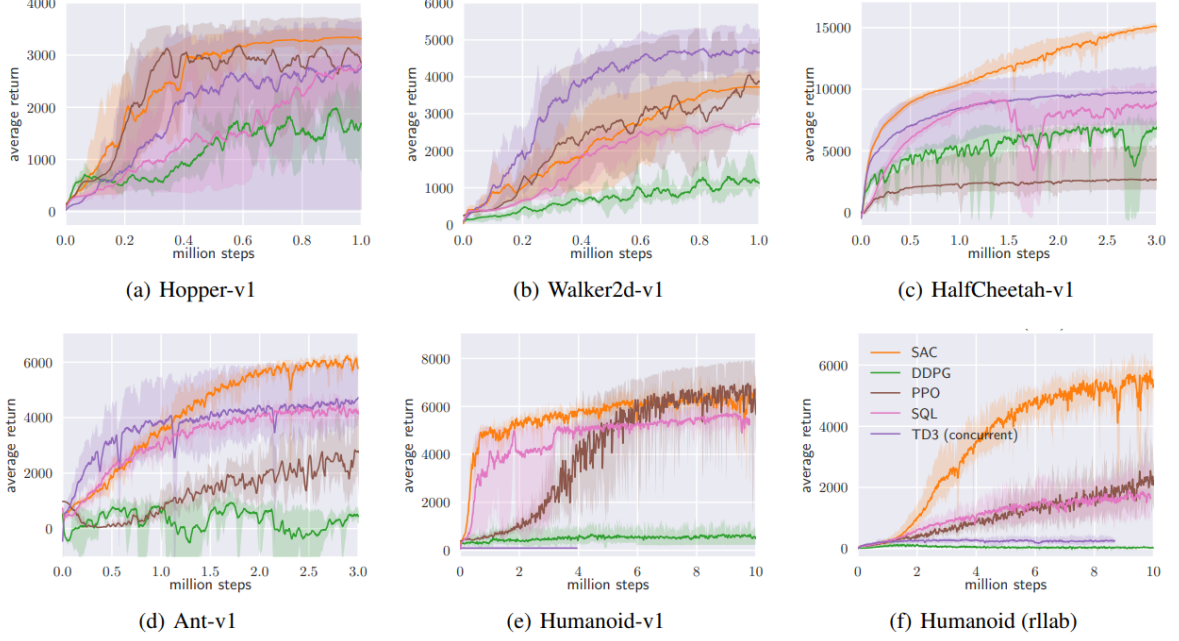


Figure 12: Learning curves for SAC, DDPG, PPO, SQL, and TD3 on six continuous control benchmarks from OpenAI Gym. SAC (orange) achieves the highest or second-highest asymptotic performance across all tasks, demonstrating its sample efficiency and stability in continuous action spaces. Figure reproduced from [16].

Maximum Entropy Objective Rather than maximizing only cumulative reward, SAC maximizes a combined objective that includes an entropy term. Using r for the environment reward (not to be confused with the probability ratio ρ_t used in PPO), this objective is written as

$$J(\pi) = \sum_{t=0}^T \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot | s_t)) \right] \quad (34)$$

where $\mathcal{H}(\pi(\cdot | s)) = -\mathbb{E}_{a \sim \pi} [\log \pi(a | s)]$ is the entropy of the policy at state s and $\alpha > 0$ is a temperature parameter that balances reward maximization against entropy maximization. High entropy discourages premature convergence to a deterministic policy and provides a built-in exploration mechanism [16].

The maximum entropy framework gives SAC a significant advantage over algorithms that rely on external noise for exploration, such as TD3. Because entropy is embedded directly in the optimization objective, SAC explores in a structured way rather than injecting random perturbations. This leads to more stable exploration in high-dimensional continuous action spaces, making SAC more sample-efficient and more robust in noisy environments compared to PPO or DDPG [34].

Soft Q-Function and Policy Update The soft Bellman equation modifies the standard Bellman backup to include the entropy term, giving

$$Q(s, a) = \mathbb{E}_{s'} \left[r(s, a) + \gamma \mathbb{E}_{a' \sim \pi} [Q(s', a') - \alpha \log \pi(a' | s')] \right]. \quad (35)$$

SAC uses two Q-networks to mitigate overestimation, training them by minimizing the soft Bellman residual. A single critic can be overly optimistic in its value estimates, so two independent critics each produce their own Q-value and SAC takes the lower of the two. This clipped double-Q mechanism, shared with TD3, makes the learning signal more conservative and prevents the actor from exploiting overestimated values [16]. The policy is parameterized as a squashed Gaussian, meaning the network outputs a mean and standard deviation, a sample is drawn via the reparameterization trick, and a tanh squashing function maps the output into the bounded action space [16].

Automatic Temperature Tuning Manually tuning α is difficult because the appropriate level of exploration changes as training progresses. Haarnoja et al. [17] reformulated temperature selection as a constrained optimization problem with objective

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi} \left[-\alpha \log \pi(a_t | s_t) - \alpha \bar{\mathcal{H}} \right] \quad (36)$$

where $\bar{\mathcal{H}}$ is a target entropy, conventionally set to $-\dim(\mathcal{A})$. In this work the action space has three dimensions (v_x, v_y, v_z) , so the target entropy was -3 . The temperature α is updated by gradient descent alongside the other network parameters, removing a sensitive hyperparameter from the practitioner’s hands. In Stable-Baselines3 this behavior is activated by setting `ent_coef = "auto"`, which allows the algorithm to learn the appropriate level of exploration autonomously. Early in training the entropy remains high, encouraging broad exploration, and it naturally decreases as the policy becomes more confident in its action selections [34].

On-Policy versus Off-Policy Learning A key structural difference between SAC and PPO is the way each algorithm handles collected experience. PPO is on-policy, meaning it only learns from data generated by its current policy and discards all transitions after each update. SAC is off-policy and stores every transition in a replay buffer, allowing it to sample and learn from the same experience many times [16]. This data recycling makes SAC considerably more sample-efficient: it can achieve comparable or better performance with a fraction of the environment interactions that PPO requires. For the UAV landing task, where each environment step involves a 0.1 s delay for the physics simulation to settle, this improved sample efficiency translates directly into shorter wall-clock training times per curriculum stage [34].

6 Environment Design

This section discusses in detail the custom Gymnasium [18] environment that was developed to train the above-mentioned Reinforcement Learning algorithms for landing the quadrotor on a moving platform. The environment is implemented as a single Python class that inherits both from `gymnasium.Env` and the ROS2 node class `rclpy.Node`. Because a ROS2 node can subscribe and publish to topics regardless of whether it runs inside a Gazebo-coupled process or a standalone training script, the same environment class can interact with the PX4 SITL running in Gazebo without any architectural modifications while simultaneously exposing the standard Gymnasium `step()`/`reset()` interface consumed by RL training libraries.

6.1 State Space

The core state information provided to the RL algorithm at each step is a 9-dimensional vector $s_t \in \mathbb{R}^9$ given by

$$s_t = \begin{bmatrix} e_x & e_y & h & v_{x,d} & v_{y,d} & v_{z,d} & d_{xy} & \hat{v}_{x,p,t} & \hat{v}_{y,p,t} \end{bmatrix}^\top \quad (37)$$

where the components of the vector are as described below:

$e_x, e_y \in \mathbb{R}$ are the position errors (horizontal) in the North-East-Down (NED) frame between the drone and the platform, while $h \in \mathbb{R}$ is the height of the drone above the platform surface. The variables $v_{x,d}, v_{y,d}, v_{z,d} \in \mathbb{R}$ are the velocity components in x, y, and z directions respectively, published via the PX4 odometry messages. The quantity $d_{xy} = \sqrt{e_x^2 + e_y^2}$ is a derived value computed from the core position errors; it is included in the observation vector as an explicit feature to spare the neural network from having to learn the Euclidean norm internally. Finally, $\hat{v}_{x,p,t}, \hat{v}_{y,p,t} \in \mathbb{R}$ are the filtered estimates of the platform velocities in the x and y directions; these are included in the observation so that the neural network policy can learn to anticipate the platform motion during training.

Each component of the state space is normalized to $[-1, 1]$ (or $[0, 1]$ for d_{xy}) by dividing by a domain-specific constant, 50m, 10m/s and 5m/s for the positions, drone velocities and platform velocities respectively. The reason to do so is to mitigate the risk of any single feature from dominating the gradient signal during the training of the neural network. [20].

Platform velocity estimation: As we have the GPS coordinates from the platform (via a `NavSatFix` message at 10 Hz) at a given instance, its velocity must be estimated. Taking difference of successive latitude and longitude values via the ROS2 topic data yields only the raw velocity estimates. These must then be filtered with a first-order moving average exponential with smoothing factor $\alpha = 0.3$ to suppress noise amplification [19], giving

$$\hat{v}_{x,p,t} = 0.3 v_{x,raw,t} + 0.7 \hat{v}_{x,p,t-1}. \quad (38)$$

It is worth mentioning that in the classical controller a pure feedback approach is used, so there is no incorporation of an additional feedforward signal there. In contrast, the RL environment provides the estimated platform velocities as **observational input** to the neural network; the policy therefore **learns on its own, implicitly via the reward signal**, to match the platform motion during training.

6.2 Action Space

The action space is defined as 3-dimensional vector of velocity commands in the NED frame, it can be defined as follows

$$a_t = \begin{bmatrix} v_{x,cmd} \\ v_{y,cmd} \\ v_{z,cmd} \end{bmatrix}, \quad v_{x,cmd} \in [-2, 2], \quad v_{y,cmd} \in [-2, 2], \quad v_{z,cmd} \in [-1, 2] \text{ m/s}. \quad (39)$$

The vertical bounds are intentionally asymmetric: the descent (positive in NED) is limited to 1.0 m/s to provide a safety margin during ground approach, while the upward command (negative in NED) is permitted up to 2.0 m/s to allow the drone to climb away from the platform when needed.

6.2.1 Action Smoothing

Raw neural networks have a tendency to oscillate rapidly between time steps, this can lead to high-frequency vibrations to the airframe [21]. To mitigate this we implemented a first-order exponential moving average to actions before they are executed. This is done as follows

$$\tilde{a}_t = \alpha \tilde{a}_{t-1} + (1 - \alpha) a_t, \quad \alpha = 0.7. \quad (40)$$

The filter is designed so as to assign 70 % weight to the previous command and 30 % to the new output, this smoothens the jerky transitions while preserving responsiveness.

6.2.2 Height-Dependent Velocity Limiting

As the drone approaches the platform ($h < h_{\text{cap}} = 2.0 \text{ m}$), the velocities are progressively capped to allow for a gentle touchdown. The maximum permissible velocity is directly proportional to the height according to

$$v_{\text{max}}(h) = \begin{cases} v_{\text{min}} + \frac{h}{h_{\text{cap}}} (v_{\text{nom}} - v_{\text{min}}), & \text{if } h \leq h_{\text{cap}}, \\ v_{\text{nom}}, & \text{if } h > h_{\text{cap}}, \end{cases} \quad (41)$$

with $v_{\text{min}} = 0.5 \text{ m/s}$, $v_{\text{nom}} = 2.0 \text{ m/s}$, and $h_{\text{cap}} = 2.0 \text{ m}$. Note that for $h > h_{\text{cap}}$ no limiting is applied and the full nominal velocity is available. Below h_{cap} the velocity cap decreases linearly with altitude; additionally, the descent velocity is hard-clamped at 0.3 m/s below this height. This approach plays a role similar to the control barrier function [22] but it is implemented as a hard clamp instead of an optimization based constraint.

6.3 Reward Function

Drafting the reward function for the drone landing comes with uncertainties that are hard to conceive before training and testing the model. The method that worked the best in regard to the thesis was carefully examining results from TensorBoard and analyzing the simulations closely to examine what was missing and what could be improved over multiple iterations. In the beginning we started with a reward function that minimized the landing error, focused on a smooth landing and smaller terminal rewards. Throughout testing and training the reward function was iteratively refined; new reward terms (r_{desc} , r_{hover} , r_{track} , etc.) were introduced to better guide the agent toward the desired landing behavior.

One method that remained unexplored and of interest was Inverse Reinforcement Learning (IRL) to get rewards. Instead of providing the reward function one provides a perfect landing scenario and the algorithm infers the rewards from those examples.

Careful reward engineering is crucial for shaping the learned behavior. Ng et al. [25] showed that potential-based shaping preserves optimal policies while accelerating learning. The reward at each step is the sum of several components, expressed as

$$r_t = r_{\text{dist}} + r_{\text{height}} + r_{\text{vel}} + r_{\text{desc}} + r_{\text{hover}} + r_{\text{smooth}} + r_{\text{track}} + r_{\text{slow}} + r_{\text{prox}} + r_{\text{term}}. \quad (42)$$

6.3.1 Dense Shaping Terms

These rewards provide feedback to the model at every time step and are important in guiding the agent to convergence.

- **Distance penalty** (r_{dist}) encourages horizontal centering: $r_{\text{dist}} = -0.1 \cdot d_{xy}$.
- **Height penalty** (r_{height}) promotes controlled descent: $r_{\text{height}} = -0.1 \cdot |h|$.
- **Velocity penalty** (r_{vel}) discourages excessive speed: $r_{\text{vel}} = -0.01 \cdot \|\tilde{a}_t\|_2$.
- **Smoothness penalty** (r_{smooth}) penalizes jerky control outputs: $r_{\text{smooth}} = -0.05 \cdot \|\tilde{a}_t - \tilde{a}_{t-1}\|_2$.
- **Velocity tracking penalty** (r_{track}) encourages the agent to match its command velocity to the estimated platform velocity: $r_{\text{track}} = -0.05 (|v_{x,\text{cmd}} - \hat{v}_{x,p,t}| + |v_{y,\text{cmd}} - \hat{v}_{y,p,t}|)$.
- **Slow approach bonus** (r_{slow}): when the drone is within 2.0 m altitude of the platform and its total velocity is below 0.5 m/s, a bonus of $r_{\text{slow}} = +0.1$ is awarded to encourage a stable final approach.
- **Descent reward** (r_{desc}): defined in Equation (43).
- **Hovering penalty** (r_{hover}): defined in Equation (45).
- **Proximity bonuses** (r_{prox}): defined in Equation (46).
- **Terminal rewards** (r_{term}): defined in Equation (48).

6.4 The Mathematics of the Hovering Exploit

As discussed in the reward function section, achieving a good reward function is a question of how well one perceives the environment and the final goal. This is not very intuitive over the first iteration and requires one to go over the reward structure many a times. As an example, during the training and testing we observed that the drone had learnt to hover over the platform to collect rewards rather than prioritizing landing. The subsequent section discusses how one overcame the hovering encountered initially by modifying the reward function to fit to the environment and the final goal. After testing our initial PPO setup trained with the metrics mentioned below, we observed that the drone never lands. It continues to hover at $h \approx 0.3$ m and distance $d_{xy} \approx 0.15$ m until the the maximum episode length is reached (500 steps) and is thereafter truncated. The metrics mentioned in the following table explain why this happens.

