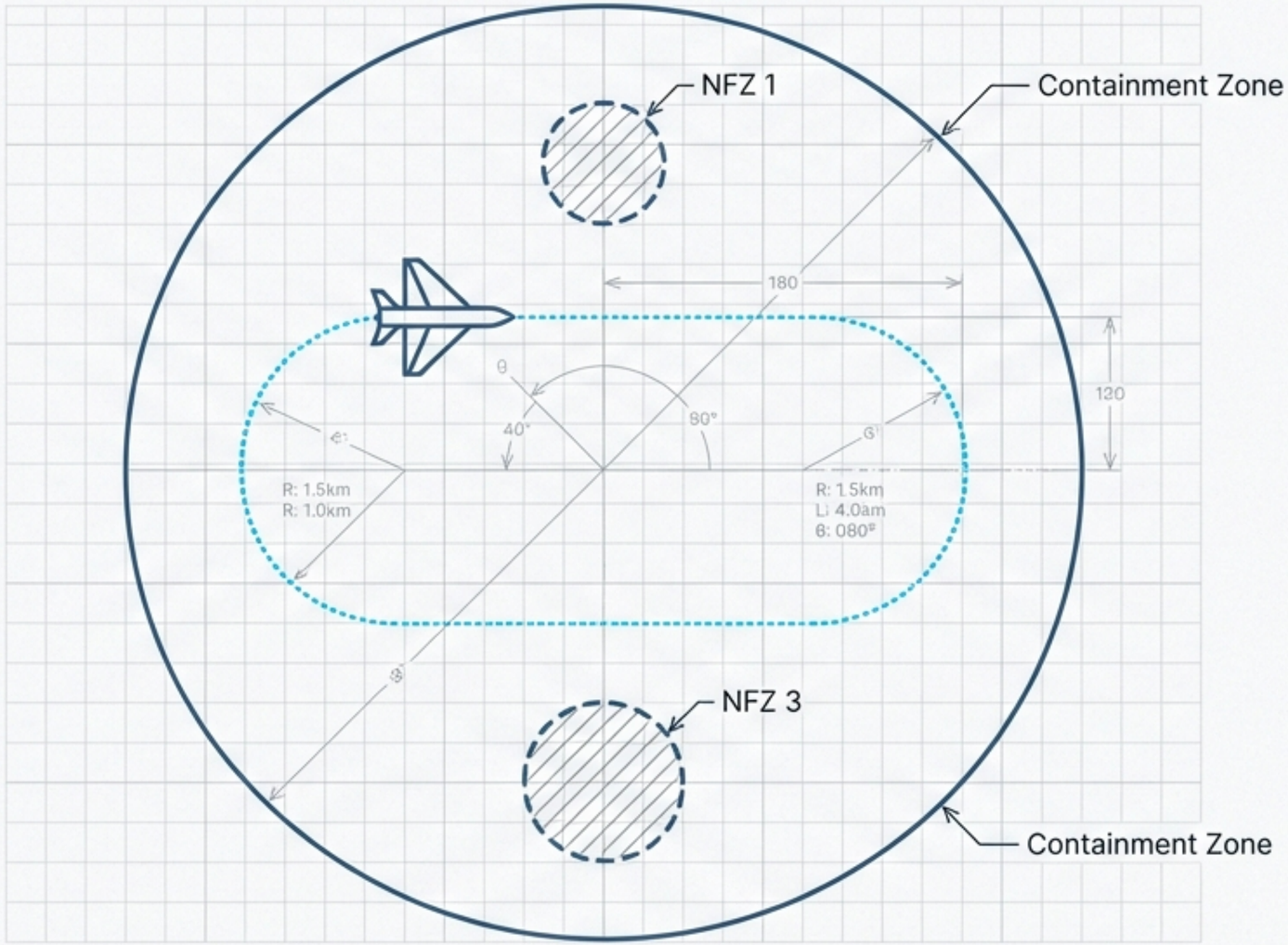
Building Simulator-Agnostic RL Environments

