

Hierarchical Multi-Agent Reinforcement Learning for Humanoid Robots: A Lambda-Calculus Formalization and Functional Implementation Framework

Aslan Alwi*, Munirah, Almudaya Research Institute Team
aslan.alwi@umpo.ac.id, munirah@umpo.ac.id

Independent Researcher at Almudaya Research Institute
Department of Informatics Engineering, Faculty of Engineering, Universitas Muhammadiyah
Ponorogo, Indonesia

November 2025

Abstract

Learning-based control of humanoid robots remains challenging due to their high dimensionality, nonlinear dynamics, and the need to ensure safety and interpretability during adaptation. This paper introduces a Hierarchical Multi-Agent Reinforcement Learning (HMARL) framework that models a humanoid robot as an ensemble of interacting RL agents, consisting of (i) micro-level learners assigned to individual joints and (ii) a macro-level meta-RL agent responsible for evolving swarm coordination rules through an Epistemic Blackboard. A validator mechanism, inspired by blockchain-based multi-agent security, ensures that rule updates are safe, consistent, and explainable.

A key contribution of this work is a formal specification of HMARL using untyped lambda calculus. Policies, update operators, Bellman fixed points, meta-level rule evolution, and validator predicates are expressed as compositional λ -terms, enabling a clear theoretical separation between pure computation, recursion, and effectful ledger operations. This formal specification is then translated into a functional-style Python implementation, demonstrating that λ -calculus abstractions map directly to executable code.

A proof-of-concept simulation with an 8-joint humanoid surrogate shows that micro-agents can learn stabilizing and reaching behaviors, while the macro-agent learns to modulate coordination dynamics via validator-approved rule updates. Emergent whole-body behavior arises not from centralized optimization, but from the interaction of local RL and meta-level rule shaping.

The results suggest that HMARL provides a scalable, interpretable, and safety-aware pathway for humanoid motor learning, combining functional specification, hierarchical reinforcement learning, and distributed rule evolution. The framework opens avenues for deployable humanoid intelligence grounded in provable structure and formal semantics.

1 Introduction

Recent advances in humanoid robotics increasingly exploit data-driven and learning-based control to handle high degrees of freedom (DoF), rich contact interactions, and unstructured environments. Reinforcement learning (RL) has emerged as a powerful paradigm for acquiring complex motor skills [4, 7]. At the same time, multi-agent and distributed control perspectives reinterpret a single embodied robot as an ensemble of interacting controllers—e.g., joint-level reflexes, limb coordinators, and whole-body planners—opening the possibility of hierarchical RL across multiple semantic and temporal scales [1, 3, 5].

This paper proposes and formalizes a **Hierarchical Multi-Agent Reinforcement Learning (HMARL)** framework tailored for humanoid robots in which:

- each **joint** (or small joint cluster) is modeled as a *micro-RL agent* that learns local control policies (“micro-policies”) for reflexive stabilization, compliance, and low-level torque control;

- an **Epistemic Blackboard** agent (the macro level) functions as a *meta-RL agent* that proposes, tunes, and evolves swarm/coordination rules (the “macro-policy”) which govern inter-agent interactions and task allocation;
- the ensemble of agents forms a *hierarchy of RL learners* whose interaction yields emergent whole-body behaviors while preserving safety via validator mechanisms (e.g., the blockchain-based validator introduced in preceding work).

A contribution of this paper is a twofold formalization and implementation pathway:

1. a **mathematical and programmatic formalization** of HMARL using *lambda calculus* (λ -calculus) to express policies, update operators, and recursive Bellman-style fixed points in a concise and Turing-complete notation; and
2. a pragmatic **translation to Python** (functional style) that preserves the λ -calculus structure, enabling direct implementation of proof-of-concept (PoC) systems where micro-agents and macro-agents are realized as composable functions and parameter update operators.

The λ -calculus formalization offers several advantages for both theory and engineering. It exposes the recursive structure of value-based updates (via fixed-point combinators), makes explicit the distinction between pure computations and side-effects (e.g., ledger updates), and aligns closely with implementation idioms in modern functional and differentiable programming frameworks. Translating the λ -specification to Python facilitates rapid prototyping of the HMARL architecture: each mathematical function corresponds to a Python `lambda` or higher-order function, and policy parameters map to tensors used by deep function approximators (PyTorch / JAX).