Component	Condition / Formula	Value
r_{dist} (Distance Penalty)	$-0.1 \cdot d_{xy}$	-0.015
r_{height} (Height Penalty)	$-0.1 \cdot h$	-0.030
r_{vel} (Velocity Penalty)	$-0.01 \cdot v $	≈ -0.002
r_{smooth} (Action Smoothness)	(Smooth hovering)	≈ 0
r_{track} (Velocity Matching)	(Static platform)	≈ 0
r_{slow} (Slow Approach Bonus)	$h < 2.0 \wedge v < 0.5$	$+0.100$
r_{prox} (Proximity Bonus)	$d_{xy} < 0.2$	$+0.500$
$r_{\text{height_bonus}}$ (Height Bonus)	$0 < h < 0.5$	$+0.200$
Total Per Step		$\approx +0.75$

Table 1: Per-step reward breakdown during hovering at $h \approx 0.3$ m, $d_{xy} \approx 0.15$ m over a static platform.

Scenario A: Hovering (The Exploit) – As we have set the episode length to 500 steps, running the full episode without termination leads to the following cumulative return $R_t = \sum r_t$

$$R_{\text{total}} \approx 0.75 \cdot 500 \text{ steps} \approx \mathbf{375}$$

Scenario B: Landing (Intended Behavior) – This is the desired outcome. Assuming a descent over a static platform that terminates at step ~ 80 with $d_{xy} < 0.1$ m:

$$R_{\text{descent}} \approx 80 \text{ steps} \cdot 0.3 \text{ (avg)} = 24$$

$$R_{\text{terminal}} = +10$$

$$R_{\text{total}} \approx 24 + 10 = \mathbf{34}$$

Conclusion: Hovering near the platform for 500 episode steps yields a roughly ten times higher return than landing. The agent is maximizing the reward function provided.

Analyzing the tensorboard output below, one can notice that the behavior shift occurs at ~ 650 K step. The reasoning behind this can be explained as follows:

0–200k Steps: The agent learns the basics. The episodes end via failure/crash or accidental landing. Here the episode length decreases significantly from 400+ steps to under a 100 steps which is the outcome the setup aims for.

200k–650k Steps (Local Optimum): The policy descends and lands. The value function $V(s)$ estimates returns based on these landing outcomes ($R \approx 20 - 40$). The policy persists because $V(s)$ does not yet know that high rewards exist elsewhere. The mean episode length and the mean reward are almost constant as one expects.

~ 650 k Steps (Discovery): Through stochastic exploration, some rollouts miss the landing threshold (< 0.08 m) but remain close. These rollouts run for 500 steps, collecting massive shaped rewards (200+). The policy realizes that it can collect more rewards by staying close to the platform, but never landing on it. The gradient signal flips.

650k–1M Steps (Convergence): Once $V(s)$ updates to reflect hovering returns (~ 375), the episode length shoots up again near the 500 mark, meaning that the episode never ends, the policy rapidly converges to hovering. This is a **phase transition** crossing a saddle point in

policy space.

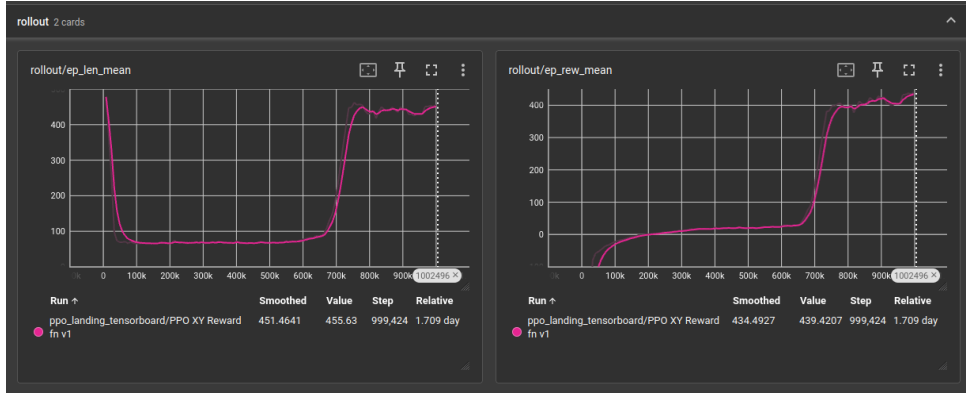


Figure 13: TensorBoard mean episode length (left) and mean episode return (right) from PPO training with the initial reward function that inadvertently favored hovering.

6.5 Applied Fixes

The following changes to the `landing_env.py` aimed at correcting the reward topography.

Descent Reward: An additional reward of $+0.5 \cdot v_z$ was added when $d_{xy} < 0.5$ to have effect on the policy as it descends, in addition to the height penalty as discussed in the table above.

Hovering Penalty: To minimize the hovering, a -0.3 per step penalty was given when $d_{xy} < 0.5$, $h > 0.2$, and $|v_z| < 0.15$.

Terminal Rewards (5× Increase): The terminal rewards were increased fivefold from $+10$ to $+50$ for a perfect landing and from $+5$ to $+30$ for an excellent landing.

Truncation Penalty: As the agent had learnt maximizing its reward in the previous iteration by hovering 500 steps over the platform and not getting any penalty for that, a -3.0 penalty was assigned to discourage that behavior.

Relaxed Threshold: Landing height threshold increased $0.08m$ to $0.12m$.

6.6 New Mathematical Outlook

Landing: ~ 100 steps ≈ 80 (accumulated) + 50 (terminal) = **130+**

Hovering: 0.45 (new step reward) $\cdot 500 - 3.0$ (truncation) = **222**

Note: While hovering is still numerically high, the gap is significantly closed, and the massive spike of $+50$ for landing creates a strong gradient pull that should snap the agent out of the hovering local optimum.

6.7 Analysis: Updated Reward Function

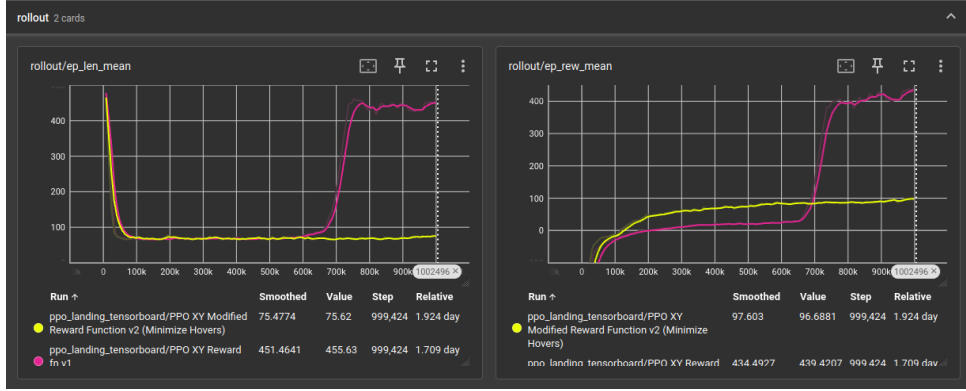


Figure 14: TensorBoard mean episode length (left) and mean episode return (right) from PPO training with the updated reward structure (in yellow) compared to the previous reward structure (in red).

Analyzing the tensorboard above we notice that the episode length of the modified reward function settles at step ~ 70 , hence supporting our hypothesis that adding a descend reward, hovering penalty, truncation penalty and increasing the terminal rewards leads to the intended landing, and flouts the hovering behavior.

6.7.1 Descent and Hovering Rewards

To counteract hovering behavior observed during initial training, explicit descent incentives were introduced. The descent reward is defined as

$$r_{\text{desc}} = \begin{cases} +0.5 \cdot \min(v_{z,\text{actual}}, 1) & \text{if } v_{z,\text{actual}} > 0 \text{ and } d_{xy} < 0.5 \text{ m,} \\ +0.2 \cdot \min(v_{z,\text{actual}}, 1) & \text{if } v_{z,\text{actual}} > 0 \text{ and } d_{xy} \geq 0.5 \text{ m,} \\ 0 & \text{otherwise,} \end{cases} \quad (43)$$

where $v_{z,\text{actual}} > 0$ indicates descent in the NED frame.

An ascending penalty is applied when the drone climbs while close to the landing surface:

$$r_{\text{asc}} = \begin{cases} -0.3 \cdot |v_{z,\text{actual}}| & \text{if } v_{z,\text{actual}} < 0 \text{ and } h < 2.0 \text{ m,} \\ 0 & \text{otherwise.} \end{cases} \quad (44)$$

A hovering penalty discourages stalling behavior near the platform without completing the landing:

$$r_{\text{hover}} = \begin{cases} -0.3 & \text{if } d_{xy} < 0.5 \text{ m, } h > 0.2 \text{ m, and } |v_{z,\text{actual}}| < 0.15 \text{ m/s,} \\ 0 & \text{otherwise.} \end{cases} \quad (45)$$

6.7.2 Proximity Bonuses

Distance-based bonuses create a potential field that draws the agent progressively closer, defined as

$$r_{\text{prox}} = \begin{cases} +1.0 & \text{if } d_{xy} \leq 0.1 \text{ m,} \\ +0.5 & \text{if } 0.1 < d_{xy} \leq 0.2 \text{ m,} \\ +0.2 & \text{if } 0.2 < d_{xy} \leq 0.5 \text{ m,} \\ +0.1 & \text{if } 0.5 < d_{xy} \leq 1.0 \text{ m,} \\ 0 & \text{otherwise.} \end{cases} \quad (46)$$

Height-based bonuses similarly reward low altitudes:

$$r_{\text{height_bonus}} = \begin{cases} +0.2 & \text{if } 0 \leq h \leq 0.5 \text{ m,} \\ +0.1 & \text{if } 0.5 < h \leq 1.0 \text{ m,} \\ 0 & \text{otherwise.} \end{cases} \quad (47)$$

6.7.3 Terminal Rewards

Sparse terminal rewards at the end of an episode provide large signals for especially good or bad outcomes. They are assigned as follows:

$$r_{\text{term}} = \text{one of the following, depending on the termination condition:} \quad (48)$$

- +50.0: landed with $d_{xy} < 0.1 \text{ m}$ (bullseye),
- +30.0: landed with $d_{xy} < 0.2 \text{ m}$ (excellent),
- +15.0: landed with $d_{xy} < 0.5 \text{ m}$ (good),
- +5.0: landed with $d_{xy} < 1.0 \text{ m}$ (acceptable),
- -1.0: landed with $1.0 \leq d_{xy} \leq 3.0 \text{ m}$,
- -7.5: landed with $d_{xy} > 3.0 \text{ m}$ (missed platform),
- -10.0: drifted beyond $d_{xy} > 15.0 \text{ m}$ (out of bounds),
- -5.0: exceeded $h > 15.0 \text{ m}$ or crashed below $h < -0.5 \text{ m}$,
- -3.0: episode timeout (exceeded 500 steps).

6.8 Episode Termination

An episode ends under one of the following four conditions:

- **Successful landing:** the height of the drone above the platform satisfies $-0.1 \leq h \leq 0.12 \text{ m}$. The thresholds account for sensor offset when the drone lands on the platform surface.

- **Horizontal boundary violation:** d_{xy} exceeds 15 m, meaning the drone is far off the platform center.
- **Vertical boundary violation:** h exceeds 15 m above the platform or drops below -0.5 m, indicating a crash through or past the platform.
- **Timeout:** the step count exceeds 500, and the episode is truncated, corresponding to 50 seconds at the 10 Hz control rate.

6.9 Reset Procedure

An apt training methodology that was thought of was to begin every episode with the same initial conditions, the scripted sequence serves as a basis for the training to converge faster to a result helping in debugging and also comparing different RL strategies and algorithms.

1. The environment model waits for the GPS/ArUco position data from both the drone and the platform.
2. A uniformly sampled target altitude z_{target} from $[9, 11]$ m is introduced to allow for variance between episodes.
3. If the drone is below a safe altitude of 5 m, it executes an emergency climb at 3 m/s to gain altitude and avoid collision with the platform or the ground. Additionally, if the drone is horizontally close to the platform ($d_{xy} < 2$ m) during this low-altitude phase, it simultaneously applies a lateral evasive velocity directed away from the platform center to prevent collision during the ascent.
4. Once above the safe altitude the drone centers itself above the platform using proportional position control and climbs or descends to the target altitude.
5. The reset terminates when the drone is within 0.5 m of z_{target} and within 1.0 m horizontally of the platform, or when a timeout of 500 cycles (50 s) is reached.
6. The action smoothing buffer is cleared to prevent leftover commands from the previous episode from influencing the new one.

This procedure ensures that every training episode begins from a consistent and safe initial condition, which is important for stable policy gradient estimation [30].

6.10 ROS 2 and PX4 Integration

The environment class simultaneously inherits from `gymnasium.Env` and `rclpy.Node`. This dual inheritance allows it to function as a standard Gymnasium environment for training with Stable-Baselines3 while also participating in the ROS 2 communication graph to exchange data with the PX4 autopilot. The ROS 2 node is kept alive for the entire duration of the training process; it is created once during environment instantiation and destroyed only when the training script exits. State information is received through three subscriptions.

- `/fmu/out/vehicle_odometry` (`VehicleOdometry`) provides position and velocity at the autopilot’s internal rate of approximately 50 Hz, subscribed with sensor data QoS.
- `/fmu/out/vehicle_global_position` (`VehicleGlobalPosition`) provides geodetic coordinates (latitude, longitude, altitude) of the drone.
- `/platform/navsat` (`NavSatFix`) provides the platform’s GPS coordinates at 10 Hz, subscribed with reliable QoS at queue depth 10.

Actions are sent through three publishers.

- `/fmu/in/trajectory_setpoint` (`TrajectorySetpoint`) publishes velocity commands, with position fields set to NaN to signal velocity-control mode to PX4.
- `/fmu/in/offboard_control_mode` (`OffboardControlMode`) is published every step with the velocity field enabled and all others disabled, maintaining offboard mode.
- `/fmu/in/vehicle_command` (`VehicleCommand`) is used to arm the vehicle and set it to offboard mode during the reset phase.

Standard RL libraries expect synchronous environment stepping, where calling `step()` blocks until the transition is complete. ROS 2, by contrast, processes incoming messages asynchronously through callbacks. This conflict is resolved by running `rclpy.spin()` in a background daemon thread, so that subscriber callbacks continuously update the environment’s state variables while the main thread executes the synchronous training loop. A 0.1 s sleep in the `step()` method acts as a rate limiter, establishing a 10 Hz control frequency [29]. The platform GPS topic is published at 10 Hz by the Gazebo bridge, matching the environment control rate. The drone odometry is available at approximately 50 Hz from PX4, but only the most recent sample is used at each 10 Hz control step.

7 Control Barrier Functions for Safe Reinforcement Learning

For a physical system like a UAV the exploration phase of training an RL policy is random, this poses some concrete risks: the drone may slam into the ground while training or fly outside the landing zone. The reward function can shape the learning to steer away from these situations over time but it is not guaranteed that an unsafe action will not be executed before the agent has learned avoid it. Control Barrier Functions (CBFs) address this problem perfectly, as they filter every action command before it is fed into the vehicle control loop, ensuring that specific safety constraints, like the ones mentioned above, never get violated no matter what the action command is outputted from the learning algorithms [22, 23].

This section highlights the development of a Control Barrier Function (CBF) safety layer which was paired with PPO and SAC to provide safety guarantees during training. The platform boundary and ground collision avoidance barrier functions were implemented. We then introduced the CBF intervention rate to serve as a diagnostic measure for assessing how well the policy has internalized the constraints introduced by the barrier functions.

