# Multi Agent Reinforcement Learning System


Realised by

Aziz Hachicha

Data Science Student

**Introduction**

Preventing collisions in multi-robot systems is a major challenge, especially when using learning-based approaches like Multi-Agent Reinforcement Learning (MARL) that lack built-in safety guarantees. Our **Layered Safe MARL** framework combines the adaptability of MARL with multiple safety layers to ensure reliable, collision-free navigation.

## Notation (global)

- $i$ — agent index.
- $\mathbf{x}_i \in \mathbb{R}^2$ — position $(x, y)$.
- $\mathbf{v}_i \in \mathbb{R}^2$ — velocity $(v_x, v_y)$.
- $\mathbf{a}_i \in \mathbb{R}^2$ — acceleration (action).
- $\Delta t$ — time step ( `dt` in code).
- $N$ — number of agents.
- $r_{\text{safety}}, r_{\text{conflict}}$ — safety & conflict radii.
- $\varepsilon$ — small constant (e.g., $10^{-6}$).

Part A — **Deployment (Runtime) Working Flow**

This is the flow used when the model runs on robots or simulators:

**1. Observation construction (sensing → state)**

**Formula (observation for agent i):**

Formula (observation for agent $i$):

$$o_i = \left[\mathbf{x}_i,\ \mathbf{v}_i,\ (\mathbf{x}_j - \mathbf{x}_i,\ \mathbf{v}_j)_{j \neq i}\right] \qquad d_{\text{obs}} = 4 + 4(N-1).$$

**Why:** relative positions $\mathbf{x}_j - \mathbf{x}_i$ remove dependence on global coordinates and emphasize interactions.

**Code:** `envs/multi_agent_env.py` → `_get_obs()`

## 2. Layer 1 — MARL policy inference (nominal action)

**Formulas (actor MLP):**

$$
\begin{aligned}
h^{(1)} &= \mathrm{ReLU}(W^{(1)} s + b^{(1)}),\\
h^{(2)} &= \mathrm{ReLU}(W^{(2)} h^{(1)} + b^{(2)}),\\
\mathbf{a}_i^{\mathrm{nom}} &= W^{(3)} h^{(2)} + b^{(3)}.
\end{aligned}
$$

**Why:** MLP gives a compact, fast inference mapping from observation to continuous acceleration.

**Code:** `models/gnn_actor_critic.py` (actor) & `agents/marl_policy.py` → `act(obs)`

## 3. Clip actions (actuator bounds)

**Piecewise definition (elementwise):**

$$
\mathrm{clip}(x, a, b) = \begin{cases} a, & x < a, \\ x, & a \le x \le b, \\ b, & x > b. \end{cases}
$$

Applied:

$$
\mathbf{a}_i^{\mathrm{clipped}} = \mathrm{clip}(\mathbf{a}_i^{\mathrm{nom}},\, -1,\, 1).
$$

**Why:** enforces physical actuator limits and prevents extreme commands.

**Code:** (clip before integration; currently `env.step()` clips)

Note: for minimal intrusiveness, clip **before** safety filter so filter preserves within-bounds magnitude.

## 4. Nearest-neighbor detection & engagement check

**Formulas:**

$$
d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|.
$$

Engagement predicate:

$$
\mathrm{engaged}_{ij} \;=\; d_{ij} \le d_{\mathrm{eng}}.
$$

**Why:** only nearby agents can cause imminent constraint violations; limit corrections to engaged neighbors.

**Code:** nearest neighbor in training loop

## 5. Conflict prioritization (urgency)

Urgency metric

$$u_{ij} = \frac{1}{d_{ij} - r_{\text{safety}} + \varepsilon}.$$

Prioritize the pairs with largest $u_{ij}$.

**Why:** In dense scenes handling all conflicts simultaneously is costly/unstable; prioritize highest-risk interactions.

## 6. Layer 3 — Safety Filter (CBVF)

### Barrier function

$$h_{ij}(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\| - r_{\text{safety}}.$$

Safe if $h_{ij} > 0$.

