Multi-agent competitive and cooperative strategy (MACCS): a centralised reinforcement learning approach to teamwork, task allocation and communication in dynamic environments.

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Name: Arunpreet Garcha

Student ID: *

Supervisor: Fahmida Abedin

UNIVERSITY COLLEGE BIRMINGHAM

Department of Computer Science

University College Birmingham

September 11, 2025

Abstract

This project explores multi-agent reinforcement learning (MARL) in a 2D game-style environment where agents can interact and collaborate. It aims to test whether multiple machine learning-driven agents can work collaboratively towards a larger team-focused objective in a dynamic environment.

To support this, the system uses a Python framework, utilising libraries like Pygame and PyTorch to implement a 2D real-time environment in which teams of agents with predefined roles are pitted against each other in pursuit of survival and exploitation. The system brings together key components for assigning tasks, managing behaviour, and sharing information, helping agents act independently while still working well as a team.

Roles are split into HQ, Peacekeeper and Gather. HQs act as centralised commanders, analysing currently available world information and distributing tasks to achieve a larger goal. The roles of Peacekeeper and Gather interact with the world and are responsible for completing their assigned task. As a result, teams are forced to learn how to cooperate in order to survive and succeed.

This project aims to assist research in game AI, robotics, and autonomous multi-agent systems, where hierarchical decision-making, adaptive AI behaviour and coordinated teamwork is required, as these areas in my opinion, essential to scaled collaboration of multiple agents.

Table of Contents

Chapter 1: Introduction

1.1 Introduction

This project explores how Multi-Agent Reinforcement Learning (MARL) can be used in a game-style 2D simulation to develop team-based AI agents that show signs of cooperation and competition. The agents are sorted into teams that need to coordinate to gather resources, defend territory, and outperform rival teams. Each faction (team) is overseen by a centralised AI Commander (HQ), who assigns tasks and manages its global strategy based on currently available intel. Simultaneously, agents on the same team are assigned into fixed roles, either as Gatherers or Peacekeepers and use reinforcement learning algorithms to learn and execute role-specific behaviours that aim to teach these agents to act in a way that promotes collaboration and goal orientation.

1.2 Aim

This work aims to design, implement, and train a game-style multi-agent system in a constrained 2D environment, promoting teamwork to achieve shared objectives.

1.3 Objectives

To accomplish that aim, the project focuses on the following objectives:

1. Designing a game environment facilitating cooperation and competition in a multi-agent setting.
2. To investigate how teams work together and strategies in an environment, and measure the effect of collaboration and decision-making on task performance.
3. Implement and draw upon an existing multi-agent reinforcement learning algorithm to generate coordinated team-based agent behaviour.

1.4 Rationale of the Project

Multi-agent reinforcement learning (MARL) generalizes the ideas of traditional RL by considering a setting in which multiple agents learn to collaborate in a shared environment. This introduces an additional layer of complexity as each agent has to adapt not just to changes in the environment, but also changes in the behaviour of others within its local region. This results in dependencies, common goals, emergent behaviors, both cooperative and competitive (Canese et al., 2021). With the rise of MARL systems being employed in applications such as robotics, autonomous vehicles, and physical AI, the ability to reason under uncertainty, partial observability and limited communication becomes more significant (Zhao, Ju, and HernándezOrallo, 2024; Tran et al., 2025). These are some of the obstacles that make team-building and role coordination difficult, particularly in rapidly evolving and unpredictable environments. This paper summarizes how those problems are addressed by current MARL frameworks. It also contains a summary of the rationale of the MACCS system of this project: a testbed for the abilities of agents to specialise, adapt, and interact as a team in a simulated multi-agent domain.

Chapter 2: Literature Review

2.1 Introduction

Multi-agent reinforcement learning (MARL) is a type of reinforcement learning that multiplies the number of machines learning and trains all of the agents to act in co-ordination by interacting with the environment and other agents (GeeksforGeeks, 2024). Contrary to the standard single-agent RL setting, MARL is complicated by both inter-agent state dependencies, shared spaces (environments), and emergent cooperative/competitive play (Canese et al., 2021). These systems are growing in importance in areas such as robotics, self-driving cars and physical AI, which all require coordination and adaptation (Tran et al., in press 2025).

Team formation with uncertain agent capabilities and dynamic environments are one of the most challenging problems under real-world constraints, i.e. partial observability and limited communication) in multi-agent systems (Zhao, Ju and Hernández-Orallo, 2024).

This literature review seeks to assess to what extent the existing MARL frameworks have addressed these issues. It describes the driving principles behind the MACCS, an environment used to investigate team-based coordination, role specialisation, and adaptive behaviour using multiagent reinforcement learning.

2.2 Multi-Agent Learning Strategies

Early research in Multi-Agent Reinforcement Learning (MARL) evolved by extending single-agent reinforcement learning (RL) techniques to environments with multiple autonomous agents, as in the works of Littman, M.L. (1994). However, introducing many learning agents leads to non-stationarity, coordination complexity, and exponential growth in the joint action and state space (Wong et al., 2021), meaning the environment as viewed by an agent is constantly changing due to its peers also interacting, which affects the learning progress of all. Coordinating many agents becomes more difficult as the number of participants increases. The possible number of actions and situations increases faster than an agent can learn to adapt to. In response, the centralised training with decentralised execution (CTDE) paradigm has appeared as a standard practice, notably through works such as MARL. Lowe et al. (2017) enable agents to learn collaboratively during training but act independently during execution.

Widely used reinforcement learning methods like Proximal Policy Optimisation (PPO) and Deep Q-Networks (DQN) have also been adapted for use in multi-agent systems (Mnih et al., 2015). PPO tends to be a popular choice because it balances stability and performance well, especially in environments where agents need to make continuous decisions. That said, it doesn't really provide built-in tools for handling coordination between multiple agents (Schulman et al., 2017).

DQN works well in simpler, discrete environments but tends to struggle with multi-agent scenarios. Its value-based structure can make it hard to keep up with the changing behaviours of other agents, especially when strategies need to adapt on the fly. To deal with those challenges, newer frameworks like Multi-Agent Actor-Critic (MAAC) have been developed. MAAC uses centralised critics during training, which helps keep learning more stable and makes coordination between agents easier, particularly in tasks where teamwork matters most (Iqbal and Sha, 2018).

Recent reviews have highlighted that as the number of agents increases, fully centralised methods become computationally inefficient, while decentralised approaches often include instability due to non-stationary environments. As Albrecht et al. (2024, p.41) note, “This non-stationarity can lead to a moving target problem because each agent adapts to the policies of other agents whose policies in turn also adapt to changes in other agents, thereby potentially causing cyclic and unstable learning dynamics.”

Hybrid frameworks that mix support for individual learning and team ability, using methods like shared critics, communication protocols, or task allocation, give agents the humanlike building blocks, allowing them to develop in such a way that teamwork is reasonably achievable if implemented correctly.

2.3 Hierarchical vs Flat Multi-Agent Reinforcement Learning

Rather than making use of a high-tech communication means for agents to communicate to reach each other, many recently developed MARL frameworks have achieved better coordination by exploiting hierarchical structure. In these systems, a high-level agent monitors the current state of its sub-ordinate agents, task-allocation systems are employed by higher-level agents to control lower sub-ordinate agents permitting strategic action development throughout the team, and maintaining the independence of agents to operate responsively. In contrast to traditional flat MARL settings, where agents act independently and make decisions based on local observations, hierarchical architectures provide a more organized mechanism. Flat systems can become inefficient, number of agents grows or the environment becomes complex or non-static. Zhou et al. (2021) have already demonstrated that it is beneficial to use a hierarchy in task-driven games such as in strategy games where agents must coordinate closely in order to succeed. Li et al. (2024) extended this by introducing a setup that can assign roles as agents go, a feature which assists in being more organized among the tasks. Mu et al. (2025) took a different approach—they incorporated a hierarchical task network (HTN) into their MARL system, providing agents with more structure for long-term planning, and increasing their coordination as a team.

These all generally fall in line with the Centralised Training with Decentralised Execution (CTDE) concept of Lowe et al. (2017). In short, the agents learn from global information, but once they act, they can act on their own. Combining that with a top-down control system appears to be a smoother running system generally. Splitting up work, getting things done more in sync, using the capabilities of your team more effectively — with the environment is complex or many factors are moving fast, this’ll get a lot of work off the ground that otherwise might bog down. It’s precisely that mix of top-down big-picture planning and locally independent agent behaviour that gives hierarchical MARL the upper hand over the flat setups.