7.1 Theoretical Background

7.1.1 Safe Sets

Consider a system whose dynamics can be written in the state-space form as

$$\dot{\mathbf{x}} = f(\mathbf{x}) + g(\mathbf{x}) \mathbf{u} \quad (49)$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\mathbf{u} \in \mathcal{U} \subset \mathbb{R}^m$ is the control input, f represents the system dynamics, and g maps the input into the state space. A safe set $\mathcal{C}$ is defined as the super-level set of a continuously differentiable function $h: \mathbb{R}^n \rightarrow \mathbb{R}$ [22]

$$\mathcal{C} = \{\mathbf{x} \in \mathbb{R}^n : h(\mathbf{x}) \geq 0\}. \quad (50)$$

If $h(\mathbf{x}) > 0$ the system is strictly inside the safe region. If $h(\mathbf{x}) = 0$ it is on the boundary. If $h(\mathbf{x}) < 0$ it has left the safe region.

7.1.2 The CBF Condition

A function h can be qualified as a Control Barrier Function if there exists an extended class- $\mathcal{K}$ function α such that for all states in the safe set

$$\sup_{\mathbf{u} \in \mathcal{U}} \left[\dot{h}(\mathbf{x}, \mathbf{u}) + \alpha(h(\mathbf{x})) \right] \geq 0. \quad (51)$$

Choosing a linear class- $\mathcal{K}$ function $\alpha(h) = \alpha \cdot h$ with $\alpha > 0$ gives the working constraint used in this implementation

$$\dot{h}(\mathbf{x}, \mathbf{u}) + \alpha \cdot h(\mathbf{x}) \geq 0. \quad (52)$$

At any given timestep the barrier filter checks if the action fed by the neural network policy satisfies (52). If it does then the action passes through unchanged, if not then the filter projects the action on the nearest one that satisfies the inequality [23].

The physical interpretation is straightforward. When the state is far from the safe set boundary (h is large) the right-hand term $\alpha \cdot h$ provides a generous margin, so almost any action is acceptable. As the state moves towards the boundary (h shrinks toward zero) the margin tightens and the filter proportionally restricts how fast the state may approach the boundary. Exactly at the boundary ($h = 0$) the constraint demands $\dot{h} \geq 0$, meaning the system must either move back inward or remain on it. The gain α determines how early the filter begins to intervene: larger values cause earlier, more conservative corrections [22].

7.1.3 Integration with RL

In the thesis, the CBF architecture sits between the RL policy and the plant. At a given step, the policy produces an action command $\mathbf{u}_{\text{rl}}$, the CBF checks it against the safety constraints, modifies any component that would violate the constraint(s), and passes the modified resulting command $\mathbf{u}_{\text{safe}}$ to the drone.

This separation means the RL algorithm does not need any architectural changes and is free

to explore, while the filter guarantees that exploration never produces a physically dangerous outcome [24]. Over time, as the learning progresses the agent learns from these modifications. The actions that get clipped contribute to lower rewards than actions that pass through unmodified, so the policy learns to generate safe commands. We see this in the later section how the CBF Intervention rate decreases with increase in the training time.

7.2 Safety Mechanisms for the Landing Task

The system state for the CBF filter consists of the horizontal position errors (e_x, e_y) , the height h above the platform, and the velocity (v_x, v_y, v_z) in NED. The control input is the velocity command $\mathbf{u} = [v_{x,\text{cmd}}, v_{y,\text{cmd}}, v_{z,\text{cmd}}]^\top$. Two formal barrier functions were designed to address the main safety concerns. These barrier functions (h_1 and h_2) follow the CBF framework described above and their normalised values are included in the extended observation space.

7.2.1 Platform Boundary CBF (h_1)

The drone must stay within a circular landing zone of radius $R = 3.0$ m centred on the platform. The barrier function is

$$h_1(\mathbf{x}) = R^2 - d_{xy}^2 \quad (53)$$

where $d_{xy} = \sqrt{e_x^2 + e_y^2}$. The safe set $h_1 \geq 0$ corresponds to $d_{xy} \leq R$. Differentiating with respect to time and writing the outward radial velocity as $v_{\text{out}} = (e_x v_x + e_y v_y)/d_{xy}$ gives

$$\dot{h}_1 = -2 d_{xy} v_{\text{out}}. \quad (54)$$

Substituting into (52) and solving for the maximum permissible outward velocity produces

$$v_{\text{out}} \leq \frac{\alpha_1 (R^2 - d_{xy}^2)}{2 d_{xy}}. \quad (55)$$

When the proposed action would exceed this limit the filter subtracts the excess along the outward unit vector $\hat{\mathbf{d}} = [e_x/d_{xy}, e_y/d_{xy}]^\top$, effectively projecting the horizontal velocity to satisfy the constraint. This constraint is activated only when $d_{xy} > 0.5 R$ to avoid unnecessary corrections when the drone is well within the landing zone.

7.2.2 Ground Collision CBF (h_2)

The drone must not crash through the platform. The barrier function is

$$h_2(\mathbf{x}) = h - h_{\min} \quad (56)$$

where $h_{\min}$ is the minimum safe height. A crucial design choice was that this constraint must not prevent landing. A naive implementation would restrict all descent near the surface, making the task unsolvable. The two variants handle this differently:

- In the PPO-CBF variant ($h_{\min} = 0.0$ m) the ground constraint activates only when the drone is below platform level ($h < -0.05$ m), at which point the descent velocity is over-

ridden to force a climb. Between -0.05 m and 0.1 m, only excessive descent speeds above 0.5 m/s are capped to 0.4 m/s, allowing a gentle final touchdown without preventing landing.

- In the SAC-CBF variant ($h_{\min} = -0.1$ m) the constraint activates only when the drone is below platform level ($h < 0$). Above the surface the agent can descend freely, which was necessary because SAC’s entropy-driven exploration would otherwise trigger the ground constraint repeatedly and prevent successful landings.

7.3 Algorithm-Specific CBF Variants

Two separate CBF-enhanced environments were developed. They share the same two barrier functions but differ in their parameterisation, reflecting the different exploration behaviours of PPO and SAC.

7.3.1 PPO-CBF Configuration

The PPO-CBF variant pairs the `CBFSafetyFilter` class with the 11-dimensional `LandingEnvironmentCBF`. Table 2 lists the parameters.

Parameter	Value
Platform radius R	3.0 m
Minimum height $h_{\min}$	0.0 m
Maximum landing velocity	0.5 m/s
α_{boundary}	1.0
α_{ground}	2.0
Landing height threshold	0.12 m

Table 2: CBF safety filter parameters for the PPO-CBF variant.

The ground CBF gain $\alpha_{\text{ground}} = 2.0$ is set higher than the boundary gain because ground collision is more dangerous than a boundary incursion. The PPO-CBF agent was trained for 250 000 time steps using the same PPO hyperparameters as the base agent (Table 6), with an explicit network architecture of $[256, 256]$ for both actor and critic. Checkpoints were saved every 10 000 steps.

7.3.2 SAC-CBF Configuration

The SAC-CBF variant uses the `CBFSafetyFilterSAC` class with `LandingEnvironmentSACCBF`. Table 3 lists the parameters.

Parameter	Value
Platform radius R	3.0 m
Minimum height $h_{\min}$	-0.1 m
Maximum landing velocity	0.8 m/s
α_{boundary}	1.0
α_{ground}	2.0
Landing height threshold	0.15 m

Table 3: CBF safety filter parameters for the SAC-CBF variant.

Three differences distinguish the SAC variant. First, $h_{\min} = -0.1$ m means the ground constraint only fires when the drone is below the platform surface, allowing a full touchdown without CBF intervention. This was necessary because SAC’s entropy bonus produces more varied actions near the surface, and the PPO-style ground constraint was found to block landings entirely. Second, the maximum landing velocity is raised to 0.8 m/s to tolerate SAC’s wider action distribution. Third, the landing height threshold is relaxed to 0.15 m, making touchdown detection more forgiving of sensor noise. The SAC-CBF agent was trained for 1 000 000 time steps with the SAC hyperparameters from Table 7, except that the replay buffer was increased to 100 000 transitions to give the critic a broader history of filtered transitions. Checkpoints were saved every 50 000 steps.

7.4 Extended Observation Space

Both CBF environments extend the base nine-dimensional observation to eleven dimensions by appending the two normalised barrier function values

$$s_t^{\text{CBF}} = \left[e_x \quad e_y \quad h \quad v_{x,d} \quad v_{y,d} \quad v_{z,d} \quad d_{xy} \quad v_{x,p} \quad v_{y,p} \quad \tilde{h}_1 \quad \tilde{h}_2 \right]^\top \quad (57)$$

where

$$\tilde{h}_1 = \text{clip}\left(\frac{h_1}{R^2}, -1, 1\right), \quad \tilde{h}_2 = \text{clip}\left(\frac{h_2}{h_{\max}}, -1, 1\right). \quad (58)$$

Including these values lets the policy network perceive how close it is to each constraint boundary. Over time the policy learns to produce actions that satisfy the constraints naturally, reducing the need for CBF intervention.

7.5 CBF Reward Modifications

The reward functions in the CBF environments differ from the base environment (see Section 8 for a detailed comparison). Both variants include a CBF compliance bonus of +0.02 per step when no constraint is active, creating a gradient signal that encourages the policy to produce inherently safe actions. The PPO-CBF variant adds explicit descent rewards ($+0.5 \cdot \min(v_z, 1)$ when close and descending, +0.2 when further away) and an ascending penalty ($-0.3|v_z|$ when ascending below 2.0 m), which were found necessary to counteract the tendency of the ground constraint to slow descent. It also includes a hovering penalty (-0.3 when the drone stalls above the target with $d_{xy} < 0.5$ m, $h > 0.2$ m, and $|v_z| < 0.15$ m/s) and a progressive high-altitude

penalty $(-0.2(h-5))$ when $h > 5$ m). The SAC-CBF variant uses an even stronger descent reward $(+0.8 \cdot \min(v_z, 1))$ when close to the platform, $+0.3$ when further away) along with a hovering penalty (-0.3) when the drone stalls above the target) and a touchdown incentive (up to $+0.25$ when $h < 0.5$ m and d_{xy} is small).

7.6 Intervention Rate as a Training Diagnostic

The CBF intervention rate, defined as the fraction of steps in an episode where at least one constraint was modified, is logged to TensorBoard under `cbf/intervention_rate` by custom callbacks in both training scripts. A typical pattern observed during training is:

- During the first 50 000 steps the intervention rate sits between 10 % and 20 %, which shows that the policies follow random exploration when untrained; this is exactly why we introduced CBF’s in the first place
- Between 50 000 and 150 000 steps the rate falls to 5 to 10 % as the policy begins learning to stay within the safe set.
- Beyond 150 000 steps the rate drops below 5 %, indicating that the policy has largely learned to produce safe actions on its own.

A persistent high intervention rate late in training would suggest that the reward is not adequately discouraging unsafe actions, or that the CBF parameters are too restrictive and the agent cannot find a feasible path to the landing surface.

8 Reward Comparison Across Environment Variants

In addition to the base landing environment described in Section 5, two CBF-augmented environment variants were developed as described in Section 7. The PPO-CBF variant pairs a Control Barrier Function safety filter with PPO, and the SAC-CBF variant pairs a separately tuned safety filter with SAC. Each variant extends the observation space from 9 to 11 dimensions and modifies the reward function to account for the safety layer. This section documents the differences in the reward structures and discusses the implications for comparing training metrics across environments.

8.1 Per-Step Shaping Reward Differences

Table 4 summarises the per-step reward components across the three environment variants.

Component	Base	PPO-CBF	SAC-CBF
Distance penalty	$-0.1 \cdot d_{xy}$	$-0.1 \cdot d_{xy}$	$-0.1 \cdot d_{xy}$
Height penalty	$-0.1 \cdot h $	$-0.1 \cdot h $	$-0.3h$ or $-0.05h$
Velocity penalty	$-0.01 \cdot \ v\ $	$-0.01 \cdot \ v\ $	$-0.01 \cdot \ v\ $
Smoothness penalty	$-0.05 \cdot \Delta a$	$-0.05 \cdot \Delta a$	$-0.02 \cdot \Delta a$
Tracking penalty	$-0.05 \cdot v_{\text{err}}$	$-0.05 \cdot v_{\text{err}}$	$-0.02 \cdot v_{\text{err}}$
Descent reward	$+0.5 \cdot \min(v_z, 1)$	$+0.5 \cdot \min(v_z, 1)$	$+0.8 \cdot \min(v_z, 1)$
Ascending penalty	$-0.3 v_z $ if $h < 2$	$-0.3 v_z $ if $h < 2$	$-0.5 v_z $
High-altitude penalty	none	$-0.2(h - 5)$ if $h > 5$	$-0.2(h - 5)$ if $h > 5$
CBF compliance bonus	none	+0.02	+0.02
Hovering penalty	-0.3	-0.3	-0.3
Touchdown incentive	none	none	up to +0.25
Slow approach bonus	+0.1	+0.1	+0.15
Proximity bonuses	+1.0/ +0.5/ +0.2/ +0.1 (identical)		
Height bonuses	+0.2/ +0.1	+0.2/ +0.1	+0.3/ +0.15/ +0.05

Table 4: Per-step shaping reward components across the three environment variants. The base and PPO-CBF descent rewards are proximity-dependent: the higher coefficient (+0.5) applies when $d_{xy} < 0.5$ m and the lower (+0.2) when the drone is further away. The SAC-CBF descent reward uses +0.8 when close and +0.3 otherwise. The SAC-CBF height penalty applies $-0.3h$ when $d_{xy} < 1.0$ m and $-0.05h$ otherwise. The ascending penalty in the base and PPO-CBF variants is conditional on height ($h < 2$ m), whereas the SAC-CBF variant applies it unconditionally. The hovering penalty is applied in both the base and PPO-CBF environments when the drone is close to the target ($d_{xy} < 0.5$ m) but has near-zero vertical velocity, discouraging stalling behaviour.

All three environments now include a descent reward, though with different coefficients. The base and PPO-CBF environments reward descent at +0.5 when horizontally close and +0.2 otherwise, while the SAC-CBF variant uses stronger coefficients to counteract the tendency of CBF-filtered actions to slow descent excessively. The smoothness and tracking penalty coefficients are reduced from 0.05 to 0.02 in the SAC-CBF variant because the safety filter already constrains high-frequency oscillations and boundary violations, making aggressive penalties on these components redundant. The PPO-CBF variant retains the base coefficients of 0.05 for both smoothness and tracking penalties.

8.2 Terminal Reward Differences

Table 5 compares the terminal reward signals. The base and PPO-CBF environments assign substantially larger bonuses for successful landings than the SAC-CBF variant.