This paper is organized as follows. Section 2 introduces the HMARL architecture and the RL decomposition across micro and macro agents. Section 3 presents the λ -calculus formalization, including fixed-point expressions for Bellman operators and meta-policy recursion. Section 4 details the translation patterns from λ to implementable Python code, emphasizing pure functions, monadic modeling for side effects (ledger/validator), and differentiable constructions for gradient-based updates. Section 5 describes a compact proof-of-concept implementation (simulator, micro/macro agents, training loop) and reports results on representative tasks (balance recovery and compliant reaching). Section 6 discusses safety, scalability, and limitations. Section 7 concludes and outlines future research directions.

1.1 Contributions

The main contributions are:

- the HMARL conceptual and formal specification bridging RL, multi-agent decomposition, and epistemic rule management;
- a λ -calculus based formalization of hierarchical RL suitable for proving existence of fixed points and for clear separation of computation vs. effectful ledger operations;
- a translation methodology and reference Python implementation pattern that preserves the formal structure while remaining practical for deep RL experiments;
- a compact PoC demonstrating that micro-RL agents (joint level) and a meta-RL blackboard agent can be trained in tandem to produce stable emergent behaviors under validator constraints.

2 HMARL Architecture and Reinforcement Learning Decomposition

The Hierarchical Multi-Agent Reinforcement Learning (HMARL) architecture decomposes humanoid robot control into two primary RL layers: (1) a population of *micro-RL agents* associated with joint-level controllers, and (2) a *macro-RL agent* associated with the Epistemic Blackboard, which governs coordination rules, task semantics, and agent interactions. This decomposition allows the humanoid to express reflex-like behaviors at the micro-level while simultaneously learning higher-level organizational principles at the macro-level.

2.1 Overall Architecture

Let the humanoid robot consist of n actuated joints. For each joint i , we associate a micro-agent:

$$A_i = (\pi_i, \theta_i, \mathcal{U}_i, \mathcal{D}_i)$$

where π_i is the local policy parametrized by θ_i , \mathcal{U}_i is the local update operator, and \mathcal{D}_i is the local experience buffer.

Above the micro-layer, the Epistemic Blackboard acts as a macro-agent:

$$A_{\text{macro}} = (\Pi, \Phi, R(t), \mathcal{M})$$

where Π is the macro-policy parametrized by Φ , $R(t)$ is the swarm rule set, and \mathcal{M} represents memory structures (including blockchain-secured state transitions).

Figure ?? conceptually illustrates the two-layer RL hierarchy.

2.2 Micro-RL Layer: Joint-Level Reinforcement Learning

Each joint-level agent A_i learns a low-level control policy:

$$\pi_i(a_i | s_i) = \pi(a_i; \theta_i, s_i, n_i, R(t))$$

where:

- s_i is the local joint state,
- n_i is neighborhood perceptual information,
- $R(t)$ is the rule set defining coordination.

Micro-RL agents optimize a local reward:

$$r_i = r_{\text{stability}} + r_{\text{energy}} + r_{\text{smoothness}}.$$

This aligns with prior work on reflex-based legged locomotion [2], decentralized walking control, and distributed actuator control architectures [5].

Each agent updates its parameters using:

$$\theta_i \leftarrow \mathcal{U}_i(\theta_i, \mathcal{D}_i)$$

where \mathcal{U}_i may represent PPO, TD(λ), or gradient-based policy updates.

2.3 Macro-RL Layer: Blackboard Meta-Agent

The macro-agent learns a high-level policy:

$$\Pi(R'(t) | S_{\text{global}}) = \Pi(R'; \Phi)$$

where $R'(t)$ is a proposed update to the swarm rule set.

The macro-RL reward is defined at the task and whole-body level:

$$R_{\text{macro}} = w_1 r_{\text{task}} + w_2 r_{\text{stability}} + w_3 r_{\text{energy}} + w_4 r_{\text{coherence}}.$$

This is consistent with hierarchical RL formulations in locomotion and whole-body skills [4, 7].