2.4 Task Allocation in Dynamic Environments

Dynamic task assignment in multi-agent systems tasks can be quite important, especially when the goals, threats, or available resources can change rapidly. In hierarchical reinforcement learning, agents must make a decision about what task to take, when to execute and for whom to work, often without full visibility into the full picture. Previous MARL work mostly preferred fully decentralised methods, allowing agents to self-organise with local policies. For instance, Noureddine et al. (2017) used DQN to enable the agent to learn its own task assignment. This provides some flexibility, but it can also be wasteful: Agents might duplicate efforts, or overlook critical tasks, especially when resources are in short supply and threats are constantly changing.

On the contrary, centralised approaches such as that of Liu et al. 's (2021)CoachPlayer do schedule tasks under full observability of environment. While that improves coordination and reaction times among agents, it can hinder things in high-speed situations when agents need to respond rapidly. To overcome this issue, Shibata et al. (2022), which propose a hybrid approach to the sort of "Learning Locally, Communicating Globally," where agents are making local decisions in real time, but sync up to share task info. It worked well on collaborative robotics, and scaled well to different team sizes with little drama.

In total, these studies indicate that methods that are either fully centralised or fully decentralised may not be appropriate. A combination of the two — units that can work independently but come together occasionally to share knowledge — appears better suited to complex, constantly changing environments.

2.5 Competitive vs Cooperative AI Dynamics

In multi-agent reinforcement learning (MARL), and particularly in game-like environments, agents have to live through situations that test their capability to cooperate and compete simultaneously. Cooperative behavior is typically incentivized by reciprocated rewards or shared value functions, which serve to teach agents how to align their actions to achieve common goals (Wang et al., 2020). On the other hand, competitive dynamics bring in direct opposition (e.g. holding territory, denying resources) that compel agents to find the optimal strategies against rival teams (Leibo et al., 2017).

Both types of interaction are handled by the Centralised Training with Decentralised Execution (CTDE) framework, which enables agents to use global information when training, while acting independently at runtime (Lowe et al., 2017). Such a framework successfully maintains coordination while leaving the autotomy of the agents untouched, but it still fails to address the complex, long-term actions in practice.

T. Wang et al. (2020) proposed a model in which agents could change roles according to the context that demanded cooperation or competition. While this premise is sound under the logic of P-AI, it requires that agents be able to pivot roles on demand. In practice I find this unrealistic for most physical systems—agents are created for particular functions and they can't assume new roles unless they are constructed to be that kind of flexible from the beginning. Because of this, I have confidence that with physical systems, training fixed-role design is closer to effective real-world multi-agent

systems - robotics, security operations, military simulations - agents are built for certain tasks and can't reconfiguration capabilities mid-operation unless they are general purpose agents.

2.6 Identifying Research Gaps

Despite steady advances in Multi-Agent Reinforcement Learning (MARL) in recent times, various core challenges that limit the field's ability to support real-world autonomous systems persist.

Scalability is a foundational concern. MARL systems suffer from a combinatorial explosion in joint action spaces and coordination complexity as agents increase. Algorithms such as MAAC (Iqbal and Sha, 2019) and QMIX (Rashid et al., 2018) show strength in small-scale settings but degrade significantly as agent populations grow (Hernandez-Leal, Kartal and Taylor, 2019). Many frameworks lack the hierarchical structure or abstraction to scale learning and coordination across large teams or divisions. This could become vital for control as more RL-enabled agents are deployed into our lives.

Communication poses another persistent limitation. While selective messaging models such as IC3Net (Singh et al., 2019) attempt to reduce redundancy, many MARL systems rely on high-bandwidth, fully connected architectures. This assumption undermines applicability in constrained environments, such as autonomous fleets or remote robotics, where agents must operate with low latency, lossy signals, or intermittent connectivity (Yu et al., 2023). Designing efficient, context-aware communication remains a largely unsolved problem.

Task allocation and role specialisation are also underexplored. Although frameworks like the Coach–player model (Liu et al., 2021) use centralised entities to manage assignments, they often assume agents can adapt roles dynamically. This flexibility is rarely available in real-world systems, where physical agents are constrained by hardware or mission profiles. Few systems combine centralised planning with decentralised execution in settings where roles are fixed, capabilities vary, and information is partial.

While we have reached a point in RL research that has developed some foundational systems, these limitations point to a broader need for hierarchical and hybrid models that blend strategic oversight with adaptive local learning. While existing methods provide partial solutions, they often stop short of modelling multi-layered control or integrating fixed-role agents within competitive and cooperative dynamics akin to the dynamics of real life.

2.7 Proposed Hierarchical MARL Framework

In response to the identified limitations, this project proposes a hierarchical multi-agent reinforcement learning (MARL) framework for scalable, role-constrained, and communication-aware team coordination. The framework is implemented in a procedurally generated 2D simulation, where agents operate within fixed roles—Gatherers and Peacekeepers—under the strategic oversight of a centralised HQ otherwise known as a Commander. The system is built on a hybrid learning architecture. The headquarters (HQ) processes global state data to assign high-level tasks, while local agents use Proximal Policy Optimisation (PPO) to make decisions based on their specific roles. This hierarchical structure enables strategic planning across the faction while preserving individual agent autonomy. To reduce communication bottlenecks, the system was designed to support a hybrid strategy combining global broadcasts with targeted agent-to-agent messaging. Although this feature was only partially implemented, full integration and testing remain planned areas for future development.

Chapter 3: Methodology

3.1 Research Strategy and Philosophical Approach

This project adopts a pragmatist and development-driven approach, prioritising practical experimentation over abstract theorising. Rather than following a purely theoretical model, an iterative system development methodology was adopted, allowing the refinement of real-time multi-agent reinforcement learning components as research and development progressed.

A deductive research strategy guided implementation, wherein system components—such as decentralised agents, fixed-role architectures, and HQ-based coordination—were tested against expected behaviours and adapted based on observed outcomes. This approach enabled continuous hypothesis testing within a dynamic, partially observable environment.

The simulated environment served as a controlled testing ground to evaluate role-based cooperation, decentralised decision-making, and adaptive learning. Agents were embedded in a non-static world with shifting tasks, supporting the emergence of complex behaviours, which were then observed and refined. This methodological approach aligns with existing MARL literature but emphasises strategic hierarchy and real-time environmental variability.

3.2 System Requirements and Design Strategy

3.2.1 Requirements

The system was developed to support decentralised multi-agent learning coordinated through a central strategic entity. The following functional requirements guided the design:

1. Agents must operate with a local state only and act autonomously.
2. Headquarters (HQ) must aggregate global state and assign tasks dynamically.
3. The environment should use a procedurally generated approach to ensure variation across episodes.
4. Agents must use reinforcement learning to improve policies through interaction.
5. The system must support both training and evaluation modes for benchmarking.
6. Coordination and task fulfilment should emerge from agent learning rather than hardcoded routines.

3.2.2 Design Strategy

Architecture: MACCS is constructed as a modular, role-based simulation tool to evaluate teamwork of agents. Although each agent follows its own rules it has access to a shared observation and action space which facilitates coordination and system coherence (Zhou et al., 2021). A central HQ module addresses strategy selection and task allocation under global state, and individual agents take local actions, which is standard way in hierarchical MARL (Mu et al., 2025). The environment embodies both procedural terrain generation and partial observability, which prompt generalisation and robustness in learning (Baker et al., 2019; Albrecht & Stone, 2018).

3.3 Hierarchical Multi-Agent Architecture

The simulation is implemented in a hierarchical style, with the agents working under the direction of a central HQ. This architecture com- plements a local agent decision making with high-level coordination from the HQ, and provide scalable team behaviour in higher-dimension environments (Mu et al., 2025; Zhou et al., 2021). Agents make their decisions individually through their local policy networks according to the environmental information they perceive in their vicinity (for example, threats, resources and tasks). The agent information is emailed to the HQ that receives reports from agents and uses the global state information to determine priority for teams (Lowe et al., 2017).

This is a high level execution, which is preceded by a loop similar with a typical hierarchical MARL system:

Agents move around in the environment and observe entities they are interested in.

They report their findings to the HQ.

The HQ issues assignments depending on the role of the agent, the distance from the objective, and the overall strategy of the faction.

This can serve to support role-based policy specialisation and co-ordination, while maintaining policy diversity among agents. It mirrors real decentralised systems as found with robotics and military operations which have some local autonomy in the framework of a central planner (Albrecht & Stone, 2018).