A Case Study in Clean Architecture with `HoldingAgentLidarEnv`

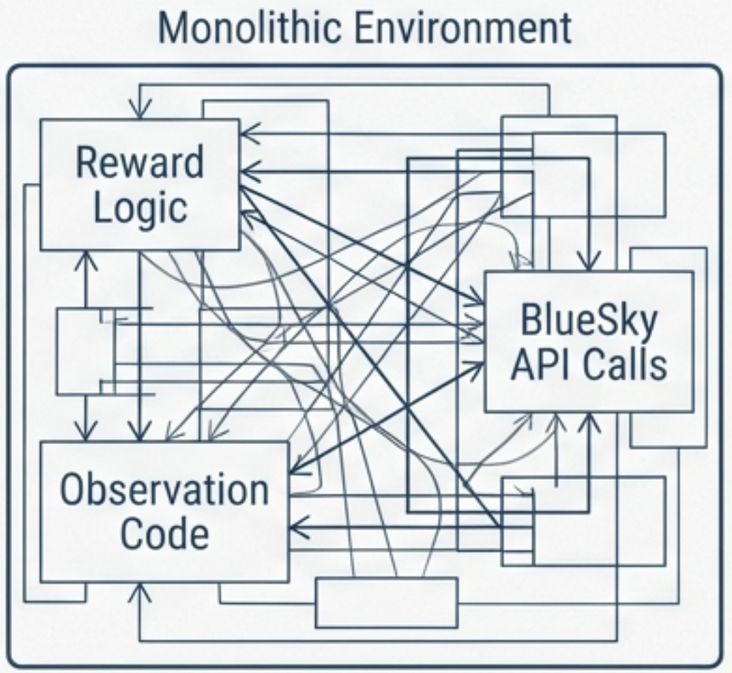


The Key to Robust RL Systems: Separating Logic from Simulation

The Problem: Tightly-Coupled Environments

Standard RL environments often fuse core logic (observations, rewards) directly with a specific simulator's API.

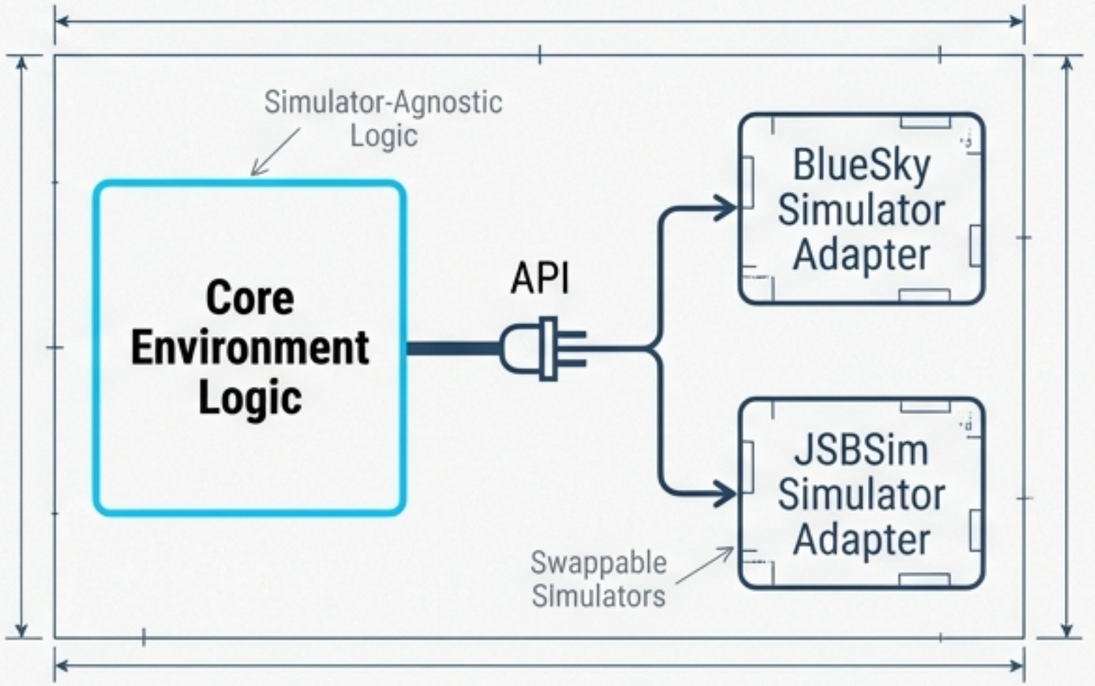
- Difficult to unit test logic in isolation.
- Migrating to a new simulator requires a major rewrite.
- Code becomes brittle and hard to maintain.



The Solution: A Clean Architecture

We enforce a strict boundary. The environment's core logic is simulator-agnostic, communicating through a defined interface.