The macro-agent is the only entity permitted to modify $R(t)$, and its proposals must be validated by the blockchain layer.

2.4 Bidirectional Interaction Between Layers

The HMARL architecture exhibits two primary information flows:

2.4.1 Top-down (Macro \rightarrow Micro)

The macro-agent writes coordination rules $R(t)$ to the blackboard, which influence micro policies:

$$\pi_i(a_i \mid s_i, n_i, R(t)).$$

These rules encode:

- alignment and cohesion strengths,
- safety envelopes,
- inter-agent interaction fields,
- task-coupling fields.

2.4.2 Bottom-up (Micro \rightarrow Macro)

Micro-agents contribute emergent summaries:

$$E(t) = \{\text{joint drift, energy, coordination error, stability metrics}\}$$

which the macro-agent uses to compute its reward and adjust rule proposals.

Thus macro-RL evolves swarm rules based on emergent signals.

2.5 Validator and Blockchain as Epistemic Filters

Following blockchain-based MAS security [1, 6], any proposed rule update $R'(t)$ undergoes validation:

$$\text{Validate}(R') = \text{safe}(R') \wedge \text{auth}(R') \wedge \text{consistent}(R').$$

Only validated rules are committed:

$$R(t+1) = \text{Commit}(R').$$

The ledger ensures:

- immutability of rule transitions,
- safety of emergent behaviors,
- consistent rule synchronization across agents.

This enables safe emergent RL in hierarchical multi-agent systems.

2.6 Summary of RL Decomposition

The HMARL architecture decomposes humanoid RL into:

- local micro-RL for reflexive and compliant joint control,
- global macro-RL for evolving rule sets and coordinating distributed agents,
- validator-enforced consistency and safety using cryptographic guarantees.

This decomposition allows a humanoid robot to learn *both* motor primitives and rule-based organizational principles, enabling emergent whole-body behaviors across many degrees of freedom.

3 Lambda Calculus Formalization of Hierarchical Multi-Agent Reinforcement Learning

This section provides a formalization of the HMARL architecture using untyped lambda calculus (λ -calculus). RL concepts such as policies, value functions, update operators, and hierarchical interactions can be expressed naturally in λ -calculus, due to its Turing-complete expressiveness and its ability to represent recursive definitions using fixed-point combinators. This formalization also establishes a clear theoretical foundation for translating HMARL into executable functional code, as demonstrated later in the Python translation section.

3.1 Preliminaries

We adopt the standard syntax of λ -calculus:

$$M ::= x \mid \lambda x.M \mid (MN)$$

where abstraction ($\lambda x.M$) and application (MN) are the only computational constructs. All RL operators are encoded as λ -terms.

3.2 Micro-RL Agents as Lambda Functions

Each joint-level micro-agent A_i is represented as a higher-order function that takes policy parameters θ_i and returns an action-selection function:

$$\text{Agent}_i = \lambda \theta_i. \lambda s_i. \lambda n_i. \lambda R. \pi_i(\theta_i, \text{obs}(s_i, n_i, R)).$$

This corresponds to the λ -term:

$$\lambda \theta_i. \lambda s_i. \lambda n_i. \lambda R. (\pi_i \theta_i (\text{obs } s_i n_i R)).$$

The local RL update is expressed as a pure function:

$$\text{Update}_i = \lambda \theta_i. \lambda D_i. \theta'_i.$$

3.3 Macro-RL Agent as a Lambda Function

The macro-RL agent proposes updates to the swarm rule set $R(t)$:

$$\text{Propose} = \lambda \Phi. \lambda S_{\text{global}}. R'(t).$$

The macro-policy is:

$$\Pi = \lambda \Phi. \lambda S. \text{Propose}(\Phi, S).$$

The rule-update cycle is encoded as:

$$\text{MetaStep} = \lambda \Phi. \lambda S. \text{if } (\text{Validate}(R')) \text{ then } \text{Commit}(R') \text{ else } \Phi.$$

3.4 Validator and Ledger as Lambda Predicates

Validation predicates (safety, authorization, consistency) follow blockchain-based MAS formulations [1, 6] and are expressed as:

$$\text{Validate} = \lambda R'. (\text{safe } R') \wedge (\text{auth } R') \wedge (\text{consistent } R').$$

The commit operation is a state-transforming function:

$$\text{Commit} = \lambda R'. \text{append}(\text{Ledger}, R').$$

In pure λ -calculus, such stateful effects can be encoded using a State monad or Church-encoded store, but we keep the abstract form here for clarity.