Outcome	Base	PPO-CBF	SAC-CBF
Bullseye ($d_{xy} < 0.1$ m)	+50	+50	+15
Excellent ($d_{xy} < 0.2$ m)	+30	+30	+10
Good ($d_{xy} < 0.5$ m)	+15	+15	+5
Acceptable ($d_{xy} < 1.0$ m)	+5	+5	+2
Missed on platform	-1	-1	-1
Off-platform ($d_{xy} > 3$ m)	-7.5	-7.5	-7.5
Out of bounds	-10	-10	-10
Too high	-5	-5	-5
Crash	-5	-5	-5
Truncation	-3	-3	-2
Landing height threshold	0.12 m	0.12 m	0.15 m

Table 5: Terminal reward signals across the three environment variants. The base and PPO-CBF environments use the same large positive terminal rewards to provide a strong gradient signal for precise landings. The SAC-CBF variant uses smaller terminal bonuses. Negative terminal signals are identical across all three variants except for the truncation penalty, which differs slightly for SAC-CBF. The landing height thresholds differ: the base and PPO-CBF both use 0.12 m, while SAC-CBF uses 0.15 m.

8.3 Implications for Cross-Environment Reward Comparison

Comparing the raw episode reward values directly (`rollout/ep_rew_mean` in TensorBoard) across these three environments is not meaningful, for four reasons.

First and foremost, the reward scales differ. A perfect landing trajectory in the base environment would accumulate a much higher cumulative reward than the SAC-CBF environment because the former awards +50 for a bullseye landing compared to only +15 in SAC-CBF.

Second, the shaping terms differ in magnitude. The base and PPO-CBF environments penalise smoothness and tracking at 0.05 per step, while the SAC-CBF variant uses 0.02. Over a 500-step episode this difference alone shifts the cumulative reward by several points even for identical trajectories.

Third, the CBF variants introduce additional positive terms (CBF compliance bonus, and in SAC-CBF a touchdown incentive and stronger height bonuses) that inflate per-step accumulation with no counterpart in the base environment. Conversely, the base and PPO-CBF environments’ larger terminal rewards dominate at the end of successful episodes.

Fourth, the landing height thresholds differ. The SAC-CBF environment registers a successful landing when the drone is within 0.15 m of the platform, compared to 0.12 m for the base and PPO-CBF environments. A more permissive threshold makes it easier to earn terminal bonuses and produces shorter episodes, further reducing the accumulation of negative shaping terms.

8.4 Metrics That Permit Fair Comparison

Despite the incompatibility of raw reward values, several metrics remain meaningful for cross-environment comparison.

- **Episode length** (`rollout/ep_len_mean`) is directly comparable because it measures the

number of environment steps before termination, which is independent of the reward function. All three environments operate at the same 10 Hz step rate, so episode length reflects how quickly each agent reaches a terminal state.

- **Task-level success metrics**, including landing accuracy (final d_{xy}), success rate (proportion of landings satisfying $d_{xy} < 1.0$ m and $-0.1 \leq h \leq 0.12$ m), and minimum distance achieved during an episode, are defined by physical quantities and are therefore independent of the reward structure.
- **Reward trends within the same environment** are valid for assessing whether an individual agent is improving over the course of training. The absolute reward magnitude is environment-specific, but its trajectory over time reveals learning progress.

9 Training Procedures

9.1 Stable-Baselines3

Stable-Baselines3 (SB3) is an open-source library providing reliable implementations of common model-free deep RL algorithms in PyTorch [31]. Reproducibility is a well-known challenge in deep RL, where small implementation details such as reward normalization, gradient clipping thresholds, or network initialization can affect performance more than the choice of algorithm itself [30]. SB3 addresses this by providing thoroughly benchmarked, unit-tested implementations behind a consistent scikit-learn-inspired API.

In this project SB3 provides the implementations of PPO, SAC, and TD3 used for training. The library handles rollout collection, replay buffer management, gradient computation, target network updates, and TensorBoard logging. One specifies the algorithm, environment, and hyperparameters, and SB3 manages the training loop internally. Model checkpointing is supported through the `CheckpointCallback` class, which periodically saves the model during training to guard against data loss.

9.2 Software Dependencies

Training requires a Python environment containing Stable-Baselines3, PyTorch, Gymnasium, and NumPy. The project specifies the following minimum versions in its requirements file.

```
numpy ≥ 1.20.0           gymnasium ≥ 1.0.0
stable-baselines3 ≥ 2.0.0 torch ≥ 2.0.0
pyyaml ≥ 6.0
```

The packages are installed in a python virtual environment (`venv`) to avoid conflicts with the system Python packages required by ROS2.

9.3 Launching the Training Pipeline

The training pipeline calls for multiple processes running simultaneously. The launch sequence for the processes is described as follows.

1. Initialize Gazebo Simulation:

Navigate into the PX4-Autopilot directory and launch the PX4 SITL (software-in-the-loop) instance. This loads the Gazebo world with the customized moving platform:

```
PX4_GZ_MODEL_POSE=0,0,2 PX4_GZ_WORLD=moving_platform \  
make px4_sitl gz_x500_mono_cam_down
```

2. Start the Micro-XRCE-DDS Agent:

This bridge converts PX4’s internal uORB messages into DDS topics for ROS 2. Navigate into the Micro-XRCE-DDS-Agent/ directory and execute:

```
MicroXRCEAgent udp4 -p 8888
```

3. Start ROS 2 Bridges:

Launch the GPS and ArUco data bridges. Note that these must be running before the environment can receive platform data.

4. Activate Virtual Environment:

Activate the Python virtual environment that contains Stable-Baselines3 and PyTorch.

5. Run Training Script:

Execute the appropriate training script (e.g., `train_ppo.py`, `train_sac.py`, `train_td3.py`, `train_ppo_cbf.py`, or `train_sac_cbf.py`).

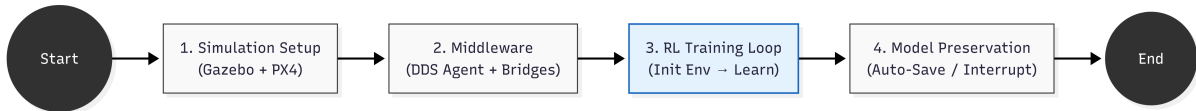


Figure 15: High-level outlook of the training pipeline. The figure abstracts the low-level initialization and can be represented via four primary stages: simulation startup, middleware initiation, core training loop and model checkpointing/saving.

The training script for each RL approach follows a similar structure. First ROS 2 is initialized with `rclpy.init()`. Thereafter the environment is instantiated and a daemon thread is launched running `rclpy.spin()` on the environment node. The SB3 model/agent is created with the desired hyperparameters and `model.learn(total_timesteps=...)` begins training. When training completes the model is saved both with a timestamp and as a “latest” file for convenient testing. If training is interrupted with `Ctrl+C` the partially trained model is saved automatically so that no progress is lost.

9.4 Hyperparameter Selection and Config

9.4.1 PPO Configuration and Selection Rationale

Hyperparameter selection for PPO requires an understanding of how each parameter affects the training dynamics. The following discussion draws on practical guidelines established by the

Stable-Baselines3 Zoo benchmarks and the analysis by Calin [33].

Rollout buffer size (`n_steps`). The hyperparameter `n_steps` defines the time steps the agent collects before it performs a policy update. As we discussed before, this update time has to be large because PPO is an on-policy algorithm and cannot reuse its past experience, therefore it has to be large enough to produce a reliable gradient estimate. The effective rollout size equals $n_steps \times n_envs$. For a single environment the practical recommendation is that the rollout size should be at least 2048 for stable training, with 4096 or more preferred for complex continuous control problems [33]. In this thesis only a single environment is used, so $n_steps = 2048$ provides the minimum stable rollout size. If this rollout buffer was smaller, it would lead to the advantage estimates being noisy and the gradient updates to be unreliable, whereas a substantially larger buffer would mean slower training without proportional benefit for the environment.

Mini-batch size (`batch_size`). The mini-batch size controls the number of transitions used in one gradient update step. The rollout buffer is split into mini-batches and the network parameters are updated once per mini-batch. A common practice is to keep the batch size to approximately one quarter of the rollout size. For the present rollout of 2048 steps this yields $2048/4 \approx 512$, but smaller batches of 64 are also well-suited to environments with relatively short episodes as they reduce the excessive noise and produce more frequent gradient updates. The batch size must also be a divisor of the rollout size so that no transitions are discarded during the splitting. With $2048/64 = 32$ mini-batches per epoch, the gradient receives sufficient data per update while maintaining a fast update tempo.

Number of epochs (`n_epochs`). Number of epochs defines how many complete passes the PPO algorithm makes over the rollout buffer during the update phase. After collecting a rollout of 2048 transitions the algorithm iterates through the entire buffer `n_epochs` times, processing it in mini-batches at each pass. From a fixed set of transitions, more epochs extract more information, but redundant reuse risks the overfitting of the current batch, which leads to a spike in the approx. KL divergence between the old and the new policies. For discrete and relatively simple environments a value of 10 epochs is recommended, whereas continuous control settings with complex dynamics typically use 3 to 5 epochs to prevent the policy from changing too aggressively [33, 10]. Even though we have a continuous action space, a value of 10 was selected in this thesis because the UAV landing environment has short episodes and a well shaped reward function that tolerates more aggressive updates.

Learning rate. The learning rate defines the magnitude of the parameter updates applied to both the actor and the critic at each gradient step. The default value of 3×10^{-4} in Stable-Baselines3 is well-tested across a wide range of environments and represents a practical compromise between convergence speed and stability [33]. For the thesis this default value was retained as the reward scale and observation normalization are within the standard range that the SB3's defaults were benchmarked against. The higher the learning rate the higher the oscillation between two strategies thus increasing the KL divergence, whereas slowing the learning rate results in slower convergence.

Discount factor (γ). The discount factor accounts for the weight of the future rewards compared to the present ones. A value of $\gamma = 0.99$ causes the agent to plan over a horizon of roughly $1/(1 - \gamma) = 100$ time steps. As for the thesis the landing task of the drone which operates at 10 Hz, this translates to 10 seconds of future planning, which is indeed appropriate for an approach and touchdown manoeuvre that typically lasts 10-25 seconds.

GAE parameter (λ). The combination $\gamma = 0.99$ and $\lambda = 0.95$ was formulated in the original GAE paper by Schulman et al. [9] and has since become the standard setting in PPO implementations including the original paper [10], OpenAI Spinning Up, and the SB3 Zoo benchmark suite [33]. Setting $\lambda = 0.95$ is effective as it provides an adequate balance between the low variance but biased TD(0) estimate and the high variance unbiased Monte Carlo estimate. For the thesis and the landing task this was not modified because the episode lengths are moderate, the critic is straightforward to train, so neither of the extreme bias variance trade off is problematic.

Clip range (ϵ). As discussed earlier, the clip range limits how much the policy is allowed to change itself in a single update. The standard value, $\epsilon = 0.2$, introduced in the original PPO paper [10], was used for this work. This prevents large jumps that would cause instability to the drone frame, thus the policy adjusts more meaningfully at each update. We noticed that the reward signals were stable during the training, if it were not the case, reducing ϵ toward 0.1 is a common corrective measure; conversely, if the policy learns too slowly, increasing ϵ toward 0.3 can help. Since this was not the case, the standard value was retained.

Initially the PPO setup was trained for 250K timesteps to gain confidence in the simulation and environment setup that we proposed. Promising results from the initial training, testing and rigorous modification of the reward function allowed us to finally train the PPO for 1,000,000 time steps without curriculum staging. Table 6 lists the hyperparameters used.

As the PPO is an on-policy method, it cannot reuse relative past experience, making curriculum learning counter-intuitive. The PPO model discussed in the evaluation trains on the full dynamic environment from the start.

Parameter	Value
Policy network	MLP [256, 256]
Learning rate	3×10^{-4}
Rollout buffer	2048 steps
Mini-batch size	64
Optimization epochs per rollout	10
Discount factor γ	0.99
GAE parameter λ	0.95
Clip parameter ϵ	0.2
Entropy coefficient	0.01
Total time steps	1 000 000

Table 6: PPO hyperparameters for the UAV landing task.

9.4.2 SAC Configuration and Selection Rationale

The SAC training was divided into two separate regimes, first where we modeled the Gazebo environment in a way that complexity increased over different learning stages. This is also commonly known as Curriculum Learning. Training an agent directly on a complicated setup (hardest version of solving a task) can lead to slow convergence or lead it to being trapped in a poor local optimum. Bengio et al. [27] showed that presenting training examples in order of increasing difficulty can significantly improve both convergence speed and final performance. Soviany et al. [28] later characterized curriculum learning as a continuation method in non-convex optimization, where training begins with a smoothed objective landscape and gradually sharpens it toward the original problem.

For the off-policy algorithm SAC a multi-staged curriculum was implemented, each stage allocating different time steps according to its complexity.

1. **Stage 1, Stationary platform (0 to 150k steps):** the platform is fixed at the origin. The agent learns fundamental descent and stabilization skills without needing to track a moving target.
2. **Stage 2, Single-axis motion (150k to 300k steps):** the platform moves sinusoidally along one axis, introducing the need for velocity tracking.
3. **Stage 3, Two-axis motion (300k to 500k steps):** the platform executes Lissajous curves, requiring simultaneous tracking in two horizontal directions. After Stage 3 completes at 500k steps, the curriculum training ends and the final model is saved.

Curriculum learning is not applicable to PPO because, as an on-policy algorithm, it discards all collected data after each update. There is no replay buffer to carry over transitions from earlier stages. Hence PPO was trained directly on the full dynamic environment for 1,000,000 steps. SAC being off-policy, retains a replay buffer of past transitions. When the curriculum advances, earlier transitions remain in the buffer and continue to be sampled alongside new ones, allowing the agent to benefit from progressive difficulty without losing foundational skills.

It should be noted that the replay buffer is limited to 50 000 transitions while the total curriculum spans 500 000 steps. This means the buffer is overwritten approximately ten times over the course of training. Consequently, by the time Stage 3 begins, transitions from Stage 1 (stationary platform) will have been largely replaced by more recent experiences. The foundational skills learned in earlier stages are therefore retained primarily in the network weights rather than through explicit replay of early transitions. The curriculum approach served as a proof of concept that the model learns progressively from previous stages, as the weights carry forward the learned behaviors even after the corresponding buffer transitions have been overwritten.

In the second regime, the SAC was immediately trained in the full dynamic environment for 1,000,000 steps without any curriculum staging, to compare with the curriculum-trained variant.

Replay buffer size (buffer_size). The replay buffer is the mechanism through which SAC achieves its sample efficiency. A buffer of 50 000 transitions was chosen for the base SAC training. Larger buffers (100k to 1M) are common for complex robotics tasks and allow the agent to

draw from a wider history of experience, stabilizing the Q-value estimates [34]. However, with curriculum learning each stage runs for 250 000 steps, and a buffer that is too large relative to the stage length would retain a disproportionate number of transitions from the early (easier) stages, potentially slowing adaptation. A 50 000-transition buffer provides a reasonable window of recent experience without excessive memory consumption.

Batch size. SAC uses a batch size of 256, which is larger than the typical PPO mini-batch of 64. In off-policy algorithms the batch is sampled from the replay buffer at every gradient step, and larger batches produce more stable critic updates because the Q-value target is computed from a broader sample of transitions [34]. In continuous control environments, where the Q-surface is smooth but high-dimensional, batch sizes of 256 to 512 are standard practice.