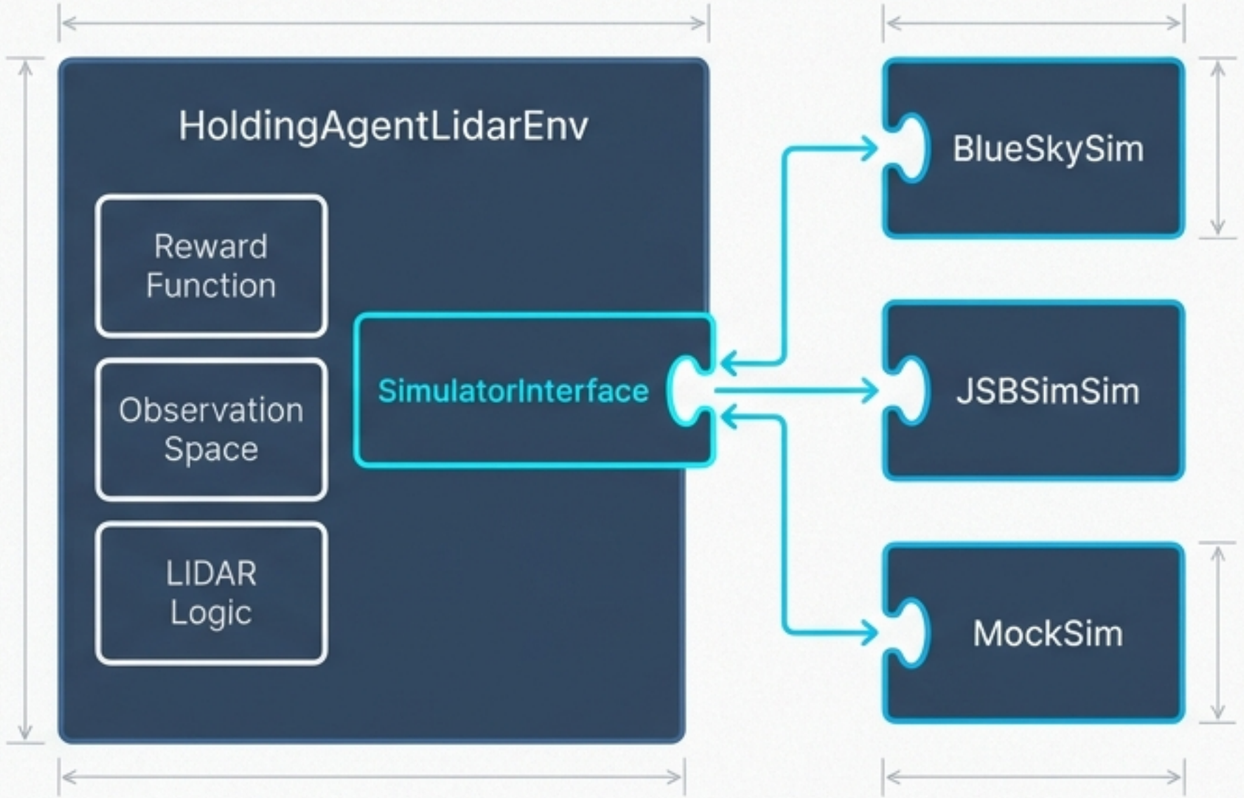
- Core logic is independently testable via mocks.
- Simulators become swappable 'plugins'.
- Promotes reusability and long-term maintainability.



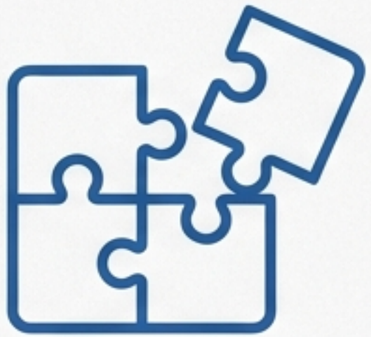
The SimulatorInterface: A Contract for Portability

The SimulatorInterface is an Abstract Base Class that defines the contract all simulator implementations must follow.
This decouples the environment from any **specific backend**.

```
class SimulatorInterface(ABC):
    def reset(**kwargs): ...
    def step(dt): ...
    def get_aircraft_position(id): ...
    def get_aircraft_heading(id): ...
    def get_aircraft_speed(id): ...
    def set_heading_change(id, change_deg): ...
    def close(): ...
```



A Single Design Choice Unlocks Four Key Benefits



Separation of Concerns

Environment logic (observations, rewards, LIDAR) is completely independent of the simulator. Easy to understand, test, and maintain.

Easy Simulator Swapping

Switch from BlueSky to JSBSim by changing one line of code.

```
sim = BlueSkySim()
 ↓
sim = JSBSim()
```

Testability

Mock the simulator interface to unit test the environment's core logic without the overhead of running a full simulation.

Reusability

The simulator interface and its implementations can be shared across multiple environment projects.