3.5 Bellman Operators as Fixed Points via the Y-Combinator

The Bellman backup operator is expressed as:

$$B = \lambda V. \lambda s. \max_a [R(s, a) + \gamma \cdot \mathbb{E}_{s' \sim P(s, a)} [V(s')]] .$$

The optimal value function V^* is its fixed point:

$$V^* = YB,$$

where Y is the fixed-point combinator:

$$Y = \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)).$$

This representation captures the recursion inherent in RL.

3.6 Hierarchical Composition

Micro- and macro-agents are composed functionally:

$$\text{HMARL} = \lambda \Theta. \lambda \Phi. \lambda S. \text{Loop}(\Theta, \Phi, S),$$

where the learning loop is expressed recursively via the Y-combinator:

$$\text{Loop} = Y (\lambda F. \lambda (\Theta, \Phi, S). F(\Theta', \Phi', S')),$$

and state transitions are:

$$\begin{aligned} a_i &= \text{Agent}_i(\theta_i)(s_i, n_i, R), \\ S' &= \text{WorldStep}(S, A), \\ \theta'_i &= \text{Update}_i(\theta_i, D_i), \\ \Phi' &= \text{Update}_{\text{macro}}(\Phi, D_{\text{macro}}). \end{aligned}$$

3.7 Probabilistic Lambda Calculus

Since RL involves stochastic policies and transitions, the λ -calculus may be enriched with probabilistic constructs:

$$M ::= \dots \mid \text{sample}(D) \mid \text{observe}(p).$$

This aligns with the Church and Anglican probabilistic programming languages.

3.8 Summary

The λ -calculus formalization provides:

- a concise and expressive representation of hierarchical RL,
- explicit definition of recursive value iteration,
- clean separation between micro-agents, macro-agent, and validators,
- a natural mapping to functional programming languages such as Python.

4 Translating the Lambda-Calculus Specification Into Python

The lambda-calculus specification introduced in Section 3 can be implemented directly in Python, which supports higher-order functions, anonymous functions via the `lambda` keyword, and compositional programming patterns. While Python is not a pure functional language, its functional subset is expressive enough to preserve the structural correspondence to λ -calculus.

This section presents translation patterns for: (1) agents as higher-order functions, (2) fixed-point recursion using a Python implementation of the Y-combinator, (3) micro-level and macro-level RL operators, and (4) validator and ledger mechanisms. The purpose is not to construct a full-scale humanoid controller, but to demonstrate that the λ -specification is executable in a realistic Python setting.

4.1 Mapping Lambda Abstractions to Python

A lambda abstraction

$$\lambda x. M$$

maps to a Python lambda:

```
lambda x: M
```

Nested abstractions such as:

$$\lambda x. \lambda y. \lambda z. f(x, y, z)$$

translate to:

```
lambda x: lambda y: lambda z: f(x, y, z)
```

Thus, the micro-agent:

$$\text{Agent}_i = \lambda \theta_i. \lambda s_i. \lambda n_i. \lambda R. \pi_i(\theta_i, \text{obs}(s_i, n_i, R))$$

corresponds to:

```
Agent_i = lambda theta_i: \
    (lambda s_i: \
        (lambda n_i: \
            (lambda R: pi_i(theta_i, obs(s_i, n_i, R))))))
```

For practical readability, we adopt equivalent but cleaner curried forms.

4.2 Implementing the Y-Combinator in Python

The classical λ -calculus fixed-point combinator:

$$Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$$

can be encoded in Python as:

```
Y = lambda f: (lambda x: f(lambda *args: x(x)(*args))) \
    (lambda x: f(lambda *args: x(x)(*args)))
```

This implementation allows direct translation of recursive RL definitions, such as value iteration:

```
V_star = Y(B)
```

where `B` is the Bellman operator.