3.4 Reinforcement Learning Framework

Proximal Policy Optimisation (PPO) was chosen as the primary reinforcement learning algorithm for its balance of stability, sample efficiency, and reliable performance in multi-agent contexts. Introduced by OpenAI (Schulman et al., 2017), PPO's clipped objective function helps prevent overly aggressive policy updates, enabling gradual adaptation in dynamic and partially observable environments. Each agent uses a decentralised actor-critic network: the actor selects actions based on local state and assigned task. At the same time, the critic estimates value functions to support advantage estimation and stabilise learning. Entropy regularisation is included to encourage exploration and prevent premature convergence.

Headquarters (HQs) also employ actor-critic PPO networks at the strategic level, operating over richer input spaces that encode faction-wide metrics, local conditions, and environmental states—the HQ's network outputs both strategy selection and a value estimate, supporting hierarchical policy learning. Although the architecture allows joint optimisation across levels, this project focuses primarily on agent-level policy training. While value-based baselines such as DQN are proposed for future benchmarking, PPO remains the core framework throughout this work.

3.5 Simulation Environment and Training Design

At the start of each episode, the terrain is procedurally generated to create a unique mix of land and water layouts. This randomness adds variety to the environment and helps agents develop more adaptable behaviours during training. Factions are placed at set distances from one another to avoid early conflict and allow for some initial exploration. Each side comes with a central HQ and a blend of gatherers and peackeepers who act as guards. Resources are randomly placed across the usable land and they are what fuel a faction's focus. Agents are run in one of two modes: 1) a training mode where agents learns and improve via reinforcement learning, and 2) an evaluation mode, in which agent behaviours are tested with fixed policies to get performance measures.

3.6 Evaluation Metrics and Testing Process

Agent and faction performance is evaluated using a combination of numerical metrics and observation testing. Progress is tracked through TensorBoard, which provides visual feedback on learning trends and policy stability across episodes (Abadi et al., 2016).

Main evaluation metrics include:

- Win Rate: How often a faction survives longer than its opponents (Samvelyan et al., 2019).
- Resource Efficiency: The amount and speed of gold/food collection over time.
- Survival Time: How long a faction remains active in the simulation.
- Learning Curves: Trends in cumulative reward and policy convergence across episodes (Foerster et al., 2016).
- Task Distribution: How frequently tasks are assigned and completed across runs.
- Action Distribution: A breakdown of agent action frequencies per episode, helping identify behavioural patterns (Papoudakis et al., 2020).

Each episode begins with randomly generated terrain, resource placements, and agent spawn positions to test how well agents generalise across different scenarios—an approach supported by prior work in procedural benchmarking for MARL (Baker et al., 2020).

Emergent challenges—such as communication failures, world misinterpretation, and task prioritisation conflicts—are documented to inform iterative improvement. Model comparisons (e.g., PPO vs DQN) are planned for future development but were not included in the current work.

3.7 Limitations and Ethical Considerations

The system faces several technical limitations. As agent populations scale, computational demands increase, leading to slower training cycles and reduced real-time responsiveness. Additionally, training time remains a constraint due to the complexity of learning in dynamic, multi-agent environments.

From an ethical perspective, the simulation operates in a fictional domain with no real-world data.

Chapter 4: Design & Implementation

4.1 System Overview and Design

The MACCS Simulation is a hierarchical, reinforcement learning-driven environment designed to model teamwork, competition, and adaptive behaviour among autonomous agents. Agents are organised into factions, each guided by a central HQ responsible for delegating tasks, setting strategic priorities, and coordinating the team's actions. The system explores Multi-Agent Reinforcement Learning (MARL) by combining decentralised agent decision-making with centralised strategic oversight.

The simulation unfolds within a procedurally generated 2D world, where factions compete to:

- Gather resources
- Defend territory
- Eliminate rival factions
- Explore the environment

Victory is achieved either through survival (Last Man Standing) or by securing economic dominance via resource collection. Each faction consists of fixed-role agents: gatherers, who focus on resource acquisition, and peacekeepers, who specialise in combat and defence. These agents are directed by their HQ to adapt and respond effectively to dynamic conditions.

The 2D environment is implemented using Pygame, with terrain and resources generated via Perlin noise to introduce variation across episodes and encourage generalisation during training.

Key tools and libraries used include:

- PyTorch for PPO algorithm implementation and network optimisation
- PyGame for rendering
- NumPy, Pandas, and SciPy for data handling
- TensorBoard for training visualisation and logging
- cProfile, pstats, and Python's logging module for profiling and debugging

Training performance was further accelerated through GPU support via CUDA, enabling faster iteration across longer episodes and larger agent populations.

Figure 1 exemplifies how each episode works to support the system.

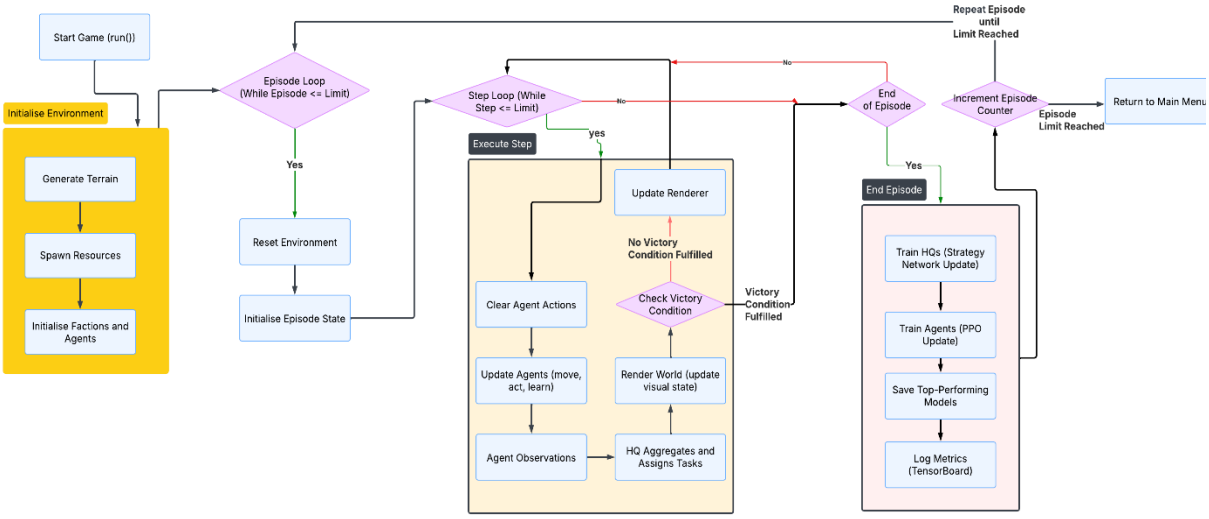


Figure 1 High-level control flow of the MACCS simulation system, showing episode via run() and intra-episode progression through step().

4.2 Implementation

4.2.1 Simulation Environment

The simulated environment developed for the MACCS system is a 2D grid world, implemented principally using Pygame, a Python framework for game development. To promote generalisable learning, the world is procedurally generated at the start of each episode, introducing variations in terrain, resource distribution, and agent placement.

4.2.1.1 World Generation (Perlin noise, terrain types)

To generate the world, the MACCS simulation uses procedural generation using a Perlin noise-based algorithm, implemented in Python using the noise library and structured within a 2D grid array.

Each grid cell is defined by a structured data type that tracks terrain type ('land' or 'water'), occupation status, controlling faction, and any present resource type. This allows for highly flexible environmental dynamics while supporting efficient agent-environment interactions.

The terrain is procedurally generated using a Perlin noise algorithm (noise.pnoise2) configured via parameters such as NOISE_SCALE, NOISE_OCTAVES, NOISE_PERSISTENCE, and NOISE_LACUNARITY, which control terrain frequency, smoothness, and detail granularity. The output is limited using WATER_COVERAGE to classify grid cells as land or water. A land connectivity filter (ensure_connected_land) is applied to preserve navigability by discarding fragmented islands. The grid is generated at a resolution defined by num_cells_width × num_cells_height, scaled to pixel coordinates using CELL_SIZE.

Figure 2 Generated world sample 1

Figure 3 Generated world sample 2

The resulting implementation creates procedurally generated terrain that varies in complexity between runs. This variation introduces unique challenges for each episode, encouraging generalisable agent learning. Agents must adapt to dynamic layouts, navigate around water bodies, and respond to shifting resource distributions—factors that collectively support the evaluation of strategic coordination and task delegation within the MACCS framework.

4.2.1.2 Spawning Agent and Resources

At the start of each episode, the MACCS system randomly positions resources and faction units within a procedurally generated terrain. Resource placement is handled by the generate_resources() function within the ResourceManager class, while the GameManager manages the agent and HQ placement. This procedural variability ensures that each simulation run presents a distinct strategic scenario, encouraging adaptability, exploration, and inter-agent communication.