The Mission: Maintain a Holding Pattern and Avoid No-Fly Zones

Objective

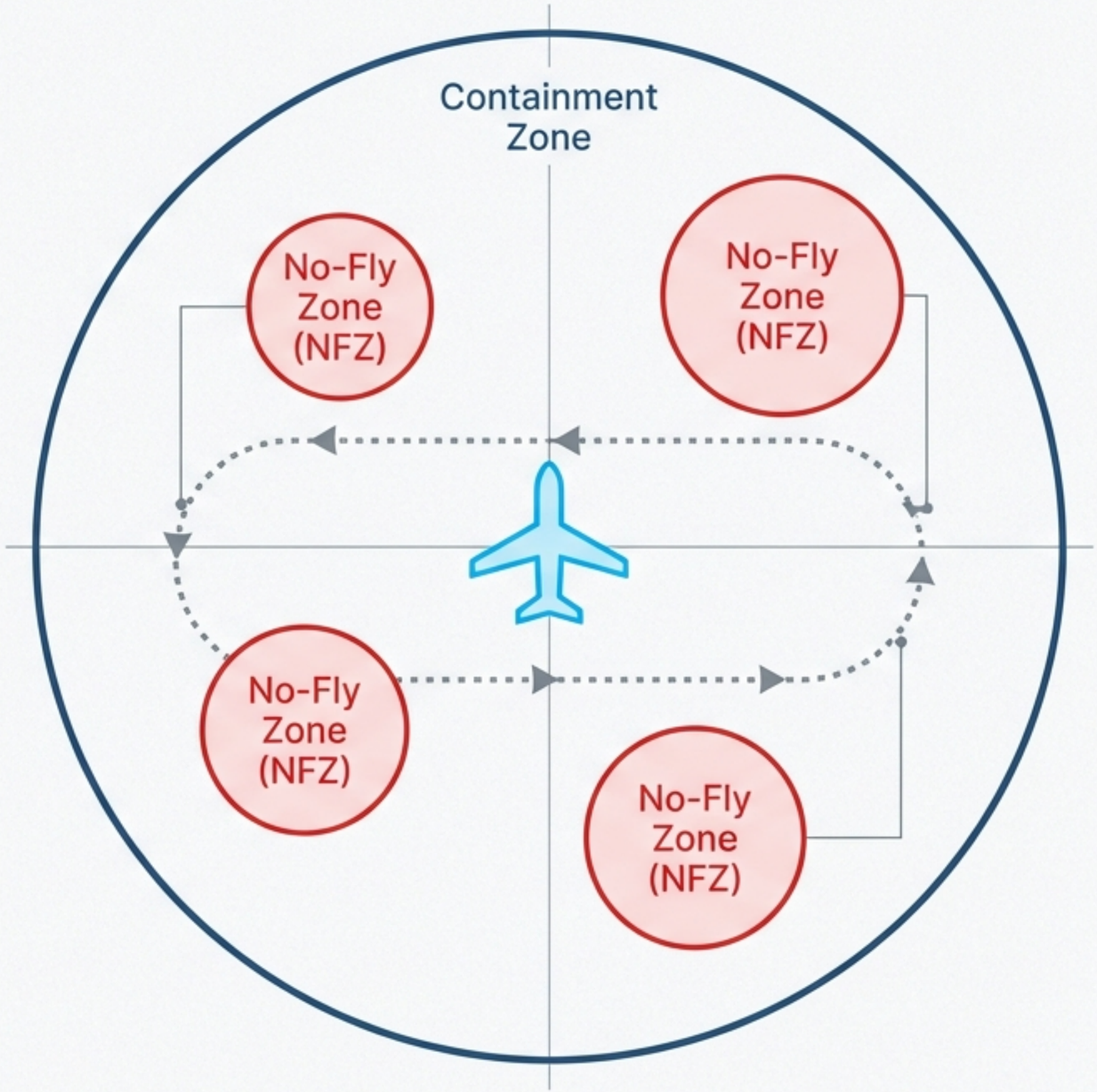
Train an agent to fly a continuous holding pattern inside a circular containment zone.

Constraint

The agent must actively avoid multiple circular No-Fly Zones (NFZs) within the area.

Control

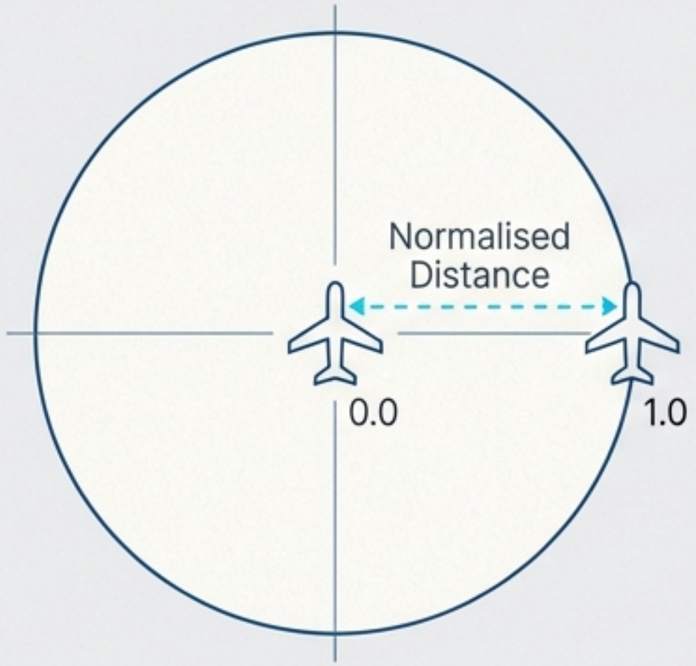
The agent issues continuous heading-rate commands to steer the aircraft.



The Agent's Perception: A Compact, Normalised Observation Space

Type: Continuous, Box(2 + num_rays) | **Range:** [0.0, 1.0]

obs[0]: Distance to Centre

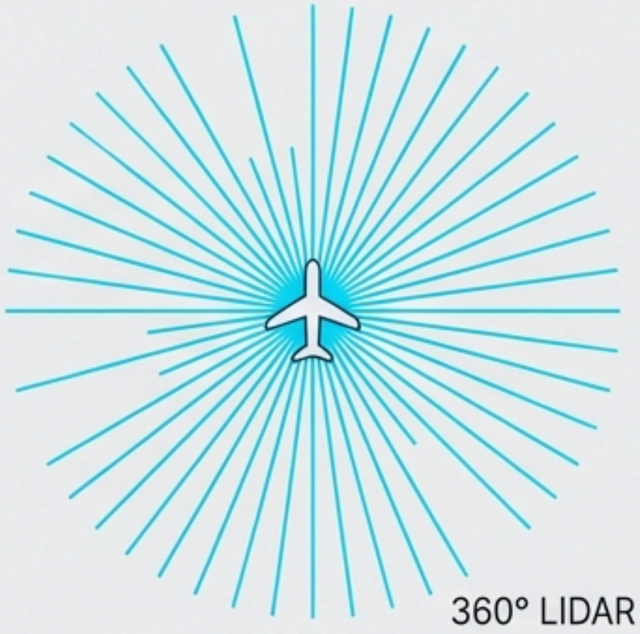


Normalised Euclidean distance from the aircraft to the centre of the holding pattern.