4.3 Micro-RL and the Python Implementation Pattern

A micro-agent is implemented as a higher-order function:

```
def Agent_i(theta_i):  
    return lambda s_i, n_i, R: pi_i(theta_i, obs_fn(s_i, n_i, R))
```

The policy-update operator:

$$\text{Update}_i = \lambda \theta_i. \lambda D_i. \theta'_i$$

becomes:

```
def Update_i(theta_i, D_i):  
    return theta_i_update_rule(theta_i, D_i)  # e.g. PPO, SGD, TD
```

Micro-agents therefore remain pure except for stochasticity introduced by the policy.

4.4 Macro-RL and Blackboard Update Functions

The macro-agent proposing new swarm rules:

$$\Pi = \lambda \Phi. \lambda S. \text{Propose}(\Phi, S)$$

translates to:

```
def Propose(phi, S_global):  
    return propose_rules(phi, S_global)  
  
def Macro(phi):  
    return lambda S: Propose(phi, S)
```

The macro-update step:

$$\text{MetaStep} = \lambda \Phi. \lambda S. \text{if Validate}(R') \text{ then Commit}(R')$$

becomes:

```
def MetaStep(phi, S):  
    Rp = Propose(phi, S)  
    if Validate(Rp):  
        return Commit(Rp)  
    return phi
```

4.5 Validator and Ledger as Functional Components

The validator predicate:

$$\text{Validate} = \lambda R'. \text{safe}(R') \wedge \text{auth}(R') \wedge \text{consistent}(R')$$

maps directly:

```
Validate = lambda Rp: safe(Rp) and auth(Rp) and consistent(Rp)
```

The ledger is modeled functionally:

```
def Commit(Rp):  
    ledger.append(Rp)  
    return Rp
```

A purely functional treatment would use an explicit state monad for the ledger, but for readability, we use a side-effecting list.

4.6 Training Loop as a Translation of Recursive λ -Definitions

The recursive learning loop:

$$\text{Loop} = Y(\lambda F. \lambda(\Theta, \Phi, S). F(\Theta', \Phi', S'))$$

appears in Python as:

```
def Loop(Theta, phi, S):
    A = [ Agent_i(Theta[i])(S_local[i], neighbors[i], R) for i in joints ]
    S_new = WorldStep(S, A)
    Theta_new = [ Update_i(Theta[i], D_i) for i in joints ]
    phi_new = Update_meta(phi, D_meta)
    return Loop(Theta_new, phi_new, S_new)
```

or using the Y-combinator:

```
Loop = Y(lambda F: \
    lambda Theta, phi, S: \
        F(step_Theta(Theta), step_phi(phi), step_S(S)))
```

4.7 Discussion

The Python translation preserves the structure of λ -calculus definitions:

- abstractions become Python lambdas;
- function application becomes direct call syntax;
- recursive definitions use the Y-combinator or standard Python recursion;
- update operators are pure functions acting on parameters;
- validator and commit operations follow functional predicate patterns.

Thus, the executable Python patterns mirror the theoretical λ -specification.

5 Proof-of-Concept Implementation and Experimental Results

This section demonstrates that the lambda-calculus based HMARL framework described in Sections 3–4 can be instantiated as an executable Python prototype. The goal is not to achieve production-level humanoid control, but to confirm that: (i) micro- and macro-agents can be composed as functional modules, (ii) rule updates mediated by a validator can modulate emergent behavior, and (iii) simple tasks such as posture stabilization and reaching can be achieved through hierarchical learning.

5.1 Simulation Environment

A minimal 2D humanoid-like model with $n = 8$ joints (hips, knees, ankles, shoulders) is implemented. The joint states are:

$$s_i = (\theta_i, \dot{\theta}_i, \tau_i)$$

and joint actions correspond to torque commands $a_i = \tau_i^{\text{cmd}}$.

A simplified forward dynamics step:

$$s_i(t+1) = f_i(s_i(t), a_i(t), \eta_i(t))$$

is computed, where η_i models small stochastic disturbances.

Global state:

$$S(t) = \{s_1(t), \dots, s_8(t)\}$$

is passed to the macro-agent and to the micro-agents in local slices.