### Heuristic corrective action used in code

If $d_{ij} < r_{\text{safety}}$ and $d_{ij} > \varepsilon$:

$$\mathbf{a}_i^{\text{safe}} = \|\mathbf{a}_i^{\text{clipped}}\| \frac{\mathbf{x}_i - \mathbf{x}_j}{d_{ij}}.$$

**Why:** simple, fast last-resort correction: reorient acceleration away from intruder while preserving magnitude (minimal intrusion).

**Code:**

- `safety/reachability_utils.py` :

## 7. Final safe action & actuation (dynamics)

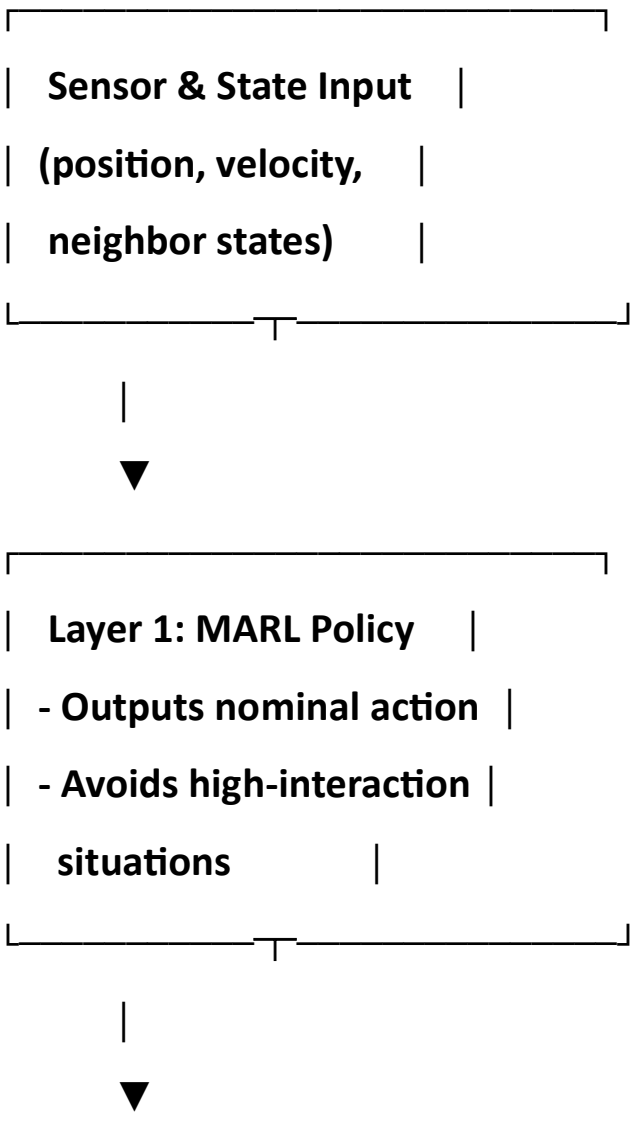
Integration (explicit Euler):