Learning rate. The learning rate of 3×10^{-4} was retained from the SB3 default. In SAC the same learning rate is applied to the actor, both critics, and the entropy coefficient, so an inappropriate choice affects the entire agent. This value is tested extensively in the SB3 benchmark suite for continuous control tasks similar to the UAV landing problem [34].

Soft update coefficient (τ). The soft update coefficient controls how quickly the target critic networks track the online critics. At each gradient step the target parameters are updated as $\theta_{\text{target}} \leftarrow \tau \theta_{\text{online}} + (1 - \tau) \theta_{\text{target}}$. A value of $\tau = 0.005$ provides slow, stable tracking. Higher values (0.01 to 0.02) cause faster updates but risk instability, while lower values increase stability at the cost of slower propagation of learning signal to the targets [34].

Train frequency and gradient steps. Both `train_freq` and `gradient_steps` are set to 1, meaning that at every environment step exactly one batch is sampled from the replay buffer and one gradient update is performed on the actor and critics. This produces $n_{\text{updates}} \approx \text{total_timesteps}$ updates over a training run, which is the default and recommended configuration for SAC in SB3 [34].

Discount factor (γ). As with PPO, $\gamma = 0.99$ is used. In continuous control environments with slow dynamics, a high discount factor is standard because the consequences of actions can propagate over hundreds of time steps. A lower discount factor would cause the critics to undervalue long-term approach strategies and focus only on immediate rewards, leading to myopic and unstable policies [34].

Parameter	Value
Policy network	MLP [256, 256]
Learning rate	3×10^{-4}
Replay buffer	50 000 transitions
Learning starts	1000 steps
Batch size	256
Soft update coefficient τ	0.005
Discount factor γ	0.99
Train frequency	1 gradient step per env step
Gradient steps	1
Initial temperature α	auto-tuned
Target entropy $\bar{\mathcal{H}}$	-3
Time steps per stage	250 000

Table 7: SAC hyperparameters with curriculum learning.

9.4.3 TD3 Configuration

TD3 was trained for 1 000 000 steps on the same base landing environment used by PPO and SAC. Table 8 lists the hyperparameters.

Parameter	Value
Policy network (actor and critics)	MLP [256, 256]
Learning rate	3×10^{-4}
Replay buffer	100 000 transitions
Learning starts	1000 steps
Batch size	256
Soft update coefficient τ	0.005
Discount factor γ	0.99
Policy delay d	2
Target policy noise $\tilde{\sigma}$	0.2
Noise clip c	0.5
Exploration noise σ	0.1 (Gaussian)
Total time steps	1 000 000

Table 8: TD3 hyperparameters for the UAV landing task.

10 Monitoring Training with TensorBoard

While the training scripts are being executed, they simultaneously log metrics to TensorBoard [32] via the built-in integration provided by Stable-Baselines3. Every algorithm writes the metrics to its own log directory. (`ppo_landing_tensorboard/`, `sac_landing_tensorboard/`, `td3_landing_tensorboard/`, `ppo_cbf_tensorboard/`, `sac_cbf_tensorboard/`). The logging can be set via the (`log_interval=`) in the respective training scripts. In our case we chose the interval to be 4, providing sufficiently frequent updates without causing excessive overhead.

TensorBoard enables tracking and visualizing scalar metrics throughout training [32]. The primary metrics recorded include the following.

Episode reward (mean and standard deviation) captures the cumulative reward per episode, indicating overall task performance.

Episode length counts the number of steps per episode. Shorter episodes may indicate early termination due to crashes, while very long episodes suggest the agent is not descending to the platform.

Value loss (PPO) or **critic loss** (SAC, TD3) measures how well the value network fits the observed returns.

Policy entropy (PPO, SAC) indicates how stochastic the policy is. Entropy should generally decrease over training as the agent becomes more certain about the best actions.

Learning rate confirms the configured learning rate schedule.

Clip fraction and approximate KL divergence (PPO only) monitor whether the clipping threshold is well-calibrated.

Actor loss (SAC, TD3) tracks the policy improvement signal.

CBF intervention rate and total interventions (CBF variants only) track the fraction of steps where the safety filter modified the action and the absolute count of modifications per episode. To launch TensorBoard one navigates to the directory containing the log folder and runs

```
tensorboard --logdir=./ppo_landing_tensorboard/
```

replacing `ppo_landing` with the appropriate directory name. The web interface, accessible by default at <http://localhost:6006>, provides interactive scalar plots that allow side-by-side comparison of training runs. Figure 16 shows the Scalars dashboard during training, and Figure 17 presents the Time Series view with rollout statistics.



Figure 16: TensorBoard Scalars dashboard showing training metrics for multiple PPO and SAC runs. The panels display actor loss, approximate KL divergence, clip fraction, clip range, critic loss, entropy coefficient, and other diagnostic quantities over the course of training. Comparing these curves across runs helps identify convergence issues and hyperparameter sensitivity. *Note: These figures are included primarily to illustrate the TensorBoard interface and the general shape of the training curves. Fine-grained details may not be fully legible in print; the quantitative evaluation is discussed separately.*

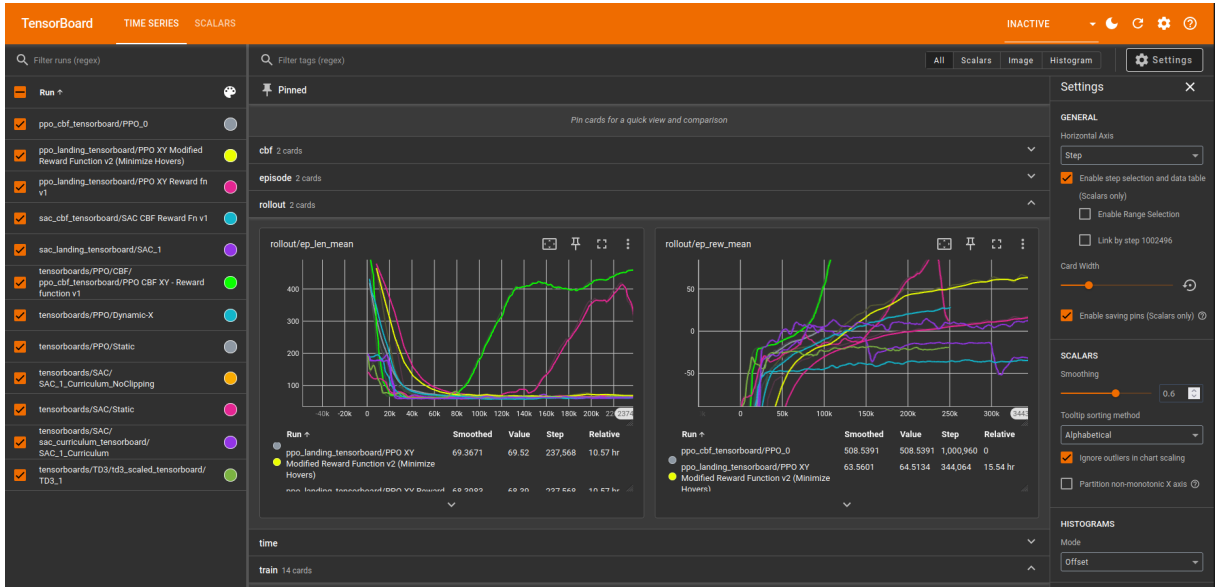


Figure 17: TensorBoard Time Series dashboard showing rollout statistics for different PPO runs (static, dynamic and complex environments), SAC (static and curriculum), and TD3 runs. The left panel displays mean episode length and the right panel shows mean episode reward. The improvement in reward over training steps and the decrease in episode duration indicates that the policies are learning to land more accurately. *Note: As above, these figures serve to illustrate the general training trends; detailed numerical results are reported in the evaluation section.*

11 Comparative Evaluation of Landing Methods

This chapter presents a systematic evaluation and comparison of all seven landing methods developed in this work: two reactive controllers (NavSat, ArUco), three reinforcement learning agents (PPO, SAC, TD3), and two safety-augmented variants (PPO+CBF, SAC+CBF). All methods were evaluated under identical conditions in the Gazebo simulation environment with PX4 SITL, landing on a moving platform (sinusoidal motion with X-amplitude 1 m and Y-amplitude 2 m) at height 0.2 m.

11.1 Evaluation Methodology

11.1.1 Simulation Setup

Each method was evaluated by running autonomous landing episodes with a randomized initial altitude between 9 m to 11 m above ground level. The reinforcement learning methods were evaluated for 100 episodes, while the reactive controllers being deterministic were evaluated for fewer episode, as no stochastic variation between the runs under the same initial condition prevails. All experiments used identical simulation parameters: Gazebo 11 physics engine, PX4 v1.14 autopilot firmware, and the x500 quadrotor model.

11.1.2 Performance Metrics

For each evaluation episode we recorded the following metrics:

- **Success Rate (%)**: Fraction of episodes where the drone touched down within a horizontal distance of 1.0 m from the platform center at or below the landing height threshold.
- **Final Horizontal Distance d_{xy} (m)**: Euclidean distance in the XY-plane between the drone and the platform center at the moment of touchdown.
- **Landing Time (s)**: Time from the start of the descent to touchdown.
- **Steps**: Number of control steps (at 10 Hz) executed per episode.
- **Final Height Above Platform h_f (m)**: Vertical distance between the drone and the platform surface at termination. A value close to 0.0 m indicates a proper touchdown.
- **Landing Quality**: A categorical grade based on the final horizontal distance:
 - *Perfect*: $d_{xy} < 0.1$ m
 - *Excellent*: $0.1 \text{ m} \leq d_{xy} < 0.2$ m
 - *Good*: $0.2 \text{ m} \leq d_{xy} < 0.5$ m
 - *Acceptable*: $0.5 \text{ m} \leq d_{xy} < 1.0$ m
 - *Failed*: $d_{xy} \geq 1.0$ m or episode truncated
- **Cumulative Reward**: Sum of per-step rewards over the episode (applicable to RL methods only).
- **CBF Intervention Rate (%)**: Fraction of steps where the Control Barrier Function modified the agent’s action (applicable to CBF methods only).

11.1.3 Evaluation Protocol

RL methods were chosen based on the based performing model checkpoint (selected based on the training mean reward convergence from the TensorBoard). The model was then loaded and evaluated, the evaluation script recorded the full trajectory at each control step including the position, velocity, height above the platform and its distance to the platform.

The results were then accumulated into a unified JSON file, from which the figures were generated. For reactive control methods, evaluations were run and the data was recorded into the JSON file as well.

11.2 Quantitative Results

11.2.1 Summary of Key Metrics

Figure 18 shows the summary of the evaluation table with the metrics that we discussed above for all the RL, reactive control based and safety augmented methods.

Comparative Evaluation Summary

Method	Success Rate	Final Dist. (m)	Min Dist. (m)	Height (m)	Time (s)	Steps	Quality
PPO	100 %	0.055 +/- 0.030	0.014	0.091	7.7 +/- 1.3	75	Perfect
SAC	100 %	0.207 +/- 0.109	0.068	0.095	11.3 +/- 3.1	110	Good
PPO + CBF	0 %	0.049 +/- 0.021	0.004	0.628	51.1 +/- 0.0	500	Failed
SAC + CBF	95 %	0.429 +/- 0.378	0.244	0.103	8.4 +/- 1.3	81	Good
TD3	85 %	0.607 +/- 0.335	0.125	0.044	8.1 +/- 1.1	79	Good
NavSat	100 %	0.558 +/- 0.000	0.517	-0.223	6.6 +/- 0.0	67	Acceptable
ArUco	100 %	0.259 +/- 0.000	0.073	-0.150	17.6 +/- 0.0	177	Good

Figure 18: Summary metrics table for all evaluated landing methods.

11.2.2 Success Rate Analysis

PPO achieved a perfect 100% success rate across all 100 evaluation episodes, making it the most reliable method tested. SAC as well performed with this with 100% success, though with lower landing precision. The reactive controllers (NavSat, ArUco) also achieved successful landings, albeit evaluated over fewer episodes. SAC+CBF attained 95% success with 5 failed episodes. TD3 achieved an 85% success rate across 100 episodes, with 15 failed episodes where the drone landed beyond the 1.0 m acceptance threshold.

The PPO+CBF variant represents a notable failure case: despite achieving the best horizontal alignment of any method ($\bar{d}_{xy} = 0.049$ m), every episode that we evaluated failed as the drone hovered above the platform until the episode was truncated. The episodes were truncated at the maximum 500 steps (51.1 s), with a mean final height of 0.63 m above the platform. This hovering behavior is analyzed in detail in Section 11.9.

11.2.3 Landing Precision

Figure 19 depicts the mean final distance and mean landing time for each method as a collective bar chart.

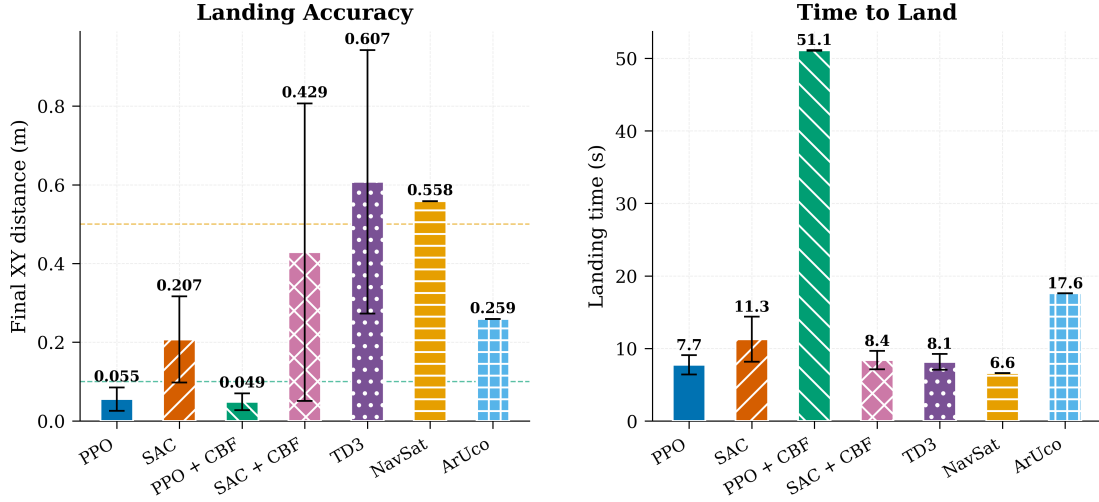


Figure 19: Mean final horizontal distance (left axis) and mean landing time (right axis) for each method. Error bars show one standard deviation.

PPO demonstrated substantially superior landing precision compared to all other methods:

- **PPO:** $\bar{d}_{xy} = 0.055$ m, with 93% of landings rated *Perfect* ($d_{xy} < 0.1$ m)
- **SAC:** $\bar{d}_{xy} = 0.207$ m, approximately 3.8 times less precise than PPO
- **ArUco:** $\bar{d}_{xy} = 0.259$ m, comparable to SAC in accuracy
- **SAC+CBF:** $\bar{d}_{xy} = 0.429$ m
- **NavSat:** $\bar{d}_{xy} = 0.558$ m
- **TD3:** $\bar{d}_{xy} = 0.607 \text{ m} \pm 0.335 \text{ m}$, the least precise of the RL methods, with the majority of landings rated *Good* (44%) or *Acceptable* (36%)

The worst episode that we measured with the PPO, the final distance was 0.122 m, meaning that even its worst landing was rated Excellent.