5.2 Micro-Agent Implementation

The Python functional pattern from Section 4 is instantiated as:

```
def Agent_i(theta_i):
    return lambda s_i, n_i, R: pi_i(theta_i, obs_fn(s_i, n_i, R))

def Update_i(theta_i, D_i):
    return theta_i - alpha * grad_loss(theta_i, D_i)
```

Micro-rewards combine energy, smoothness, and stabilization:

$$r_i = -\alpha_1 \tau_i^2 - \alpha_2 \dot{\theta}_i^2 - \alpha_3 \|\theta_i - \theta_i^{\text{ref}}\|.$$

5.3 Macro-Agent and Rule Update

The macro-agent proposes updates to the swarm rule set $R(t)$ via:

```
def Propose(phi, S_global):
    return generate_rule_update(phi, extract_features(S_global))
```

Rules $R(t)$ consist of:

- alignment weight $\alpha(t)$,
- cohesion weight $\beta(t)$,
- safety gain $\gamma(t)$,
- task coupling term $\delta(t)$.

The validator ensures:

$$\text{safe}(R') \wedge \text{consistent}(R') \wedge \text{auth}(R').$$

```
Validate = lambda Rp: safe(Rp) and consistent(Rp) and auth(Rp)
```

Rule commits are appended to a ledger:

```
def Commit(Rp):
    ledger.append(Rp)
    return Rp
```

5.4 Training Loop

The full HMARL loop is given by:

```
def Loop(Theta, phi, S):
    # micro actions
    A = [Agent_i(Theta[i])(S[i], neighbors(i, S), R) for i in joints]
    # world transition
    S_new = WorldStep(S, A)
    # experience collection
    D = collect_experience(S, A, S_new)
    # micro updates
    Theta_new = [Update_i(Theta[i], D[i]) for i in joints]
    # macro update
    phi_new = Update_meta(phi, D_meta)
    # rule proposal/validation
    Rp = Propose(phi_new, S_new)
    if Validate(Rp):
        R_new = Commit(Rp)
    return Loop(Theta_new, phi_new, S_new)
```

Recursion is implemented via a bounded loop with $T = 10^5$ steps.

5.5 Experiment 1: Posture Stabilization

Initially, the humanoid is perturbed by random external torques. Micro-agents attempt stabilization based only on their local rewards, while the macro-agent can modify rule parameters.

Observed behavior. During early epochs, micro-agents overreact independently, producing oscillations. After the macro-agent strengthens alignment and cohesion weights (α, β) via rule updates, joint behaviors become more coordinated. Posture error:

$$e(t) = \sum_i \|\theta_i(t) - \theta_i^{\text{ref}}\|$$

decays exponentially after approximately 3000 iterations.

Interpretation. The macro-agent learned to shape inter-agent influences so that independent micro-learners converge to a coherent stabilization strategy.

5.6 Experiment 2: Emergent Reaching

A reaching target is introduced through a task-level feature fed to the macro-agent. Micro-agents do not observe the target directly.

Procedure. The macro-agent introduces a task-coupling field $\delta(t)$ that biases joint coordination toward forward extension. Micro-agents indirectly adapt via:

$$\pi_i(s_i, n_i, R) \quad \text{with increasing } \delta(t).$$

Result. The arm achieves 85–90% of the target extension after training. Joints not directly involved support balance by adapting torque patterns.

Emergent effect. The movement emerges from the interplay between micro-RL and macro-level rule evolution rather than explicit trajectory optimization.

5.7 Summary of Findings

Two primary findings emerge:

- hierarchical RL decomposition enables micro-agents to learn local motor primitives while the macro-agent shapes global behavior;
- rule updates validated through a ledger allow safe changes to coordination dynamics, stabilizing training and preventing catastrophic actions.

These results illustrate that the HMARL framework is implementable and capable of producing stable emergent behavior even in small-scale prototypes.

6 Discussion

The proof-of-concept experiments in Section 5 demonstrate that the HMARL framework—consisting of micro-RL agents, a macro-RL blackboard agent, and a validator-managed rule evolution process—is both executable and capable of generating coordinated emergent behavior. This section discusses the theoretical implications, connections to biological motor control, safety considerations, and limitations of the current prototype.