At environment initialisation, the system uses parameter-driven logic to place resources on available land tiles. Apple Trees are assigned probabilistically based on TREE_DENSITY, while Gold Lumps are generated within clustered "gold zones" controlled by GOLD_ZONE_PROBABILITY and GOLD_SPAWN_DENSITY. This spatial randomness introduces ecological heterogeneity across runs.

Agents are assigned resource collection and management tasks, fostering cooperative behaviour. The simulation currently includes two primary resource types: Gold Lumps and Apple Trees, each contributing differently to faction survival and progression.

4.2.1.2.2 Gold Lumps

Gold Lumps act as a currency source, enabling factions to purchase new units up to their maximum capacity. Each Gold Lump is instantiated with a fixed quantity defined by `GoldLump_base_quantity`. These are finite resources; once depleted through mining interactions, they are permanently removed from the terrain. These are randomly scattered across valid land tiles during world initialisation and can be depleted through agent interaction. Once an instance is depleted, it is immediately removed from the world.

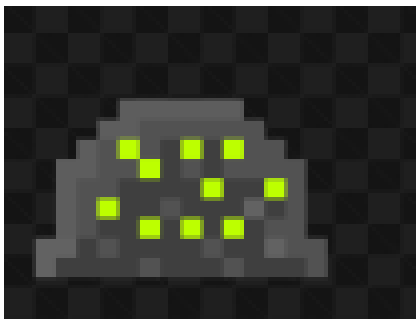


Figure 4 Gold Lump artwork

```python
class GoldLump:
    def __init__(self, x, y, grid_x, grid_y, terrain, resource_manager,quantity=5):
        """
        Initialise a gold lump with a given quantity of gold.
        :param x: X position of the gold lump.
        :param y: Y position of the gold lump.
        :param grid_x: X position on the grid.
        :param grid_y: Y position on the grid.
        :param terrain: Reference to the terrain object for interaction.
        :param quantity: Initial quantity of gold in the lump.
        """
        self.x = x
        self.y = y
        self.grid_x = grid_x
        self.grid_y = grid_y
        self.terrain = terrain
        self.quantity = random.randint(1, 20)  # Set quantity of gold
        self.resource_manager = resource_manager

        # Load and scale the gold image
        self.image = pygame.image.load(GOLD_IMAGE_PATH).convert_alpha()
        image_size = int(SCREEN_HEIGHT * 0.03)  # Scale to 3% of screen height
        self.image = pygame.transform.scale(self.image, (image_size, image_size))
```

Figure 5 Gold Lump Class Code

4.2.1.2.3 Apple Trees

Apple Trees begin with a base number of apples (Apple_Base_quantity) and regenerate one apple every APPLE_REGEN_TIME seconds until they reach maximum capacity. Over-foraging leads to temporary resource removal, adding a renewable yet exhaustible resource dynamic to the simulation. Apples also act as a health source for agents who have taken damage, either from an enemy or if they have suffered a penalty for an invalid action.

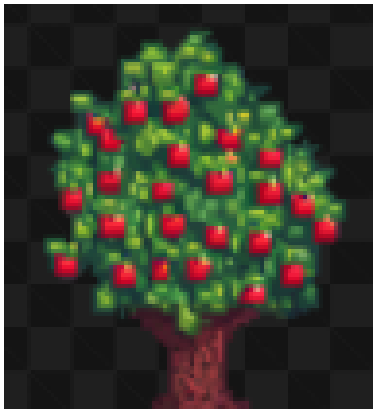


Figure 7 Apple Tree artwork

```python
class AppleTree:
    def __init__(self, x, y, grid_x, grid_y, terrain, resource_manager, quantity=10):
        """
        Initialise an apple tree with a given quantity of apples.
        :param x: X position of the tree.
        :param y: Y position of the tree.
        :param grid_x: X position on the grid.
        :param grid_y: Y position on the grid.
        :param terrain: Reference to the terrain object for interaction.
        :param quantity: Initial number of apples on the tree.
        """
        self.x = x
        self.y = y
        self.grid_x = grid_x
        self.grid_y = grid_y
        self.terrain = terrain
        self.resource_manager = resource_manager
        self.quantity = quantity  # Number of apples

        # Load the sprite sheet
        sprite_sheet = pygame.image.load(TREE_IMAGE_PATH).convert_alpha()

        # Dimensions of the sprite sheet and individual sprites
        sprite_width = sprite_sheet.get_width() // 6  # Assuming 6 frames horizontally
        sprite_height = sprite_sheet.get_height()

        # Extract the final tree (6th frame)
        self.image = sprite_sheet.subsurface(
            pygame.Rect(sprite_width * 5, 0, sprite_width, sprite_height)
        )
```

Figure 6 Apple Tree Class Code

The resource spawning algorithm ensures that no two resources occupy the same tile and avoids terrain deemed untraversable (e.g., water) by skipping any cell that does not match the criteria. This prevents inaccessible or invalid states and promotes fair distribution.

```python
# Iterate through terrain grid
for x in range(grid_width):
    for y in range(grid_height):
        cell = self.terrain.grid[x][y]

        # Ensure the cell is valid for placing a resource
        if cell['type'] == 'land' and not cell['occupied']:
            # Check bounds (prevent placing outside the world grid)
            if not (0 <= x < grid_width and 0 <= y < grid_height):
                continue
```

Figure 8 Resource spawning placement filtering code

4.2.1.2.4 Agent and Faction Spawning

Factions are placed before any agents and a minimum distance apart to avoid immediate conflict and allow for early-stage resource gathering and exploration. Each faction is made up of a single faction HQ (castle), peacekeepers (offensive/defensive roles) and gatherers (agricultural/worker role). HQ_SPAWN_RADIUS ensures that faction headquarters are sufficiently spaced apart to prevent immediate conflict, while HQ_Agent_Spawn_Radius regulates the dispersal radius around HQs for initial agent placement.

Figure 10 Gather Agent artwork

Figure 9 Peacekeeper Agent artwork

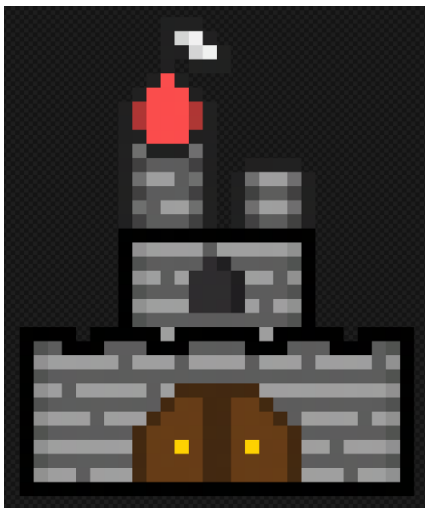


Figure 11 Faction/HQ class artwork

The initialise_agents() function within the GameManager class oversees the placements of faction HQS and agents, ensuring that agents spawn around their respective HQS while keeping some distance from adversarial factions. This allows a grace period at the beginning of each run, allowing some brief exploration before factions spot each other. Each agent is tagged with a unique identifier, faction ID, and preassigned role, which guides its decision-making throughout the episode.

The initialise_agents function is responsible for spawning and initialising faction agents within the simulation grid. Figure 12 shows the first part of this function, covering agent allocation and the initial spawn logic.

```python
# In the GameManager class
def Initialise_agents(self, mode):
    """
    Initialise agents for the game, passing the mode to each agent for appropriate setup.
    :param mode: The mode in which to Initialise the agents ('train' or 'evaluate').
    """
    if hasattr(self, "agents_initialised") and self.agents_initialised:
        print("Agents already initialised. Skipping duplicate initialisation.")
        return
    self.agents_initialised = True
    print("Initialising agents...")

    """Creates agents and assigns them to factions using stratified grid coverage for HQ placement."""
    minimum_distance = HQ_SPAWN_RADIUS  # Minimum distance between HQs
    print("Calculating valid positions...")
    num_factions = len(self.faction_manager.factions)

    # Step 1: Precompute valid HQ positions
    valid_positions = [
        (i, j)
        for i in range(self.terrain.grid.shape[0])
        for j in range(self.terrain.grid.shape[1])
        if self.terrain.grid[i][j]['type'] == 'land' and not self.terrain.grid[i][j]['occupied']
    ]

    # Shuffle positions for randomness
    random.shuffle(valid_positions)

    # Step 2: Select HQ positions with minimum distance enforcement
    selected_positions = []
    for pos in valid_positions:
        if all(math.dist(pos, existing_hq) >= minimum_distance for existing_hq in selected_positions):
            selected_positions.append(pos)
        if len(selected_positions) == num_factions:
            break

    if len(selected_positions) < num_factions:
        print("Failure: Not enough valid positions for all factions' HQs.")
        raise ValueError("Not enough valid positions for all factions' HQs.")

    # Step 3: Assign positions to factions
    for faction, position in zip(self.faction_manager.factions, selected_positions):
        # Convert grid position to pixel coordinates
        base_pixel_x, base_pixel_y = position[0] * CELL_SIZE, position[1] * CELL_SIZE
        faction.home_base["position"] = (base_pixel_x, base_pixel_y)

        # Mark the HQ position in the terrain grid
        self.terrain.grid[position[0]][position[1]]['occupied'] = True
        self.terrain.grid[position[0]][position[1]]['faction'] = faction.id
        self.terrain.grid[position[0]][position[1]]['resource_type'] = None

        print(f"Faction {faction.id} HQ placed at grid {position}, (pixel ({base_pixel_x}, {base_pixel_y}))")
```