obs[1]: Relative Bearing to Centre

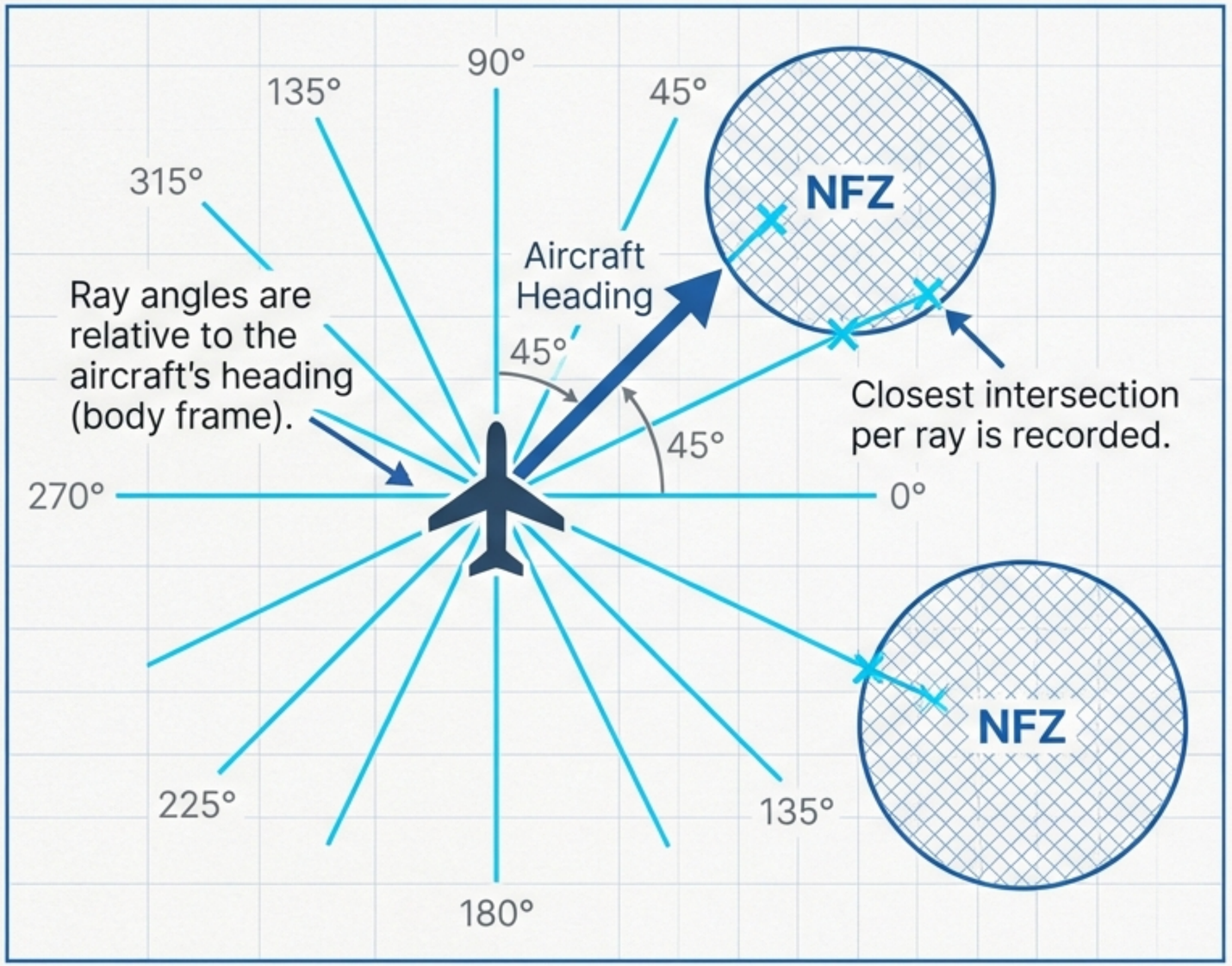
The direction to the centre, relative to the aircraft's current heading. 0.5 means the centre is directly ahead.

obs[2:2+num_rays]: LIDAR Distances



A 360° ring of simulated LIDAR distances to the nearest NFZs. 1.0 means no detection; 0.0 means a collision.