In contrast, SAC’s worst case was 0.724 m, and SAC+CBF’s worst case was 2.91 m almost violating the CBF boundary condition in this case (this is interesting as the model was trained for 1,000,000 steps - during evaluation the CBF condition was nearly triggered). TD3’s worst case reached 1.758 m, well beyond the acceptance threshold, and even its best landing (0.135 m) was only rated Excellent.

11.2.4 Landing Time and Efficiency

NavSat was the fastest method 6.6 s (67 steps), as we tuned a high gain for a faster descent and traded off with accuracy, followed by PPO at $7.7 \text{ s} \pm 1.3 \text{ s}$ (75.7 steps) and TD3 at $8.1 \text{ s} \pm 1.1 \text{ s}$ (79.9 steps). SAC+CBF followed closely at $8.4 \text{ s} \pm 1.3 \text{ s}$ (81.9 steps). SAC was notably slower at $11.3 \text{ s} \pm 3.1 \text{ s}$ (110 steps), depicting the slight hover during final approach to try to achieve a precise landing. ArUco was the slowest successful method at 17.6 s (177 steps), as during the descent the marker was lost and the drone had to hover until the marker came into detection

again, to proceed with the landing.

The high variance in SAC’s landing time (standard deviation of 3.1 s) indicates inconsistent approach strategies across episodes, whereas PPO and TD3 exhibited tight consistency ($\sigma = 1.3$ s and $\sigma = 1.1$ s respectively).

11.3 Trajectory Analysis

11.3.1 Descent Profiles

Figure 20 shows the height above platform versus time for each method. PPO showed a smooth, balanced descending profile with minimal variance across episodes that depicted a well learned descent policy balancing speed with controlled approach, the SAC algorithm also showed a similar trend.

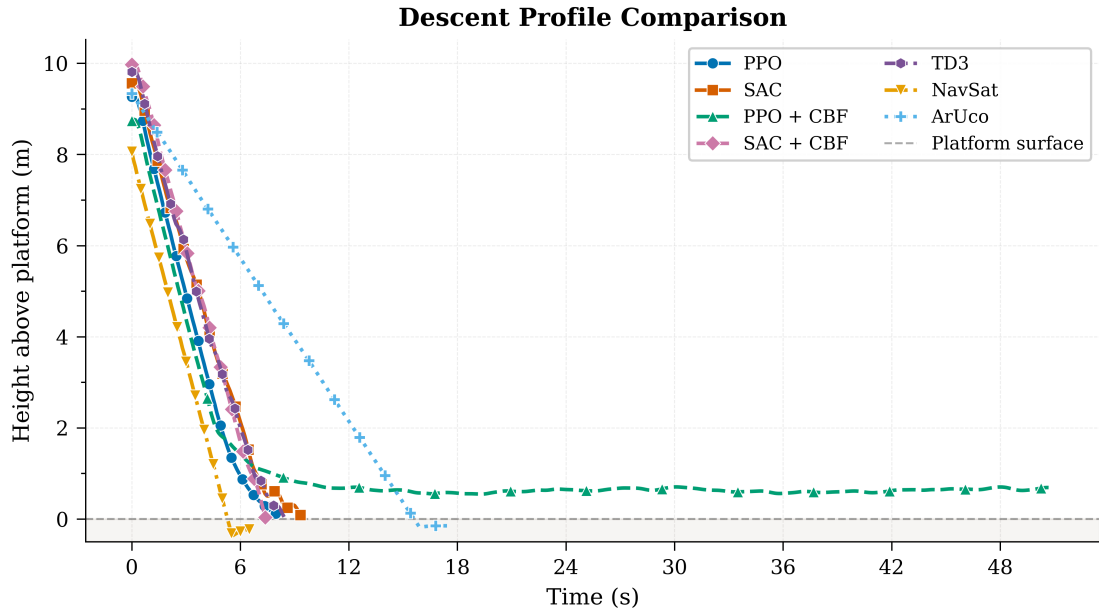


Figure 20: Descent profiles: height above the landing platform versus elapsed time.

The NavSat controller produces a direct, steep descent driven by the proportional-derivative control law, reaching the platform quickly but with limited ability to correct lateral errors during the final approach. TD3 exhibits a descent profile similar to SAC in shape, reaching the platform in approximately 8 s, though with less lateral precision during the final approach. The descent profile for the PPO + CBF method settles at approx. 0.69 m depicting the hovering.

11.3.2 Horizontal Distance Over Time

Figure 21 plots the XY distance from the platform center as a function of time. This reveals how quickly each method converges toward the target horizontally.

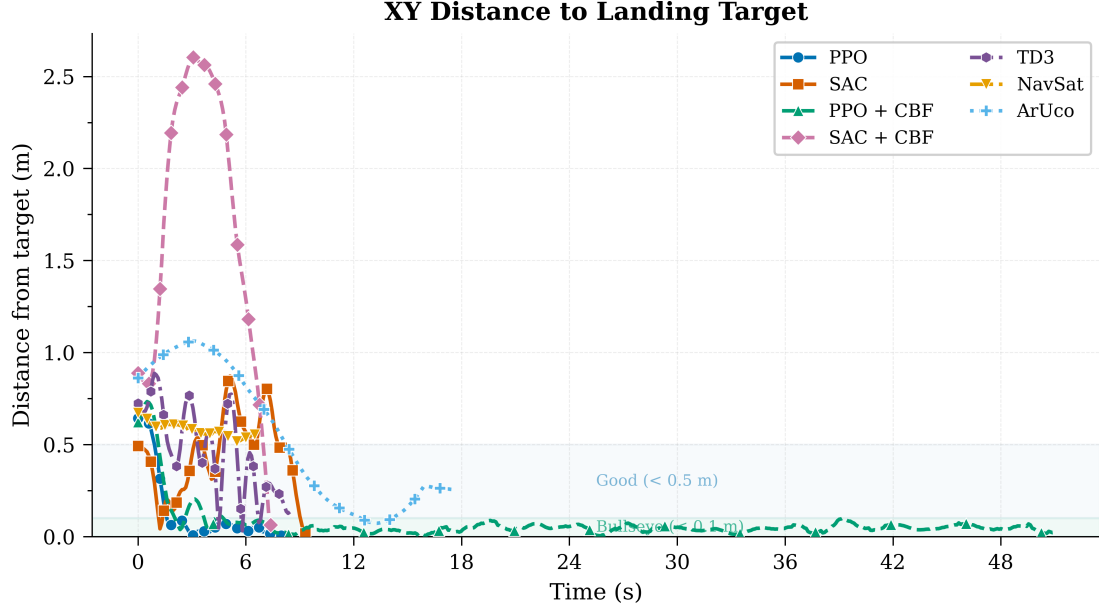


Figure 21: Horizontal distance to the platform center (d_{xy}) versus time. PPO converges most rapidly and maintains the smallest distance throughout the descent.

PPO consistently reduces d_{xy} within the first 3 seconds and maintains sub-0.1 m alignment for the remainder of the episode. SAC converges more slowly, with several episodes maintaining $d_{xy} > 0.3$ m until touchdown. TD3 shows a similar convergence pattern to SAC but maintains a higher residual d_{xy} at touchdown, reflecting its lower landing precision. The SAC+CBF traces reveal occasional spikes where the boundary constraint pushes the drone inward, temporarily increasing the distance before it re-converges.

11.3.3 2D Landing Trajectories

Figure 22 presents bird’s-eye view (XY) trajectories for all methods, showing the horizontal approach paths toward the platform center.

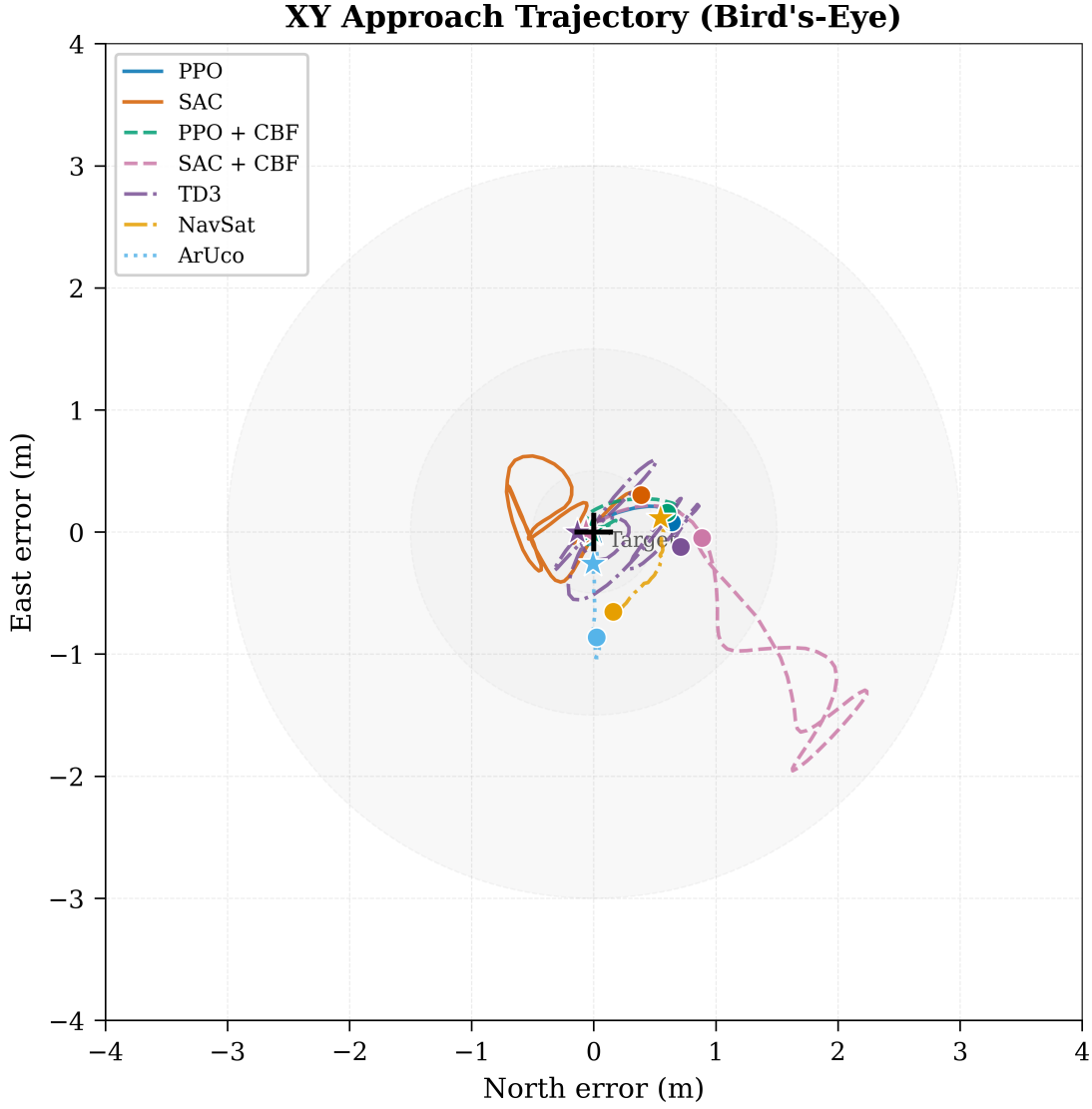


Figure 22: Top-down (XY) landing trajectories. The red circle marks the platform boundary. PPO trajectories converge tightly; SAC and SAC+CBF show wider spread. Reactive methods follow smooth, predictable arcs.

PPO trajectories converge tightly toward the platform center. SAC trajectories exhibit wider spread and occasional looping behavior, indicating suboptimal lateral correction. The reactive controllers follow more predictable, smooth arcs determined by their gain parameters.

11.3.4 3D Flight Paths

Figure 23 shows a three-dimensional view of the flight paths, showing the complete approach geometry including altitude.

3-D Landing Trajectories

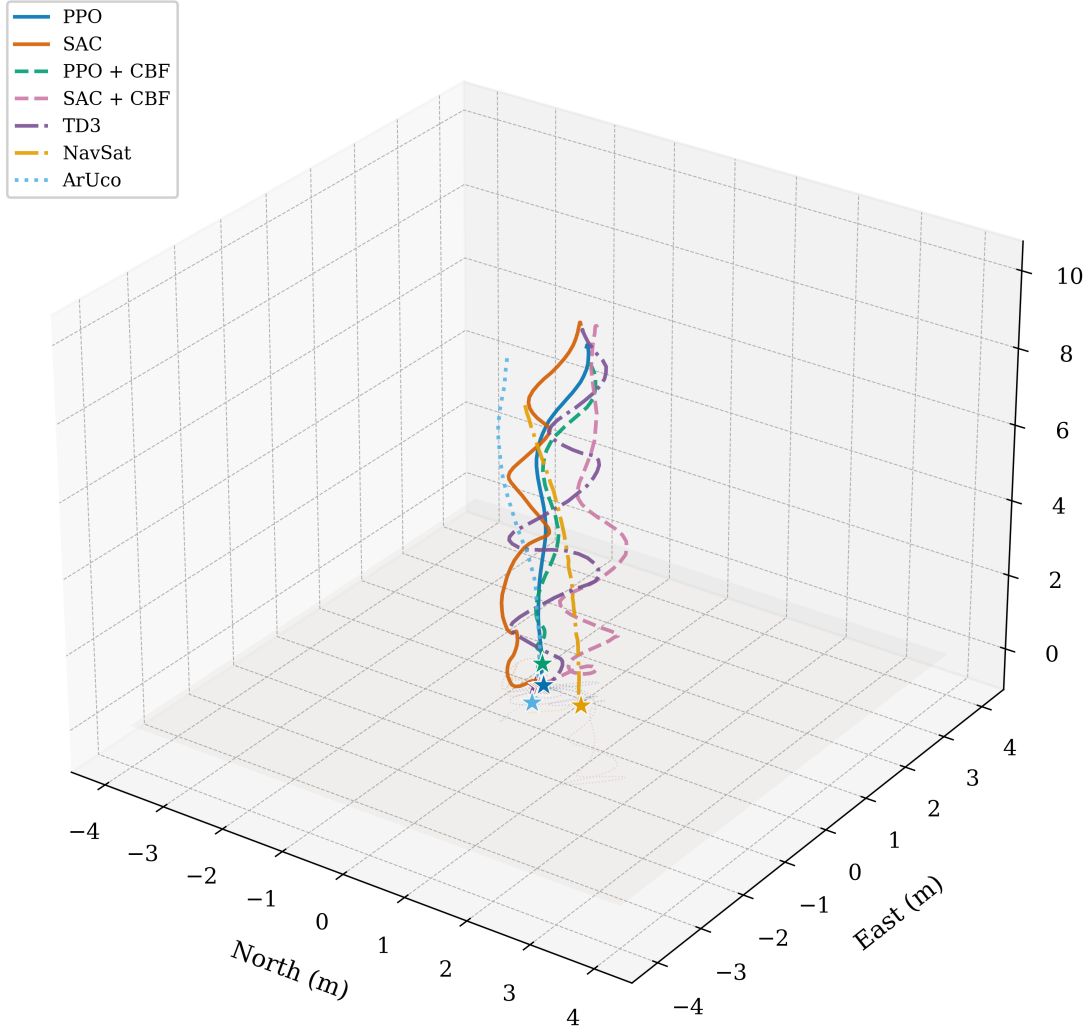


Figure 23: 3D flight trajectories for all methods. PPO shows a nearly vertical descent with tight lateral convergence. SAC and SAC + CBF dynamic approach (aimed at entropy maximization) is visible.