Figure 12 Initialise_agents function code pt.1

The continuation of the initialise_agents function is shown in Figure 13, completing the agent initialisation by assigning roles, positions, and team affiliations necessary for strategic coordination.

```
print(f"Faction {faction.id} HQ placed at grid {position}, (pixel ({base_pixel_x}, {base_pixel_y}))")

# Step 4: Spawn agents for the faction using spawn_agent
faction_agents = []
print(f"Spawning agents for faction {faction.id}...")

# Define network_type for each agent (you can modify this logic to dynamically choose the network type)
network_type = "PPOModel"  # Example network type (can also be dynamic based on the agent role or other factor

print("Peacekeeper")
# Pass the required arguments to spawn_agent
for _ in range(INITAL_PEACEKEEPER_COUNT):
    agent = self.spawn_agent(
        base_x=base_pixel_x,
        base_y=base_pixel_y,
        faction=faction,
        agent_class=Peacekeeper,  # Pass the agent class
        state_size=DEF_AGENT_STATE_SIZE,  # Pass the state size
        role_actions=ROLE_ACTIONS_MAP,  # Pass role-specific actions
        communication_system=self.communication_system,  # Pass the communication system
        event_manager=self.event_manager,  # Pass EventManager
        network_type=network_type  # Pass network type for the agent (e.g., PPOModel, DQNModel)
    )
    if agent:
        faction_agents.append(agent)

print("Gatherer")
# Pass the required arguments to spawn_agent
for _ in range(INITAL_GATHERER_COUNT):
    agent = self.spawn_agent(
        base_x=base_pixel_x,
        base_y=base_pixel_y,
        faction=faction,
        agent_class=Gatherer,
        state_size=DEF_AGENT_STATE_SIZE,  # Pass the state size
        role_actions=ROLE_ACTIONS_MAP,  # Pass role-specific actions
        communication_system=self.communication_system,  # Pass the communication system
        event_manager=self.event_manager,  # Pass EventManager
        network_type=network_type  # Pass network type for the agent (e.g., PPOModel, DQNModel)
    )
    if agent:
        faction_agents.append(agent)

# Add the newly spawned agents to the faction and the global agent list
print(f"Faction {faction.id} has {len(faction_agents)} agents.")

# ✅ Clear and reassign to avoid shared references
faction.agents = []  # Make sure each faction starts fresh
faction.agents.extend(faction_agents)  # Assign its agents properly
self.agents.extend(faction_agents)  # Keep global tracking

# Print faction agent counts
peacekeeper_count = sum(1 for agent in faction.agents if isinstance(agent, Peacekeeper))
gatherer_count = sum(1 for agent in faction.agents if isinstance(agent, Gatherer))
print(f"Faction {faction.id} has {peacekeeper_count} Peacekeepers and {gatherer_count} Gatherers.")
```

Figure 13 Initialise_agents function code pt.2

The spawning logic, along with procedurally generated terrain and resources, ensures that no two training sessions are alike. As agents adapt to different resource proximities, terrain layouts, and enemy locations, the implemented MARL framework is tested for generalisability in coordinated decision-making.

4.2.2 Agent Class

4.2.2.1 General Agent Architecture

In the MACCS simulation, all agents inherit from a shared BaseAgent class, which holds common logic for movement, perception, communication, task execution, and health management.

Upon instantiation, each agent is assigned a role—Gatherer or Peacekeeper—and affiliated with a specific faction. Core attributes include:

- Position: Represented by pixel-based (x, y) coordinates relative to the simulation grid.
- Role: Determines the agent's functional scope and permissible actions.
- Health: A bounded numerical value subject to decay through combat or penalties (e.g., movement failure).
- Faction: The team-based identity of the agent, used for coordination and allegiance filtering.
- Policy Model: A reinforcement learning model (PPO or DQN) instantiated at runtime via the initialise_network() function, with the PPO model selected by default due to its implementation stability within the system.

```
def initialise_network(self, network_type_int, state_size, action_size):
    """
    Initialise the network model based on the selected network type (using integer values).
    """
    if network_type_int == NETWORK_TYPE_MAPPING["PPOModel"]:
        print("\033[93m" + f"Initialising PPOModel with state_size={state_size}, action_size={action_size}" + "\033[0m")
        return PPOModel(state_size, action_size)
    elif network_type_int == NETWORK_TYPE_MAPPING["DQNModel"]:
        print("\033[93m" + f"Initialising DQNModel with state_size={state_size}, action_size={action_size}" + "\033[0m")
        return DQNModel(state_size, action_size)
    else:
        raise ValueError(f"Unsupported network type: {network_type_int}")
```

Figure 14 Initialise_network helper function

```
NETWORK_TYPE_MAPPING: dict = {
    "none": 0,
    "PPOModel": 1,
    "DQNModel": 2,
    "HQ_Network": 3,

    }
"""
Mapping of network types to their corresponding interger IDs.
```

Figure 15 Network Mappings Dictionary

Agents operate decentralised, using local policy networks to make independent decisions based on perceived environmental features and task directives issued by their faction's headquarters (HQ). The agent's internal state representation, created by BaseAgent.get_state, comprises the agent's normalised position and health, coordinates of the nearest threat and resource, and a one-hot encoded task vector for its current task.

4.2.2.2 Agent Roles and Behaviours

While BaseAgent provides the core logic for each agent, role-specific behaviours are defined in the Peacekeeper and Gatherer subclasses and executed via the attached AgentBehaviour module. This

module maps network-selected actions to concrete methods such as navigation, combat, or resource collection.

- Gatherers prioritise resource acquisition, like gold and apples, and attempt to move towards the target, using the agent and the target location as an indicator. Once in range, they invoke actions such as mine_gold() or forage_apple().
- Peacekeepers perform threat detection and combat. Using functions like detect_threats() and eliminate_threat(), they neutralise enemy agents and report outcomes to their HQ. If they do not have a task, then they fall back to exploring the world until a threat is found.

4.2.3 Faction Class (HQ)

Each group of agents operates under a centralised command structure implemented via the Faction class, commonly called the Headquarters (HQ) or commander. The HQ is the hierarchical coordinator, responsible for maintaining global awareness, assigning tasks, and managing high-level strategies such as resource gathering, recruitment, and defence. While agents execute tasks independently, their broader behaviour is shaped by faction-level decision-making through task assignment.

The HQ continuously maintains a global state, stored in structured faction dictionaries and updated through agent reporting methods, aggregating real-time incoming data from agent reports. Key metrics include:

- HQ health and coordinates
- Gold and food reserves
- Counts of friendly and enemy agents
- Locations of nearby threats and resources
- Agent density and explored territory

```
self.global_state = {key: None for key in STATE_FEATURES_MAP["global_state"]}
# Populate the initial global state
self.global_state.update({
    "HQ_health": 100,  # Default HQ health
    "gold_balance": 0,  # Starting gold
    "food_balance": 0,  # Starting food
    "resource_count": 0,  # Total resources count
    "threat_count": 0,  # Total threats count
```

Figure 16 Faction global state code

Task Assignment Logic

Agents request tasks from the HQ when idle or upon task completion. The HQ uses role-specific heuristics, such as proximity and resource value, to determine suitable assignments through the Assign_Task() function. Each task is tagged with a unique ID, allowing for coordination among multiple agents when necessary (e.g., joint combat or area exploration). All tasks use a TaskState() enumeration that allows for easy tracking of a task's progress across the system. If a task is detected as a success or a failure, the appropriate measures will be taken, and the agent will be added back to the pool of available agents.