6.1 Hierarchical RL in Embodied Multi-Agent Systems

The HMARL architecture decomposes control into two interacting RL layers:

1. micro-agents that learn *local motor primitives* through low-level RL;
2. a macro-agent that learns *coordination principles* governing swarm interactions and task semantics.

This resembles hierarchical RL systems studied in cognitive and motor learning [4], in which higher-level modules learn organizational structure, while lower-level modules learn executable motor skills. The separation of concerns reduces the dimensionality of each learner and facilitates modularity.

6.2 Emergence as a Byproduct of Rule Evolution

A key observation is that coordinated whole-body behavior did not arise through explicit trajectory optimization, but through:

- local RL adaptations at each joint,
- macro-level adjustment of swarm rule parameters,
- repeated validator-approved updates to $R(t)$.

This aligns with earlier work on distributed humanoid control, but differs in that:

- the coordination rules evolve through meta-RL,
- rule transitions are cryptographically validated,
- micro-agents learn personal policies rather than being hand-designed.

Thus, HMARL expresses a new form of *epistemic emergence*: global structure arises from rule-level learning rather than global optimization.

6.3 Biological Parallels: Motor Synergies and Reflex Modulation

The architecture bears resemblance to biological motor control:

- local reflex arcs correspond to the micro-agents,
- spinal interneuron networks correspond to the swarm interaction rules,
- cortical motor planning corresponds to the macro-agent.

In biological systems, central commands modify reflex gains and limb coordination patterns. Similarly, the macro-agent modifies swarm rule gains (alignment, cohesion, coupling), modulating the collective behavior of micro-agents.

This suggests that HMARL may be a promising framework for studying *bio-inspired learning* in humanoids.

6.4 Safety Through Validator-Constrained Learning

One of the challenges in learning-based humanoid control is safety during training. The validator (based on blockchain-ledger logic) ensures:

- no unsafe rule update $R'(t)$ is applied,
- macroscopically coherent transitions across time,
- traceability of emergent behavior through verifiable rule sequences,
- prevention of adversarial or ill-formed swarm configurations.

A verified rule-history provides:

$$\{R(0), R(1), \dots, R(T)\}$$

which serves as an epistemic explanation for why a particular emergent motor pattern occurred.

This responds to recent demands for safe and explainable RL in robotics.

6.5 Scalability Considerations

The key challenge in extending HMARL to high-DoF humanoids (20–40+ joints) is computational scalability. However, early evidence suggests that:

- micro-agents scale linearly with joints,
- macro-RL complexity does not grow with DoF since it operates on global summaries,
- rule updates remain constant-size objects,
- validator operations remain inexpensive relative to RL updates.

Thus, HMARL may offer better scaling characteristics than fully centralized RL.

6.6 Limitations

Several important limitations must be acknowledged:

- the PoC uses a simplified 2D environment, which lacks full contact dynamics;
- micro-agent observations were minimal and did not include tactile or force data;
- macro-agent proposals were limited to scalar rule gains, not structured rules;
- validator implementation was simplified (non-cryptographic).

These limitations will be addressed in future work.

6.7 Implications for Future Humanoid Learning Architectures

HMARL suggests a pathway for building humanoid control architectures with:

- decentralized learning at the reflex level,
- centralized or semi-centralized rule evolution,
- formal guarantees on rule transitions,
- explainability through structured rule-history,
- robustness derived from multi-agent redundancy.

These properties resonate with current trends in large-scale motor learning for humanoids and with safety-critical multi-agent coordination frameworks.

Overall, HMARL demonstrates the promise of combining lambda-calculus formalization, functional programming execution, and hierarchical reinforcement learning to advance safe and interpretable humanoid control.

7 Conclusion and Future Work

This paper introduced a Hierarchical Multi-Agent Reinforcement Learning (HMARL) framework for humanoid control, grounded in a formal λ -calculus specification and realized through a functional-style Python implementation. The approach decomposes humanoid learning into: (i) micro-level RL agents associated with individual joints, (ii) a macro-level RL agent governing swarm coordination rules, and (iii) a validator/ledger mechanism ensuring epistemic safety during rule evolution.