Sensing Threats: How the 360° LIDAR Simulation Works

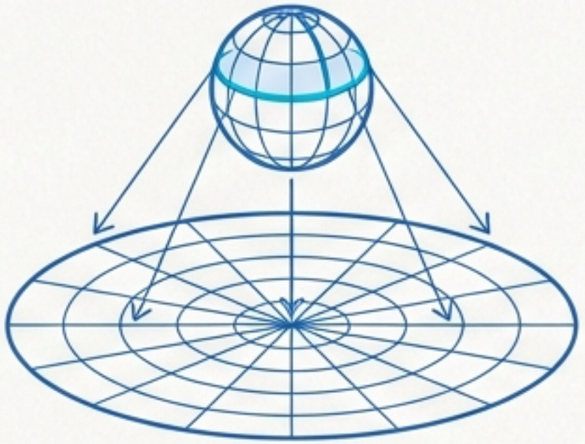


- **Method**
  Efficiently calculates intersections between rays and circles.
- **Distribution**
  Rays are evenly distributed 360° around the aircraft.
- **Frame of Reference**
  Ray angles are relative to the aircraft's heading, not the world frame.
- **Convention**
  Ray direction uses aviation standard: 0° is North, 90° is East.

From World to Local: Managing Coordinate System Transformations

The simulator provides Latitude/Longitude, but the environment's physics and LIDAR calculations require a local XY Cartesian plane (in metres).

Pyproj (High Accuracy)

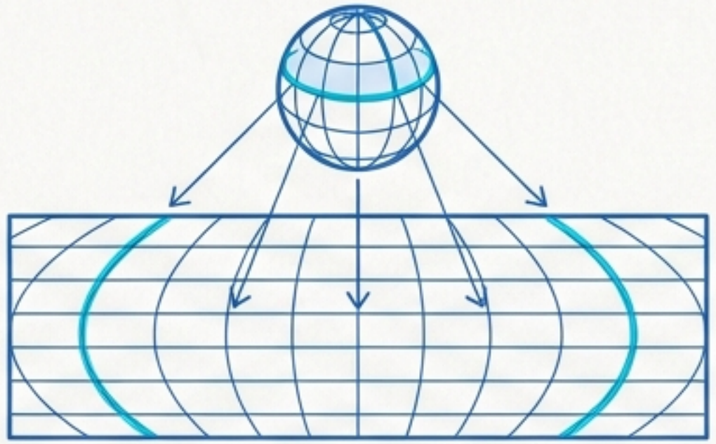


Method: Azimuthal Equidistant projection.

Benefit: Preserves true distances and bearings from the centre point. Geodetically accurate for any radius, anywhere on the globe.

Use Case: Recommended for simulations with a large radius (> 50-100km).

Equirectangular (Fast Approximation)



Method: A simpler, faster mathematical conversion.

Benefit: No external dependencies.

Trade-off: `<1%` error for radii up to `~100km` at mid-latitudes (< 60°).

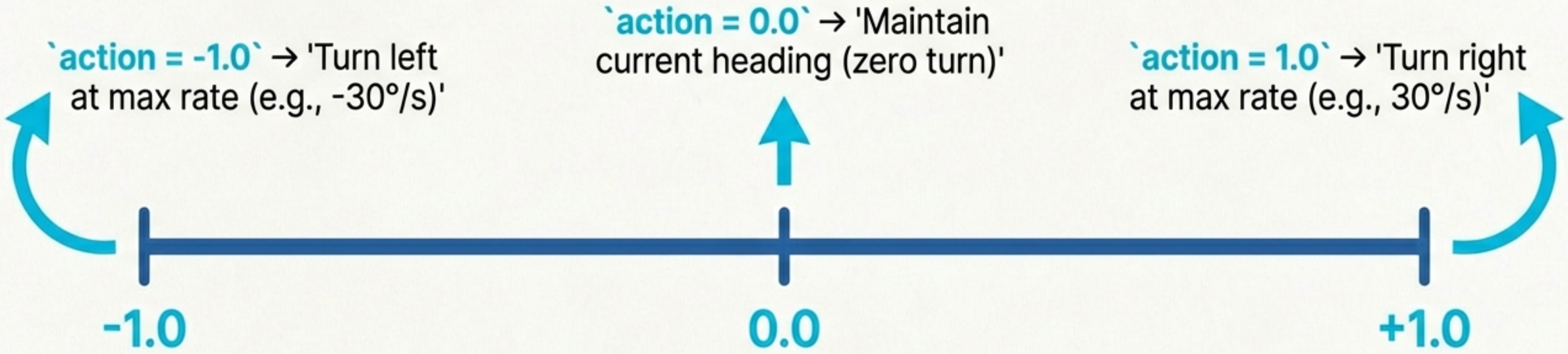
Design Rationale Providing a fast, dependency-free fallback (Equirectangular) ensures the environment is easily usable out-of-the-box, while allowing for professional-grade accuracy with `pyproj` when required.

The Agent's Control: Commanding Heading-Rate Changes

`Type: Continuous, Box(1) | Range: [-1.0, 1.0]`

The agent outputs a single continuous value which is scaled to a physical rate of heading change.

heading change.