Strategic Planning and Prioritisation

The HQ selects macro-strategies (e.g., COLLECT_GOLD, COLLECT_FOOD, DEFEND_HQ) based on global state indicators. These strategies influence task distribution and can trigger faction-wide actions such as unit recruitment or redeployment. For example, a low food reserve triggers resource-based tasks to be distributed, while enemy proximity shifts the system into a defensive stance.

Figure 4.3 illustrates the strategic selection and task assignment loop that the headquarters (HQ) employs to adapt to global conditions.

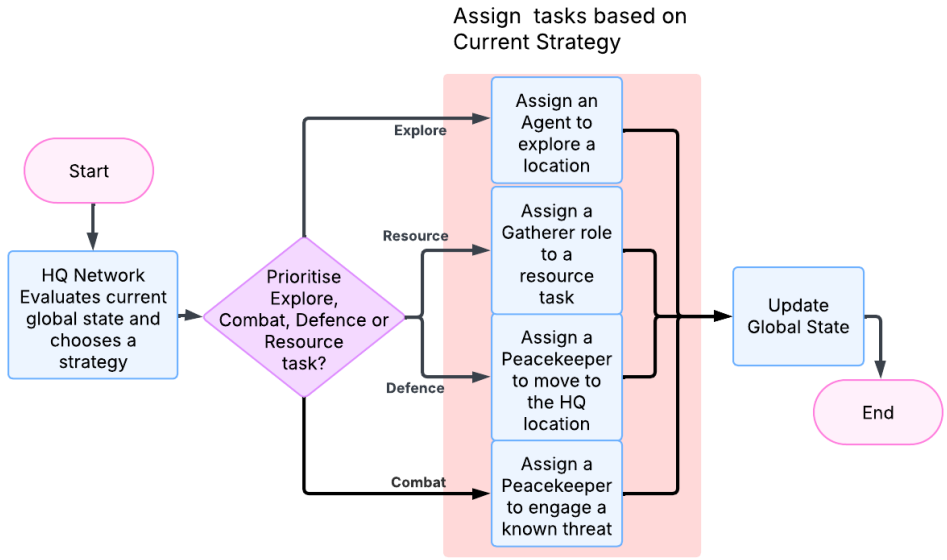


Figure 17 HQ Strategic decision Flow in MACCS, showing the loop used by the HQs to decide current strategy and assign relevant tasks.

Figure 18 illustrates the decision-making flow the Headquarters (HQ) uses to select strategic actions based on observed global conditions.

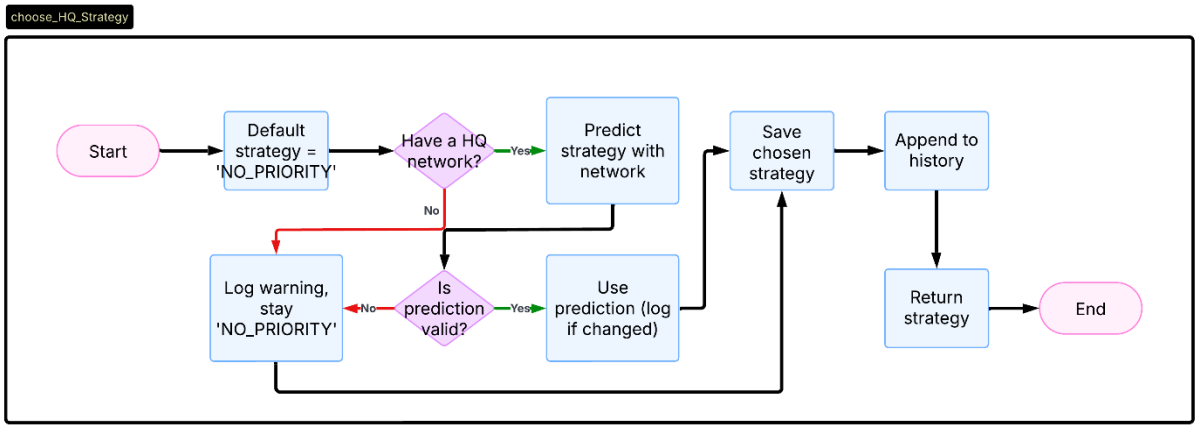


Figure 18 Choose_HQ_Strategy logic diagram

The corresponding implementation of this decision-making logic is provided in Figure 19, detailing the choose_HQ_Strategy function as implemented within the Faction class.

```python
def choose_HQ_Strategy(self):
    """
    HQ chooses a strategy using its neural network (or fallback),
    and stores it as the current_strategy.
    """
    if utils_config.ENABLE_LOGGING:
        logger.log_msg(
            f"[HQ STRATEGY] Faction {self.id} requesting decision from HQ network...",
            level=logging.INFO)

    strategy = "NO_PRIORITY"  # Default
    previous_strategy = self.current_strategy

    if hasattr(self, "network") and self.network:
        predicted = self.network.predict_strategy(self.global_state)

        if predicted in utils_config.HQ_STRATEGY_OPTIONS:
            strategy = predicted
            if strategy != previous_strategy:
                logger.log_msg(
                    f"[HQ STRATEGY] Faction {self.id} network picked different strategy: {strategy}",
                    level=logging.INFO)
            else:
                logger.log_msg(
                    f"[HQ STRATEGY] Faction {self.id} network continued with strategy: {strategy}",
                    level=logging.INFO)
        else:
            logger.log_msg(
                f"[HQ STRATEGY] Invalid strategy returned: {predicted}. Defaulting to NO_PRIORITY.",
                level=logging.WARNING)
    else:
        logger.log_msg(
            f"[HQ STRATEGY] No HQ network found. Falling back to NO_PRIORITY.",
            level=logging.WARNING)

    self.current_strategy = strategy
    self.strategy_history.append(strategy)  # Track it
    return strategy
```

Figure 19 choose_HQ_Strategy code

Figure 20 illustrates the internal decision-making flow used by the HQ's neural network during strategic prediction. This diagram outlines the key steps, from encoding the current global state to selecting the most probable macro-strategy based on model outputs.