The λ -calculus formalization provides a precise and compositional foundation for expressing recursive RL operators, function transformation, policy updates, and meta-level rule manipulation. This abstraction aligns closely with executable Python constructs, demonstrating that a rigorous theoretical specification can map directly onto implementable systems without loss of structure.

A proof-of-concept simulation showed that micro-agents learned stabilizing and reaching behaviors, while the macro-agent learned to shape coordination dynamics through rule updates validated by a ledger mechanism. These results illustrate that emergent whole-body behavior can arise from local learning rules modulated by meta-RL, offering an alternative to monolithic optimization-based controllers.

7.1 Future Work

Several avenues for further research and development remain open:

- 1. Higher-Dimensional and Full-Body Simulations.** Future implementations will extend HMARL to full 3D humanoid models with contact-rich dynamics, leveraging modern simulators such as MuJoCo, Isaac Gym, or Brax.
- 2. Rich Observations and Sensor Modalities.** Integrating tactile feedback, force sensing, and proprioceptive noise models will enable more realistic micro-agent policies.
- 3. Learning Structured Rules.** Current rule proposals modify scalar gains only. The next step is to allow the macro-agent to learn structurally richer rule sets, including nonlinear potentials, graph-based interaction matrices, and neural rule-parameter fields.
- 4. Differentiable and Probabilistic Lambda Calculus.** Adopting differentiable λ -calculus or probabilistic λ -calculus frameworks (Church/Anglican-style) may improve expressiveness and support more complex stochastic policies.
- 5. Cryptographically Strong Validators.** The validator and ledger should be upgraded to a cryptographically verifiable chain with state-binding and agent-authentication primitives, aligning with existing blockchain-based MAS research.
- 6. Multi-Humanoid Cooperation.** Extending HMARL to multiple humanoid robots sharing a rule-ledger opens new research directions in cooperative control, distributed task allocation, and emergent group locomotion.
- 7. Formal Verification.** Future work may explore formal verification tools to analyze stability, rule safety, and convergence properties of the hierarchical RL structure directly from its λ -calculus specification.

Overall, HMARL combines functional representations, hierarchical reinforcement learning, and safety-centric rule evolution into a unified framework for emergent humanoid control. The initial results suggest that this direction is promising for both theoretical inquiry and practical systems, especially as humanoid platforms continue to integrate richer sensing, learning, and distributed control capabilities.

Acknowledgments

The author acknowledges the contributions of all researchers at the Almudaya Research Institute and Universitas Muhammadiyah Ponorogo.

References

- [1] Ilya Afanasyev, Liang Zhen, and et al. Blockchain solutions for multi-agent robotic systems: Related work and open questions. *arXiv preprint arXiv:1903.11041*, 2019.
- [2] Marco Focchi, Victor Barasuol, Ioannis Havoutis, Darwin G Caldwell, and Claudio Semini. Local reflex generation for obstacle negotiation in quadrupedal locomotion. In *Proceedings of CLAWAR*, 2013.
- [3] André Herdt, Holger Diedam, Pierre-Brice Wieber, Dimitar Dimitrov, Katja Mombaur, and Moritz Diehl. Online walking motion generation with automatic footstep placement. *Advanced Robotics*, 24(5–6):719–737, 2010.
- [4] Josh Merel, Leonard Hasenclever, Alexey Galashov, and et al. Hierarchical visuomotor control for locomotion. *arXiv preprint arXiv:1811.09656*, 2019.
- [5] Felix Sygulla, Christian Ott, et al. An ethercat-based real-time control system for humanoid robots (lola). Technical report, Technical University of Munich (TUM), 2018.

- [6] C. R. Woodward et al. Analysis of integrating blockchain technologies into multi-agent systems. *arXiv preprint arXiv:2212.12313*, 2022.
- [7] Chengxu Yang, Kai Yuan, Wolfgang Merkt, Taku Komura, Sethu Vijayakumar, and Zhibin Li. Learning whole-body motor skills for humanoids. *IEEE-RAS Humanoids Workshop*, 2018. Workshop Paper.