The 3D visualization really shows that the PPO learns a nearly vertical descent strategy once aligned, while SAC follows more diagonal paths.

11.3.5 Mean Trajectory with Confidence Bands

Figure 24 depicts the mean descent profile with $\pm 1\sigma$ bands for each RL method, computed over normalized time to account for episode length variation.

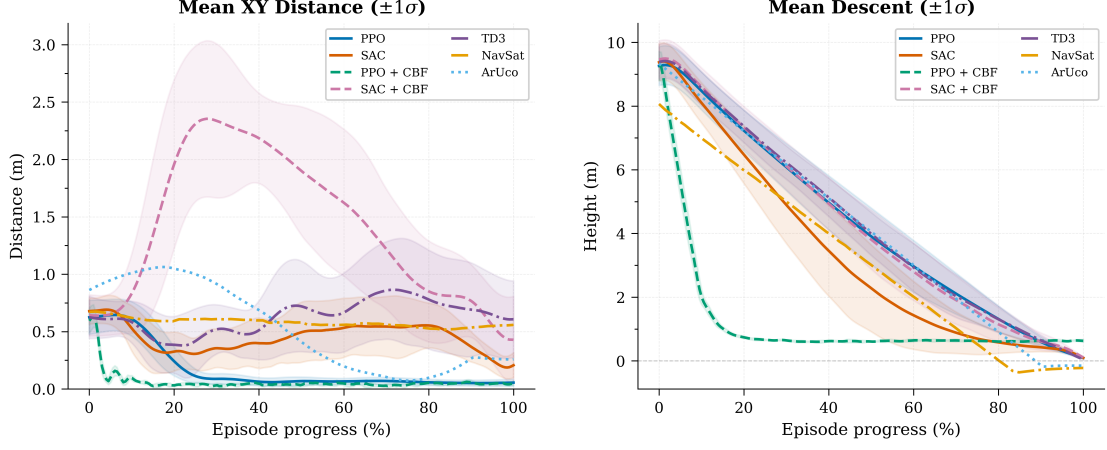


Figure 24: Mean height profile (solid line) with $\pm 1\sigma$ confidence bands (shaded region) over normalized episode time. PPO’s narrow band confirms high consistency; SAC’s wider band reflects variable approach strategies.

PPO’s narrow confidence band means that nearly all episodes follow the same descent profile, showing a stable and well converged policy. SAC’s wider band suggests the policy still exhibits exploration-like variance during inference, likely due to its entropy-regularized training objective.

11.4 Statistical Comparison

Figure 25 shows box-plot distributions of the key performance metrics across all methods.

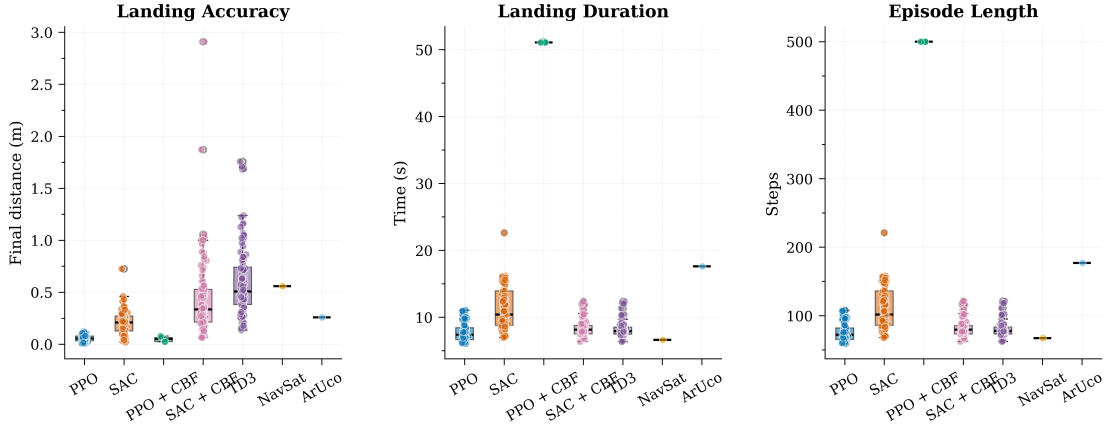


Figure 25: Box-plot comparison of final landing distance, landing time, and cumulative reward across all methods.

PPO has the tightest distribution with the lowest median and smallest interquartile range, confirming its precision advantage is consistent rather than coincidental. SAC’s distribution is wider with several outliers beyond 0.5 m. TD3 shows a wide distribution with a median around 0.5 m and outliers extending beyond 1.5 m, indicating inconsistent lateral alignment during touchdown. SAC+CBF shows the widest distribution including outliers near 3.0 m.

11.5 Combined Dashboard

Figure 26 summarizes all key evaluations that we discussed above into one individual figure. Final discussion takes place in the next section where we also discuss the RL approach compared to the reactive controllers.

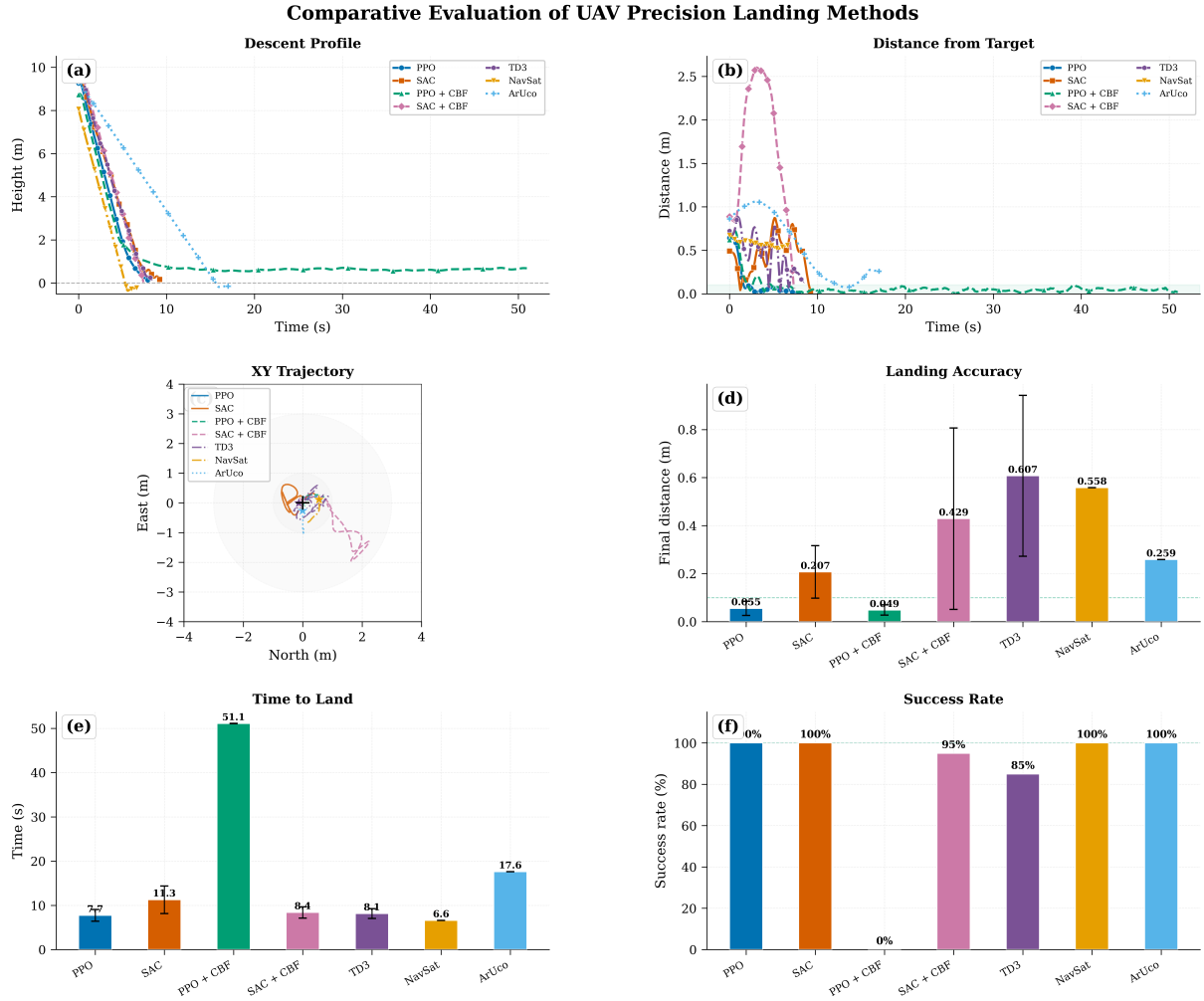


Figure 26: Combined evaluation dashboard.

11.6 Reactive vs. Reinforcement Learning Methods

Table 9: Qualitative comparison: reactive controllers vs. reinforcement learning approaches.

Criterion	Reactive Controllers	RL Methods
Design effort	High: requires control theory expertise, gain tuning, state machine design	Moderate: requires reward engineering and hyperparameter tuning
Precision	Limited by sensor accuracy (GPS: ± 2 m)	Can learn to compensate for sensor noise; PPO achieves 0.055 m
Adaptability	Fixed behavior; any change requires manual re-tuning	Adapts to different conditions through training
Interpretability	Full transparency: every gain and threshold is explicit	Black-box policy; actions may be non-intuitive
Safety guarantees	Deterministic bounds from velocity clamping	No inherent guarantees (addressable via CBF)
Compute at runtime	Negligible	Negligible (inference is a matrix multiply)
Training cost	None	Significant (1M+ timesteps, hours of GPU time)

One of the agenda of this thesis was to discuss the relevance of the RL methods. With the table above we discuss how the RL methods improve metrics of the autonomous landing task.

The NavSat controller achieves the fastest landing but bad precision (0.558 m), directly reflecting the coarse GPS sensor. ArUco delivers better precision (0.259 m) but at the cost of significantly longer landing time (17.6 s) and dependence on visual conditions (lighting, marker visibility).

PPO surpasses all reactive methods in landing accuracy by a factor of 4.7 times (vs. ArUco) to 10.1 times (vs. NavSat), while also being faster than ArUco and comparable in speed to NavSat.

This shows that the learned policy effectively extracts more information useful for the landing task than a hand tuned controller can.

11.7 Comparison Among RL Methods

Figure 27 shows the training convergence across 1,000,000 timesteps for the reinforcement learning methods, illustrating the mean episode reward and episode length rollouts. The red curves distinct divergence (PPO+CBF) clearly shows the hovering failure discussed later in the evaluation, while the other methods show a similar time and reward convergence.

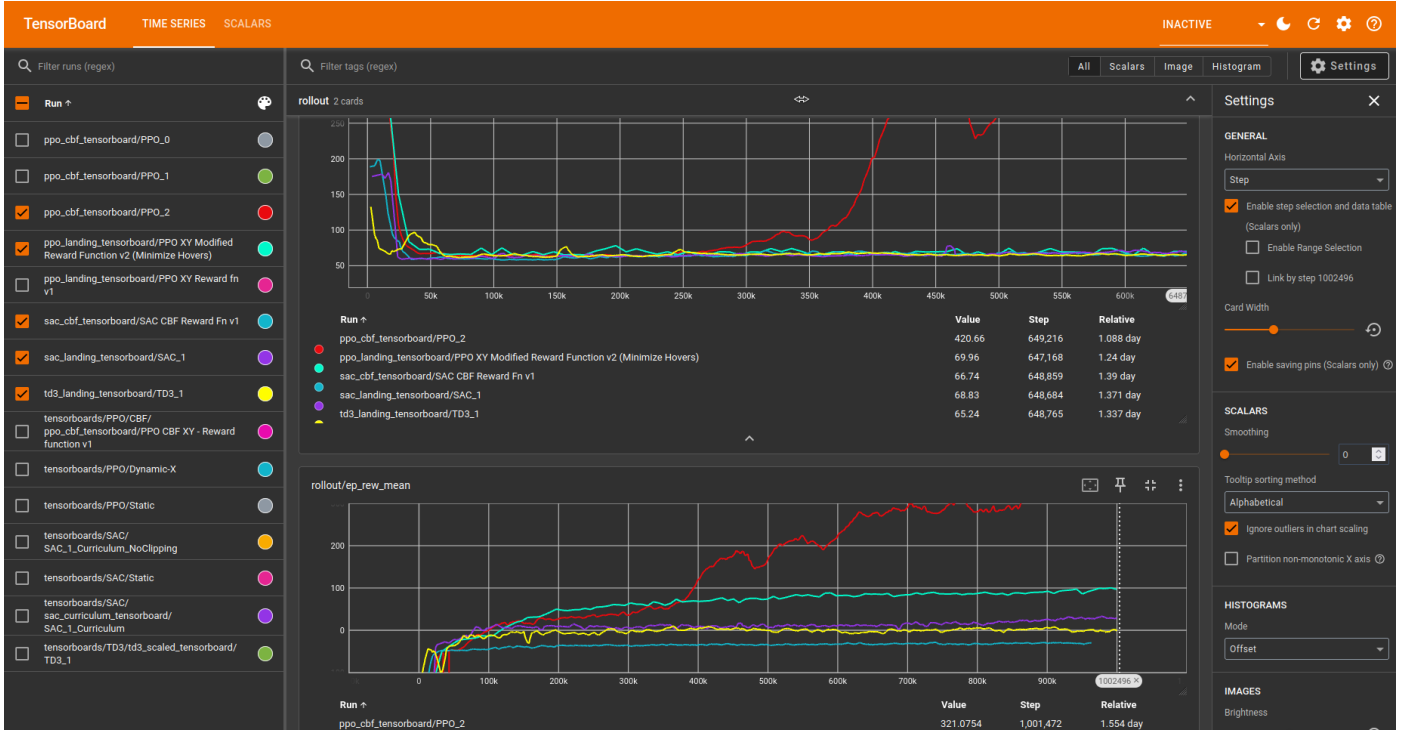


Figure 27: TensorBoard training curves showing the mean episode length (top) and mean episode reward (bottom) across 1,000,000 timesteps for the evaluated RL models.

11.7.1 PPO vs. SAC

PPO outperformed SAC across every metric:

11.8 Comparison Among RL Methods

11.8.1 PPO vs. SAC

PPO outperformed SAC across every metric:

- **Precision:** PPO's mean final distance (0.055 m) was 3.8 times better than SAC's (0.207 m).
- **Speed:** PPO landed in 7.7s vs. SAC's 11.3s (31% faster).
- **Quality:** 93% Perfect for PPO vs. 19% Perfect for SAC.
- **Reward:** PPO accumulated 102.1 ± 14.7 per episode vs. SAC's 34.8 ± 18.5 (with the same reward function design).