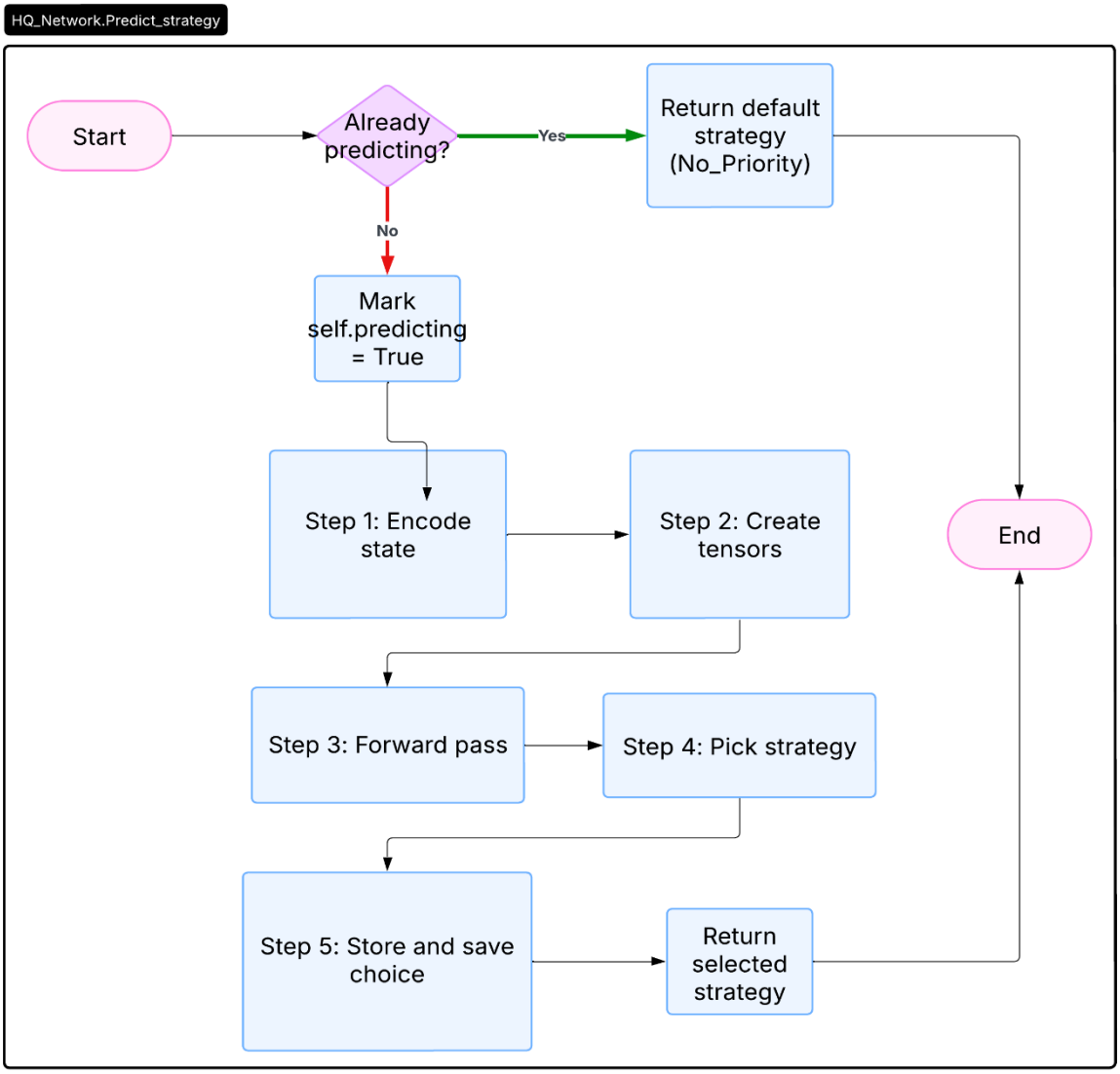


Figure 20 HQ_Network.Predict_strategy logical diagram

The corresponding implementation of this neural decision process is shown in Figure 21, where the predict_strategy function processes structured state inputs, performs a forward model inference, and Selects the highest-scoring strategy.

```python
def predict_strategy(self, global_state: dict) -> str:
    """
    Given the current global state, returns the best strategy label.
    """
    if hasattr(self, 'predicting'):
        return self.strategy_labels[0]
    self.predicting = True
    try:
        # 1. Extract structured input parts
        state, role, local_state, global_state_vec = self.encode_state_parts()

        logger.log_msg(
            f"[INFO] Predicting strategy for global state: {global_state}",
            level=logging.DEBUG)
        logger.log_msg(
            f"[DEBUG] Encoded state vector: {state}",
            level=logging.DEBUG)

        # 2. Convert to tensors and move to correct device
        device = self.device
        state_tensor = torch.tensor(
            state, dtype=torch.float32, device=device).unsqueeze(0)
        role_tensor = torch.tensor(
            role, dtype=torch.float32, device=device).unsqueeze(0)
        local_tensor = torch.tensor(
            local_state, dtype=torch.float32, device=device).unsqueeze(0)
        global_tensor = torch.tensor(
            global_state_vec, dtype=torch.float32, device=device).unsqueeze(0)

        # 3. Forward pass
        with torch.no_grad():
            logits, _ = self.forward(
                state_tensor, role_tensor, local_tensor, global_tensor)

        # Log raw logits
        logits_list = logits.squeeze(0).tolist()
        for i, value in enumerate(logits_list):
            logger.log_msg(
                f"[LOGITS] {self.strategy_labels[i]}: {value:.4f}",
                level=logging.INFO)

        # 4. Select strategy with highest probability
        action_index = torch.argmax(logits).item()
        selected_strategy = self.strategy_labels[action_index]

        # 5. Store memory
        self.add_memory(state, role, local_state,
                        global_state_vec, action_index)

        logger.log_msg(
            f"[HQ STRATEGY] Selected: {selected_strategy} (index: {action_index})",
            level=logging.INFO)

        return selected_strategy
    finally:
        delattr(self, 'predicting')
```

Figure 21 HQ_Network.predict_strategy code

Figure 22 illustrates the execution logic for strategic actions decided by the HQ. This diagram captures how the HQ validates an intended action, checks necessary preconditions (such as available gold or nearby threats), and handles fallback behaviour if the action cannot be completed.

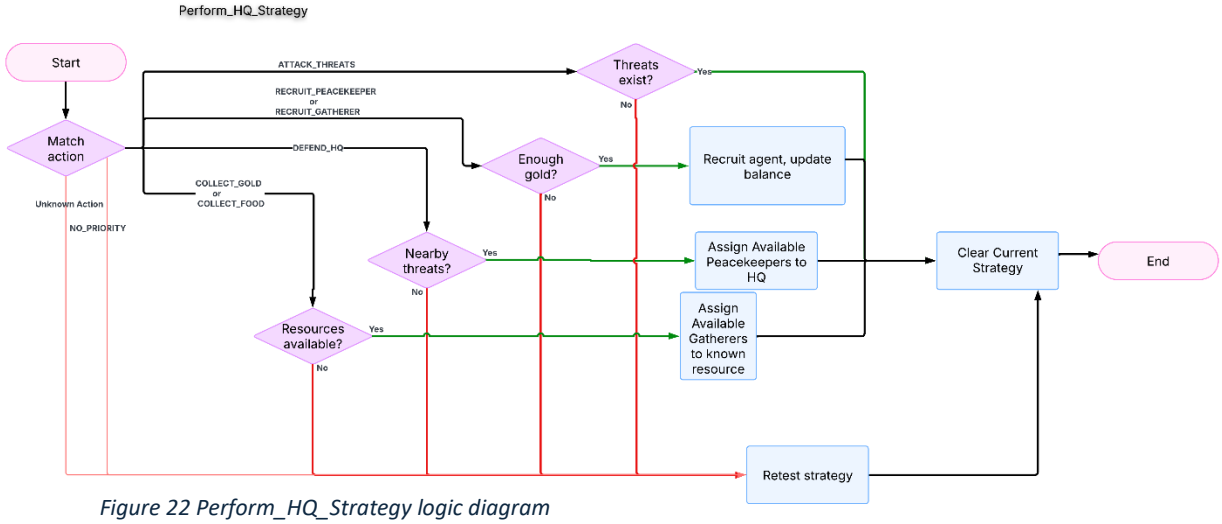


Figure 22 Perform_HQ_Strategy logic diagram

The implementation of this strategy execution flow is shown here in Figure 23, where the perform_HQ_Strategy function manages action validation, conditional branching for various strategic objectives, and dynamic strategy reassessment if the chosen action fails.

```
def perform_HQ_Strategy(self, action):
    """
    HQ executes the chosen strategic action if valid.
    If invalid, it re-evaluates strategy using the HQ network.
    """
    print(
        f"[HQ_STRATEGY] Faction {self.id} perform_HQ_Strategy - Trying to  {action}")
    if utils_config.ENABLE_LOGGING:
        logger.log_msg(
            f"[HQ EXECUTE] Faction {self.id} attempting strategy: {action}",
            level=logging.INFO)

    def retest_strategy():
        new_action = self.choose_HQ_Strategy()
        if new_action != action:
            logger.log_msg(
                f"[HQ RETEST] Strategy '{action}' invalid. Retesting and switching to '{new_action}'",
                level=logging.WARNING)
            self.perform_HQ_Strategy(new_action)
        logger.log_msg(
            f"[HQ EXECUTE] Faction {self.id} executing updated strategy: {new_action}",
            level=logging.INFO)

    # ========== STRATEGY: Recruit Peacekeeper ==========
    if action == "RECRUIT_PEACEKEEPER":
        Agent_cost = utils_config.Gold_Cost_for_Agent
        if self.gold_balance >= Agent_cost:
            current_balance = self.gold_balance
            new_balance = current_balance - Agent_cost
            self.gold_balance = new_balance
            self.recruit_agent("peacekeeper")
            print(f"Faction {self.id} bought a Peacekeeper")
        else:
            logger.log_msg(
                f"[HQ EXECUTE] Not enough gold to recruit peacekeeper.",
                level=logging.WARNING)
            self.current_strategy = None
            return retest_strategy()

    # ========== STRATEGY: Recruit Gatherer ==========
    elif action == "RECRUIT_GATHERER":
        Agent_cost = utils_config.Gold_Cost_for_Agent
        if self.gold_balance >= Agent_cost:
            current_balance = self.gold_balance
            new_balance = current_balance - Agent_cost
            self.gold_balance = new_balance
            self.recruit_agent("gatherer")
            print(f"Faction {self.id} bought a Gatherer")
        else:
            logger.log_msg(
                f"[HQ EXECUTE] Not enough gold to recruit gatherer.",
                level=logging.WARNING)
            self.current_strategy = None
            return retest_strategy()
```

Figure 23 faction class. perform_HQ_Strategy code

4.2.5 Communication System

The communication system in MACCS enables decentralised coordination between agents and their respective headquarters (HQ). While minimal in its current implementation, it establishes the foundational mechanisms for information sharing and collaborative behaviour across agents of the same faction.