This provides smooth, continuous control suitable for flight dynamics.

Shaping Behaviour: The Multi-Component Reward Function

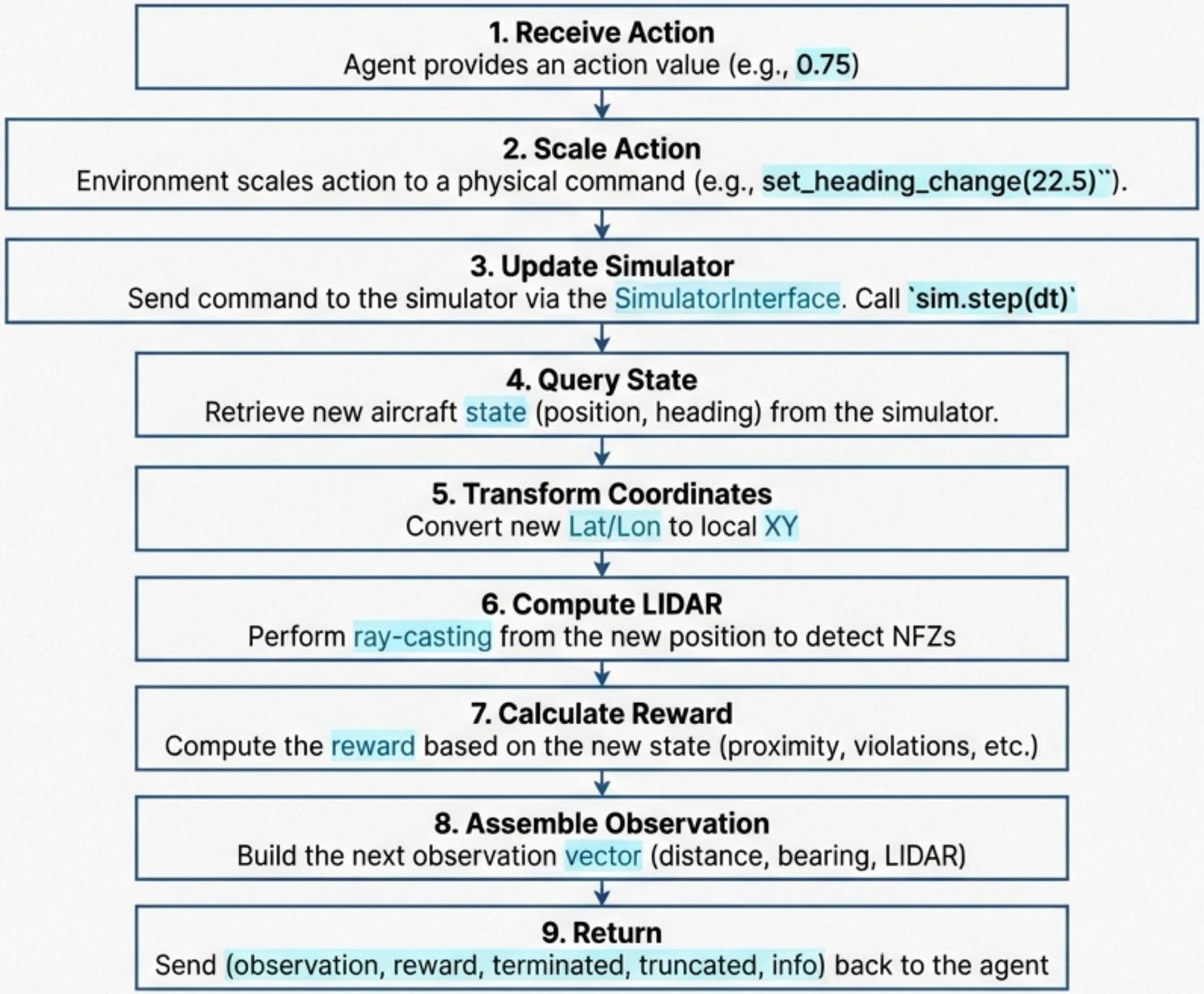
The reward function is a sum of carefully balanced components designed to encourage the desired behaviour.

`+0.1`	**Survival Reward:** A small, constant reward for every step the agent survives without incident. Encourages longer episodes.
`-10.0`	**Collision Penalty:** A moderate penalty for entering an NFZ.
`-100.0`	**Boundary Penalty:** A large penalty for exiting the containment zone.
`Variable`	**Proximity Penalty:** A smooth, continuous penalty that increases as the agent gets closer to an NFZ boundary, providing a useful gradient for learning.

Design Rationale: Non-Terminating Violations

Episodes do not terminate immediately upon an NFZ or boundary violation. This allows allows the agent to experience consequences and learn recovery behaviours, which is more robust for on-policy algorithms like PPO.