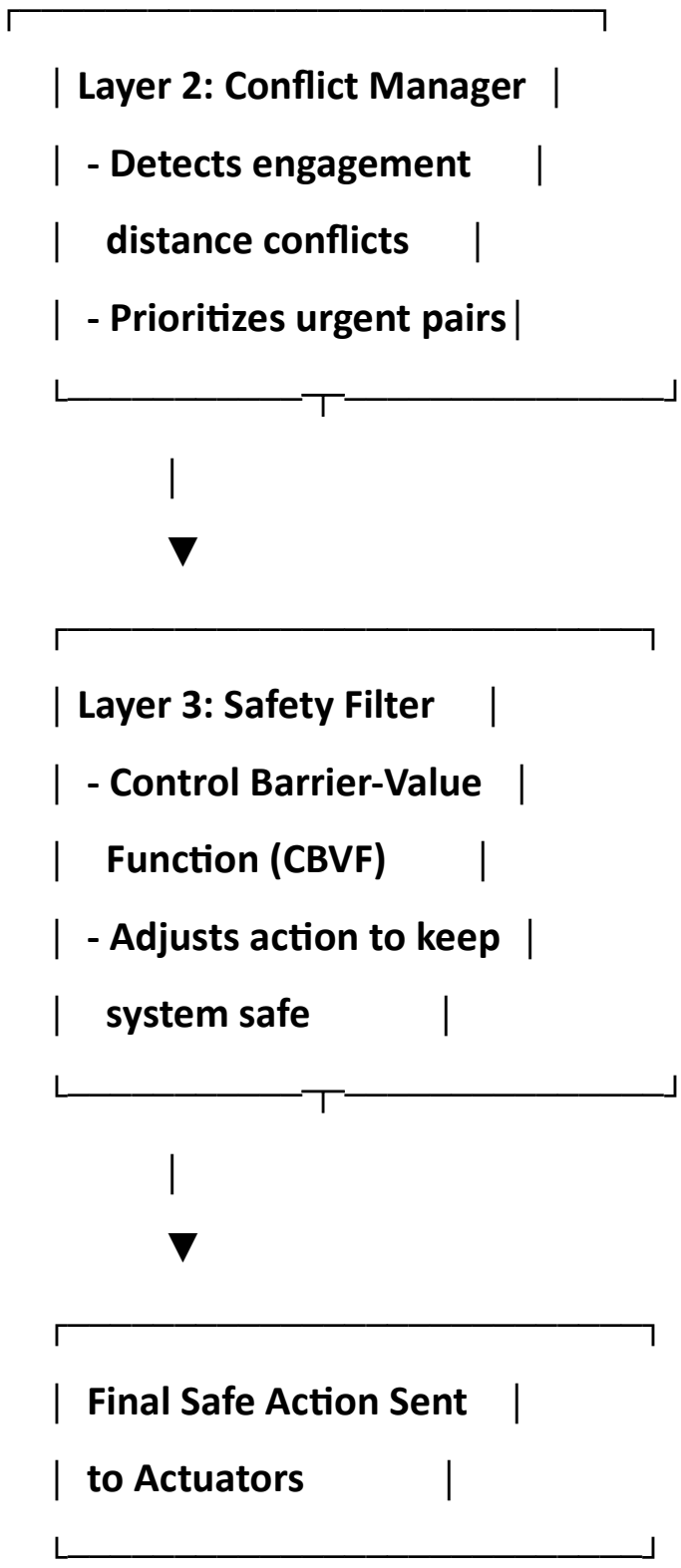
$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \mathbf{a}_i^{\text{final}} \Delta t,$$
$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{v}_i^{t+1} \Delta t.$$

**Why:** cheap, predictable, widely used for control loops when $\Delta t$ small.

**Code:** `envs/multi_agent_env.py` :

## System Overview




Layer 2: Conflict Manager
- Detects engagement
distance conflicts
- Prioritizes urgent pairs
Layer 3: Safety Filter
- Control Barrier-Value
Function (CBVF)
- Adjusts action to keep
system safe
Final Safe Action Sent
to Actuators

## Part B — Training (math, code mapping, working flow)

This section explains how the system learns the MARL policy and critic, including loss functions, curriculum, optimization, and checkpoints.

Overview of training flow

1. Initialize environment and MARLPolicy (actor & critic).
2. For step in total training steps:
    - Reset env and run episodes (episodes of length episode_length).
    - For each time step in episode: compute actions, apply CBVF, step env, compute rewards.
    - Form critic targets and update critic with MSE loss.
3. Periodic logging and checkpoint (torch.save).

**Code:** `training/train.py`

### 1. Loss functions — Critic (and note about actor)

For agent $i$ at time $t$, target $y_i^t$ is set as:

$$y_i^t = -r_i^t$$

(predicted as negative immediate reward — *a one-step dummy target*).

Loss:

$$\mathcal{L}(\phi) = \tfrac{1}{2}\left(V_\phi(o_i^t) - y_i^t\right)^2.$$

**Why:** Direct supervised regression of critic to observed immediate cost is simple and stable for initial experiments.

**Code:** `agents/marl_policy.py` → `update()`:

## 2. Curriculum schedule (safety ramp during training)

**Piecewise definition of $\lambda(t)$:**

$$\lambda(t) = \begin{cases} \dfrac{2t}{T}, & t \le T/2, \\ 1, & t > T/2, \end{cases}$$

then

$$s(t) = \lambda(t)^2, \quad r_{\text{safety}}(t) = s(t)\, r_{\text{safety}}^{\max}, \quad r_{\text{conflict}}(t) = s(t)\, r_{\text{conflict}}^{\max}.$$