The reason for the performance gap can be attributed to several factors. PPO is an on policy algorithm. This means that it optimizes the current policy directly using the most recent experience, which can lead to a more stable convergence for the relatively low dimensional task (9-dimensional observation space and 3-dimensional action space). SAC on the other hand being an off-policy algorithm uses and optimizes both actor and critic networks with entropy regularization, which definitely provides better exploration but may lead to overfitting, falling back to suboptimal strategies in the buffer. Additionally, PPO's network architecture ([64, 64]) is more compact than SAC's ([256, 256]), which may reduce overfitting for this task's complexity.

11.8.2 TD3 Performance

TD3 achieved an 85% success rate with a mean final distance of $0.607\text{ m} \pm 0.335\text{ m}$, making it the least precise of the three RL algorithms. Despite using the same observation space, action space, and reward function as PPO and SAC, TD3’s landing precision was 11.0 times worse than PPO and 2.9 times worse than SAC.

However, TD3 was the second fastest RL method to land at $8.1\text{ s} \pm 1.1\text{ s}$, comparable to PPO’s 7.7 s and notably faster than SAC’s 11.3 s .

The deterministic policy gradient used by TD3 (with additive Gaussian exploration noise $\sigma = 0.1$) may be less effective than SAC’s stochastic policy for learning fine-grained lateral corrections during the final approach phase, where small adjustments are critical. Moreover the twin critic architecture and the delayed policy updates whilst stabilizing training for continuous control could be the cause of slow convergence of the precision critical components of the policy.

11.8.3 Effect of Network Architecture

It is also worth noticing that the PPO uses a significantly smaller network (two hidden layers of 64 neurons each, totaling ≈ 8800 parameters for the actor) compared to SAC and TD3 (two hidden layers of 256 neurons, totaling $\approx 70\,000$ parameters). Despite this 8 times difference in model capacity, PPO achieves superior results, suggesting that the landing task does not require large function approximators and that smaller networks may generalize better by avoiding overfitting to training-specific noise.

11.9 Control Barrier Function Analysis

11.9.1 SAC + CBF Performance

The SAC+CBF variant achieved a 95% success rate with a mean final distance of $0.429\text{ m} \pm 0.378\text{ m}$. The CBF intervention rate was $12.1\% \pm 10.5\%$, indicating that on average, the safety filter modified 12% of the actions commanded by the SAC policy.

Compared to baseline SAC (100%, $\bar{d}_{xy} = 0.207\text{ m}$), the CBF-augmented variant shows degraded accuracy and reliability.

However, the CBF layer provides a formal safety guarantee: across all 100 episodes, the drone never exceeded the platform boundary radius of 3.0 m and never collided with the ground. This safety-performance trade-off is a fundamental characteristic of CBF-augmented control.

11.9.2 PPO + CBF: The Hovering Problem

The PPO+CBF variant failed: the drone constantly hovered above the platform without landing until the episode was truncated. Despite achieving excellent horizontal alignment ($\bar{d}_{xy} = 0.049\text{ m}$), all 5 episodes terminated by truncation at 500 steps with a mean height of 0.63 m above the platform.

Root cause analysis identified six contributing factors:

1. **Insufficient descent incentive:** The original reward function provided only $+0.2$ for generic descent, insufficient to overcome the CBF-imposed braking near the ground.

2. **CBF ground constraint over-restriction:** The ground barrier function ($h_2 = z - z_{min}$) with $\alpha_2 = 2.0$ aggressively limited descent velocity below 0.5 m, creating a velocity-height equilibrium that prevented touchdown.

These findings highlight a non-trivial interaction between the CBF safety layer and the RL reward landscape. The safety constraints create an implicit “comfort zone” where the agent receives positive rewards without needing to complete the risky final descent. Resolving this requires jointly tuning the reward function, CBF parameters, and landing thresholds, which is a significantly more complex design problem than training either component in isolation.

Corrective modifications were implemented (increased descent rewards, added hovering penalty, relaxed CBF ground and velocity constraints, increased landing threshold to 0.12 m), but the retrained model showed the same hovering behavior. The reasoning could not be evaluated in time for inclusion in this comparison.

11.10 Discussion

11.10.1 Overall Ranking

Based on the evaluation results, the methods can be ranked by landing precision as:

1. **PPO:** Best overall with the highest precision, high consistency, and fast landing
2. **SAC:** Good precision and reliability, but slower and less consistent
3. **ArUco:** Good precision for a reactive method, but significantly slower
4. **SAC+CBF:** Moderate precision with safety guarantees, slight reliability degradation
5. **NavSat:** Fast but imprecise, limited by the GPS sensor
6. **TD3:** Fast descent but lowest precision among RL methods (85% success, $\bar{d}_{xy} = 0.607$ m)
7. **PPO+CBF:** Failure case with excellent alignment but inability to land

11.10.2 Reward Function Design

The reward design topic comes yet again into discussion, PPO’s success can be partially attributed to the dense, multi-component reward function (Table 10) that provides continuous gradient toward the desired behavior: approach, align, descend, and land.

Table 10: Key reward function components used across the RL environments.

Component	Base (PPO/SAC/TD3)	CBF	SAC-CBF
Distance penalty ($-\cdot d_{xy}$)	−0.1	−0.1	−0.1
Height penalty ($-\cdot h $)	−0.1	−0.1	−0.3*
Descent bonus (aligned)	+0.5	+0.5	+0.8
Terminal reward (perfect)	+50	+50	+15
Hovering penalty	−0.3	−0.3	n/a
CBF-safe bonus	n/a	+0.02	n/a

* SAC-CBF uses $-0.3 \cdot h$ when $d_{xy} < 1.0$ m, otherwise $-0.05 \cdot h$.

Despite receiving the same reward signal, TD3’s off-policy deterministic gradient was less effective at optimizing for precision than PPO’s on-policy updates or SAC’s entropy-regularized exploration.

11.10.3 Limitations of This Evaluation

Several limitations should be noted:

1. **Simulation only:** All the evaluations were done using the PX4-Autopilot SITL coupled with Gazebo simulation. All evaluations were conducted in the Gazebo simulation. Real-world factors such as sensor latency, communication delays, and motor response variability are not captured.
2. **Single training seed:** Each RL method was trained with a single random seed. Results may vary with different seeds, and a multi-seed evaluation would strengthen the conclusions.
3. **Hyperparameter sensitivity:** No systematic hyperparameter search was conducted. It is possible that SAC or TD3 could match PPO’s performance with different hyperparameters or network architectures.

12 Conclusion and Future Work

12.1 Summary of Contributions

The thesis dealt with the problem of autonomous UAV landing on a moving platform using both classical control and reinforcement learning approaches with the following contributions.

1. **Simulation Setup:** A custom Gazebo simulation world with a customizable moving platform with an embedded ArUco marker was created and the PX4-Autopilot was comprehensively bridged between the simulation and ROS2 network protocols.
2. **Reactive Controller Design:** Two classical controllers were developed: NavSat (GPS-only), ArUco (vision-based). The controller design was grounded in control-theoretic analysis, modeling the drone as a first-order velocity lag (PT_1) plant and designing proportional controllers with velocity feedback for damping.
3. **Reinforcement Learning for Landing:** Three RL algorithms (PPO, SAC, TD3) were trained to perform the landing task using velocity commands. A carefully designed reward function with dense shaping, proximity bonuses, descent incentives, and quality-graded terminal rewards was developed to guide learning. PPO achieved 100% success with a mean landing precision of 0.055 m, outperforming all other methods.
4. **Control Barrier Function Safety Layer:** A formal CBF-based safety filter was designed and integrated with RL policies (PPO+CBF, SAC+CBF). Two barrier functions enforce platform boundary and ground collision avoidance. The safety layer guarantees constraint satisfaction during both training exploration and deployment.
5. **Comprehensive Comparative Evaluation:** A standardized evaluation framework was built to benchmark all metrics under similar conditions.

12.2 Key Findings

1. **RL methods outperformed classical control in precision:** PPO achieved final landing distances (0.055 m) that are almost 10 times better than GPS-based control (0.558 m) and 4.7 times better than vision-based control (0.259 m), while being faster or comparable in landing time.
2. **On-policy learning excels for this task:** The PPO being an on-policy algorithm outperformed SAC and TD3 (both being off-policy) in every metric despite having a smaller neural network. The relatively low dimensional state and action space and deterministic environment dynamics favor on-policy methods that directly optimize the current policy.
3. **CBF safety comes at a performance cost:** The CBF safety layer guarantees constraint satisfaction but worsen the landing performance. SAC+CBF’s success rate dropped from 100% to 95% and precision worsened by 2.1 times. PPO+CBF variant completely failed to land due to a learned hovering equilibrium to maximize rewards.
4. **Reward function design is critical:** The dense reward function with multiple shaping components was essential for successful training. The initial PPO training and the PPO+CBF hovering failure demonstrated that reward functions designed for unconstrained settings can produce pathological behavior when combined with safety constraints.
5. **Velocity control abstraction enables effective learning:** By interfacing through velocity commands rather than lower-level attitude or rate control, the RL agents operate on a manageable 3-dimensional action space while the PX4 autopilot handles stabilization, motor mixing, and attitude control.

12.3 Future Work

Several directions for future research emerge from this work:

1. **Sim-to-real transfer:** The most immediate extension is deploying the trained policies on physical hardware. Domain randomization (varying physics parameters, sensor noise characteristics, and wind conditions during training) could improve transferability to a real quadrotor and moving platform.
2. **CBF-RL co-design:** The PPO+CBF hovering failure reveals the need for joint optimization of the reward function and CBF parameters. Approaches such as reward shaping that accounts for CBF intervention, or training the CBF parameters alongside the policy, could resolve this issue.
3. **Multiple random seeds:** Training each method with multiple random seeds would strengthen the statistical conclusions and identify which reward terms contribute most to performance.
4. **Dynamic CBF constraint adaptation:** Rather than using fixed CBF parameters, learning the barrier function gains ($\alpha_1, \alpha_2, \alpha_3$) as a function of the landing phase could allow tighter constraints during approach and relaxed constraints during touchdown.

12.4 Final Remarks

The thesis shows that reinforcement learning, especially PPO, can achieve sub-10 cm landing precision on a moving platform, a level of precision that exceeds what classical reactive controllers achieve with the same sensors by an order of magnitude. The combination of the reward shaping and velocity level control, and the PX4 autopilot’s low-level stabilization creates a tractable learning problem that converges reliably within one million training steps.

However, the integration of formal safety guarantees through Control Barrier Functions remains an open challenge. While the CBF safety layer successfully prevents constraint violations, its interaction with the RL reward landscape can produce unintended equilibria. Bridging formal safety and learning-based control in a principled manner is a promising direction at the intersection of control theory and machine learning with significant implications for real-world autonomous systems deployment.

References

- [1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.
- [2] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.
- [3] Puterman, M. L. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.
- [4] Deisenroth, M. P., Neumann, G., & Peters, J. (2013). A Survey on Policy Search for Robotics. *Foundations and Trends in Robotics*, 2(1–2), 1–142.
- [5] Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems*, 12.
- [6] Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3–4), 229–256.
- [7] Greensmith, E., Bartlett, P. L., & Baxter, J. (2004). Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning. *Journal of Machine Learning Research*, 5, 1471–1530.
- [8] Konda, V. R., & Tsitsiklis, J. N. (2000). Actor-Critic Algorithms. *Advances in Neural Information Processing Systems*, 12, 1008–1014.
- [9] Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2016). High-Dimensional Continuous Control Using Generalized Advantage Estimation. *International Conference on Learning Representations*.
- [10] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.
- [11] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust Region Policy Optimization. *International Conference on Machine Learning*, 1889–1897.
- [12] Engstrom, L., Ilyas, A., Santurkar, S., et al. (2020). Implementation Matters in Deep RL: A Case Study on PPO and TRPO. *International Conference on Learning Representations*.
- [13] Thrun, S., & Schwartz, A. (1993). Issues in Using Function Approximation for Reinforcement Learning. *Proceedings of the Fourth Connectionist Models Summer School*, 255–263.
- [14] van Hasselt, H. (2010). Double Q-learning. *Advances in Neural Information Processing Systems*, 23, 2613–2621.
- [15] Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *International Conference on Machine Learning*, 1587–1596.

- [16] 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*, 1861–1870.
- [17] Haarnoja, T., Zhou, A., Hartikainen, K., et al. (2018). Soft Actor-Critic Algorithms and Applications. *arXiv preprint arXiv:1812.05905*.
- [18] Brockman, G., Cheung, V., Pettersson, L., et al. (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540*.
- [19] Groves, P.D. (2013). *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems* (2nd ed.). Artech House.
- [20] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *International Conference on Machine Learning*, 448–456.
- [21] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.
- [22] Ames, A. D., Coogan, S., Egerstedt, M., Notomista, G., Sreenath, K., & Tabuada, P. (2017). Control Barrier Functions: Theory and Applications. *European Control Conference*, 3420–3431.
- [23] Ames, A. D., Xu, X., Grizzle, J. W., & Tabuada, P. (2019). Control Barrier Function Based Quadratic Programs for Safety Critical Systems. *IEEE Transactions on Automatic Control*, 62(8), 3861–3876.
- [24] Cheng, R., Orosz, G., Murray, R. M., & Burdick, J. W. (2019). End-to-End Safe Reinforcement Learning through Barrier Functions for Safety-Critical Continuous Control Tasks. *AAAI Conference on Artificial Intelligence*, 33(1), 3387–3395.
- [25] Ng, A. Y., Harada, D., & Russell, S. (1999). Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. *International Conference on Machine Learning*, 278–287.
- [26] Franklin, G. F., Powell, J. D., & Emami-Naeini, A. (2015). *Feedback Control of Dynamic Systems* (7th ed.). Pearson.
- [27] Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum Learning. *International Conference on Machine Learning*, 41–48.
- [28] Soviany, P., Ionescu, R. T., Rota, P., & Sebe, N. (2021). Curriculum Learning: A Survey. *International Journal of Computer Vision*, 130, 1526–1565.
- [29] Kendoul, F. (2012). Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems. *Journal of Field Robotics*, 29(2), 315–378.
- [30] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep Reinforcement Learning That Matters. *AAAI Conference on Artificial Intelligence*, 32(1).

- [31] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22(268), 1–8.
- [32] TensorFlow. “Get Started with TensorBoard.” TensorFlow Core. Accessed February 2026. <https://www.tensorflow.org/tensorboard>.
- [33] Calin, D. (2025). The Complete Practical Guide to PPO with Stable-Baselines3. *Reinforcement Learning Path*. <https://www.reinforcementlearningpath.com>.
- [34] Calin, D. (2025). Step-by-step Soft Actor Critic (SAC) Implementation In SB3 with PyTorch. *Reinforcement Learning Path*. <https://www.reinforcementlearningpath.com>.
- [35] Anthropic. (2024). Claude: AI Assistant. <https://www.anthropic.com/claude>.
- [36] PX4 Autopilot Development Team, “PX4 User Guide,” <https://docs.px4.io/main/en/>, Accessed: December 2025.
- [37] PX4 Autopilot Development Team, “Gazebo Simulation,” https://docs.px4.io/main/en/sim_gazebo_gz/, Accessed: December 2025.
- [38] TU Berlin Spec Gazebo Sim Team, “Custom Simulation World,” https://git.tu-berlin.de/spec_gazebo_sim/custom-sim-world, Accessed: December 2025.