Anatomy of a Timestep: The `env.step()` Execution Flow

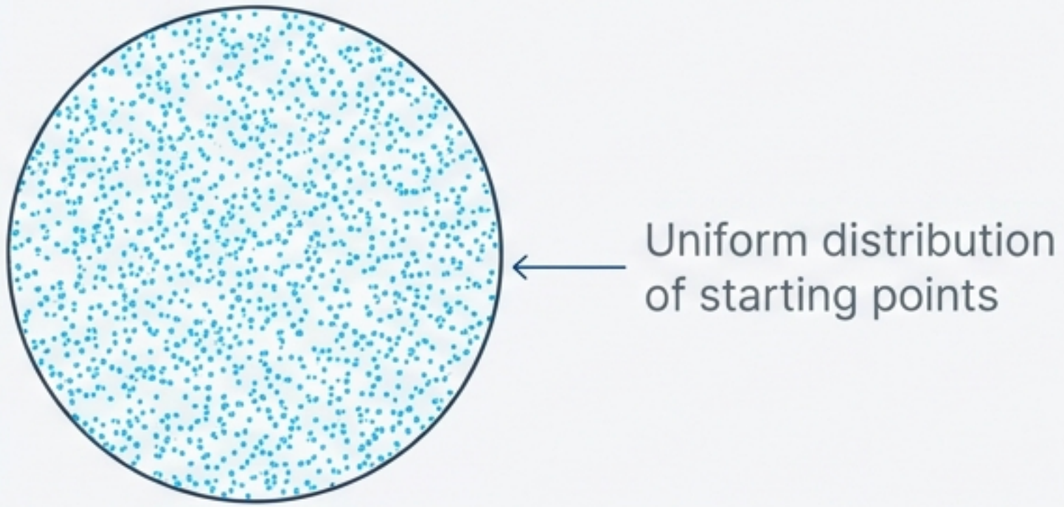


Ensuring Fair Starts: Episode Reset and Randomisation

At the start of each episode, the aircraft's state is randomised to ensure the agent learns a general policy, not one specific to a single starting condition.

****Randomised Parameters****

- **Initial Heading:** Uniformly sampled from [0, 360) degrees.
- **Initial Position:** Placed at a random point within the containment zone.



Key Insight: Uniform Area Sampling

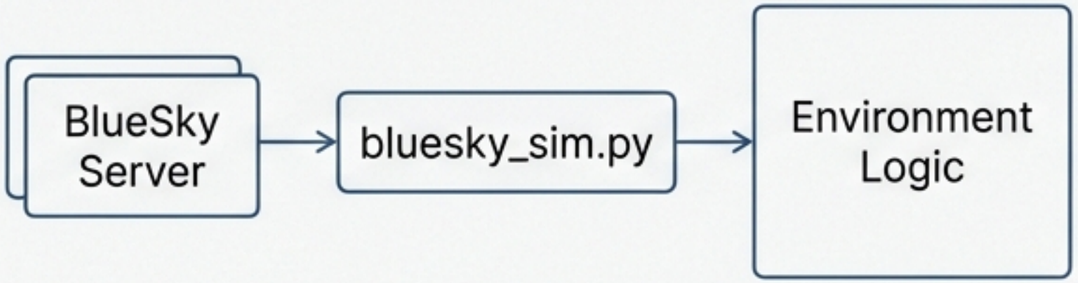
To ensure a uniform distribution across the circular *area*, the starting radius is sampled as `R * sqrt(U(0,1))` where R is the zone radius and random variable. Sampling the radius linearly would bias starting positions towards the centre.

Usage and Extensibility: Your Simulator, Your Choice

Getting Started: BlueSky

To use with the BlueSky simulator:

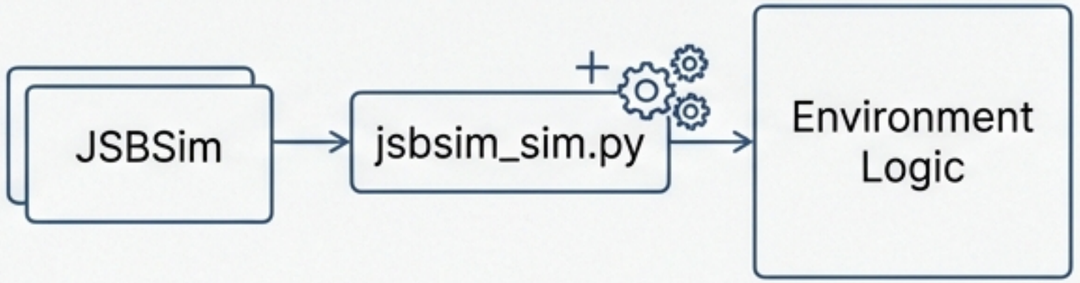
1. Implement the methods in `envs/simulators/bluesky_sim.py`.
2. Connect to your BlueSky server instance.
3. The environment logic works out-of-the-box.



Extending: Add JSBSim

To add a new simulator like JSBSim:

1. Create a new file: `envs/simulators/jsbsim_sim.py`.
2. Implement the methods of the `SimulatorInterface` contract.
3. Use it with the *exact same* environment code.



Dependencies: `Required: gymnasium, numpy | Optional: pyproj`

The Principle Defines the Product

The clean separation of environment logic from simulator I/O is not an implementation detail —it is the foundational design choice that enables testability, portability, and reuse.