**Why:** gradually increase difficulty/safety constraints to allow learning basic navigation first.

**Code:** `training/curriculum.py` → `get_safety_params(step, total_steps, max_r_safety, max_r_conflict)`:

## 3. Episode dynamics & reward (training-time same as deployment dynamics)

**Dynamics:** identical to Section A (Euler integration).

**Reward (per-agent) — used as training signal:**

$$\begin{aligned} r_{\text{dist}}^i &= -\|\mathbf{x}_i^t - \mathbf{g}_i\|, \\ \hat{d}_i^t &= \frac{\mathbf{g}_i - \mathbf{x}_i^t}{\|\mathbf{g}_i - \mathbf{x}_i^t\| + \varepsilon}, \\ r_{\text{align}}^i &= \mathbf{v}_i^t \cdot \hat{d}_i^t, \\ r_0^i &= r_{\text{dist}}^i + 0.1\, r_{\text{align}}^i, \\ P^i &= 0.5 \sum_{j:\|\mathbf{x}_i - \mathbf{x}_j\| < r_c} \left(r_c - \|\mathbf{x}_i - \mathbf{x}_j\|\right), \\ r^i &= r_0^i - P^i. \end{aligned}$$

**Why:** reward balances goal reaching and safety (penalty) with a small velocity alignment bonus to encourage smoothness.

**Code:** `envs/multi_agent_env.py` inside `step()`.

## 4. Training loop (detailed working flow)

**High-level math / algorithmic steps:**

For training steps $step = 0, \ldots, T-1$:

1. Reset environment: sample $\mathbf{x}^0, \mathbf{v}^0, \mathbf{g}$.
2. Compute $r_{\text{safety}}(step), r_{\text{conflict}}(step)$ via curriculum.
3. For episode time $t = 0, \ldots, L-1$:
   - For each agent $i$:
     - Observe $o_i^t$.
     - Compute nominal action $\mathbf{a}_i^{\text{nom}} = \pi_\theta(o_i^t)$.
     - Clip to bounds: $\mathbf{a}_i^{\text{clipped}} = \text{clip}(\mathbf{a}_i^{\text{nom}}, -1, 1)$.
     - Find nearest neighbor $j^*$ (or prioritized pair).
     - Apply CBVF filter if necessary to get $\mathbf{a}_i^{\text{final}}$.
   - Apply dynamics to get next state.
   - Compute reward $r_i^t$.
   - Form critic target $y_i^t = -r_i^t$ (current code).
   - Update critic by minimizing MSE $\frac{1}{2}(V_\phi(o_i^t) - y_i^t)^2$ (via Adam).
4. Every 100 steps, log; at end save model state dict.

**Exact code:** `training/train.py` (full loop) — key excerpts

## 5. Loss & optimization mathematics

**Critic MSE gradient:**

$$\mathcal{L}(\phi) = \tfrac{1}{2}\big(V_\phi(o) - y\big)^2, \qquad \nabla_\phi \mathcal{L} = \big(V_\phi(o) - y\big)\, \nabla_\phi V_\phi(o).$$

**Adam update (per parameter):**

$$\phi \leftarrow \phi - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where $\hat{m}_t, \hat{v}_t$ are bias-corrected moment estimates.

**Code:** `agents/marl_policy.py` (MSELoss + Adam optimizer initialization).

## 6. Saving & evaluation

**Saving:** `torch.save(policy.model.state_dict(), 'models/latest_policy.pt')` — stores actor+critic weights.

**Evaluation flow:** `evaluate.py` loads env and policy, runs `cfg['evaluation']['steps']` steps, computes mean reward.

**Code:** `evaluate_agent()` in `evaluate.py`.

## Conclusion

This document presents a full, working-flow oriented mapping between the Layered Safe MARL design and its mathematical foundations, including the runtime safety layers and the training pipeline. The deployment flow shows how MARL outputs are clipped, prioritized, and corrected by a Control Barrier-Value Filter before actuation; the training flow explains how the critic is currently trained using MSE on one-step negative rewards and how a curriculum gradually introduces safety constraints.