The system supports two communication modes:

- Broadcasts: Messages (e.g., “resource_found”, “enemy_spotted”) are sent to all allied agents except the sender. These facilitate the rapid dissemination of strategic information without central mediation.
- Direct messages: Agents can target specific peers to share tactical data (e.g., “help_request”, “threat_info”). These interactions populate local memory structures used for short-term decision-making.

However, in the current version of MACCS, most agent communication occurs implicitly through updates to a centralised global state maintained by the HQ. While the preliminary implementation of the Communication System is in place, it is not yet actively utilised.

In future extensions, agents could leverage learned communication policies to dynamically select when and whom to communicate with, enhancing scalability and improving coordination efficiency in larger agent populations.

4.2.6 Reinforcement Learning Integration

4.2.6.1 Machine Learning Architecture

Proximal Policy Optimisation (PPO) is the primary architecture type for the project's machine learning components, powered by PyTorch. Agent inputs include local features (position, health), contextual cues (nearest threat/resource), and a one-hot task vector.

The PPO Model uses an actor-critic setup, supporting on-policy learning via Generalised Advantage Estimation (GAE). It stores trajectories of states, actions, log probabilities, rewards, and values during each episode. Policy updates use a clipped surrogate loss function, balancing learning speed and stability, with entropy bonuses added to encourage exploration.

While a DQN architecture was planned as an addition to the project, PPO was chosen as the default because it can recover from weak policy updates that would otherwise be negative and hinder training. The PyTorch library was selected due to its GPU-accelerated training functionalities. These two frameworks promise faster training and the ability to manage larger agent populations.

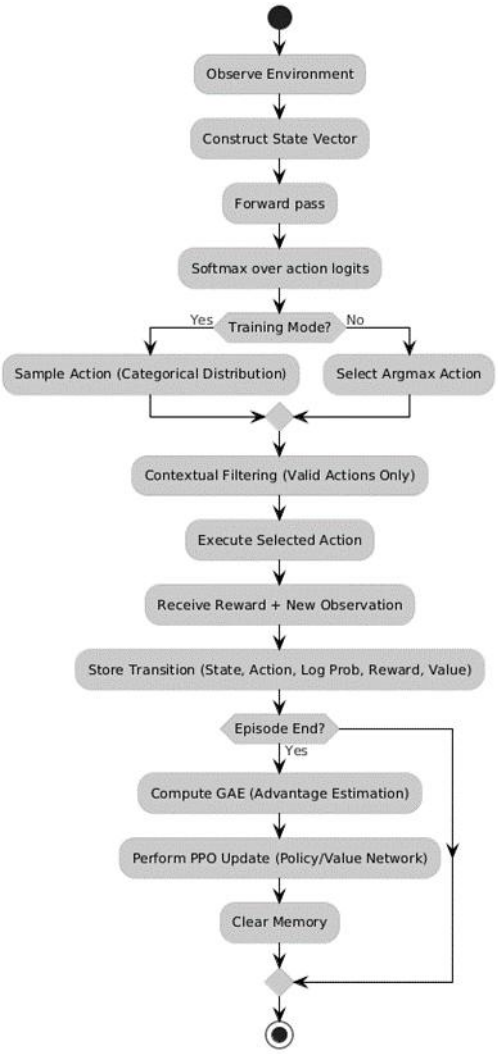


Figure 24 PPO network flow

4.2.6.2 Training vs Evaluation Mode

Agents alternate between training and evaluation modes. During training, they collect experience tuples across episodes, storing state-action-reward trajectories in a local buffer. These are used post-episode to update network weights.

At episode end, the PPO training pipeline computes:

- Discounted returns
- State and global advantages (for actor-critic learning)
- Clipped PPO losses

This cycle is followed by memory clearance to prevent stale data contamination in future updates.

In evaluation mode, learning is disabled. Agents select actions via inference only, enabling performance benchmarking without exploration-induced variance. This mode is critical for measuring policy generalisation and robustness.

4.2.6.3 Reward Structure

MACCS uses a role-sensitive reward system to shape learning agents that receive continuous feedback based on task progression, success, and efficiency. Rewards are issued according to the following structure:

- Success: Base reward (+1.0), with additional bonuses depending on task type:
- Gathering success: +0.25
- Threat elimination success: +0.5
- Failure: Base penalty (−1.0), with minor scaling based on wasted distance.
- Ongoing Progress: Small positive shaping (+0.3 minus a distance penalty) to encourage moving towards objectives.
- Invalid Action: Moderate penalty (−0.5) to discourage random or redundant actions.

Agents receive both individual-level rewards and optional global critic signals from the HQ, enabling the learning of decentralised policies that align with broader faction strategy.

All reward signals are logged to TensorBoard, enabling visualisation of agent performance over time and supporting empirical tuning of learning dynamics.

Chapter 5: Analysis and Findings

A series of structured tests were conducted under varying environmental and training conditions to evaluate the performance of the MACCS system. The primary objective of this analysis was to assess how well the system fulfils decentralised multi-agent tasks and how efficiently it converges on stable, role-specialised behaviours.

Key parameters were changed across test runs, including:

- Environment randomness (static vs. procedurally generated terrain)
- Agent composition (variations in gatherer/peacekeeper ratio)
- Faction density (number of opposing teams)
- Training duration (total steps and episodes)

Evaluation was based on two primary criteria:

1. Task fulfilment is defined as the rate at which agents complete assigned tasks (e.g., resource gathering and threat elimination) as a measure of functional competence and strategic alignment.
2. Operational efficiency – measured by average episode reward and convergence rate across training runs, indicating learning stability, adaptability, and coordination under varying environmental constraints.

5.1 Test Configurations and Summary Results

The table below summarises the setup and results of the seven test scenarios used during evaluation. These tests varied factors like environment randomness, number of factions, agent roles, and episode length to see how the system performed under different strategic conditions. Each test was analysed using average rewards by role, task success rates, and other detailed metrics—full breakdowns can be found in the appendices (Appendix A.1–A.7). Notes in the table link each test to its corresponding appendix for more detail.

Test ID	Environment Randomisation	Factions	Agent Composition	Steps	Episodes	Average Reward	Task Success Rate	Notes
1	No	2	2 Gatherers, 2 Peacekeepers	5000	10	HQ: 14.22 Gatherer: 22.1 Peacekeeper: -43.73	9.6%	Baseline static world See Appendix A.1
2	Yes	3	3 Gatherers, 3 Peacekeepers	5000	10	HQ: 13.1 Gatherer: 18.97 Peacekeeper: 7.58	10.0%	Increased competition See Appendix A.2
3	Yes	3	1 Gatherer, 3 Peacekeepers	5000	10	HQ: 9.23 Gatherer: -60.42 Peacekeeper: -25.67	4.3%	Peacekeeper majority See Appendix A.3
4	Yes	3	3 Gatherers, 1 Peacekeeper	5000	10	HQ: 3.65 Gatherer: -224.37 Peacekeeper: -80.75	7.1%	Gatherer majority See Appendix A.4
5	Yes	3	2 Gatherers, 2 Peacekeepers	15000	30	HQ: 16.99 Gatherer: 97.71 Peacekeeper: -68.78	7.6%	Resource-heavy config See Appendix A.5
6	No	2	2 Gatherers, 2 Peacekeepers	15000	30	HQ: 9.82 Gatherer: 28.68 Peacekeeper: -34.98	12.1%	Default settings, longer-run See Appendix A.6
7	Yes	3	2 Gatherers, 2 Peacekeepers	20000	30	HQ: 10.5 Gatherer: -348.44 Peacekeeper: -385.83	20.2%	Longer run, increased factions, default settings See Appendix A.7

5.2 – Results and Observations

This section evaluates the outcomes of seven test configurations (Appendix A.1–A.6), each designed to measure agent performance, coordination efficiency, and strategic adaptation in varied multi-agent settings. The core metrics assessed include average role reward, task success rate, and headquarters (HQ) strategy distribution.

5.2.1 - Role-Specific Performance Trends

Headquarters (HQ) had somewhat stable reward curves across various test settings, which correspond to its capability of strategically commanding environments (Mu et al., 2025; Zhou et al., 2021). In stiffer or resource-rich environments - Test 1 (static map and two factions) and Test 5 (high resources) - the HQ obtained average rewards of 14.22 and 16.99 respectively (in Appendix A.1, A.5). These results suggest that centralised planning is not always better and works best when the agent roles are largely balanced and the environment is cooperative. Even in a more dynamic or competitive setting as in Test 2 and Test 7, the HQ obtained moderate performance (13.1 and 7.84 scores, respectively) (Appendix A.2, A.7). The lower performance in Test 7 probably ties to the phenomena of task congestion and overloading under pressure—an issue studied extensively in highly-populated MARL domain (Papoudakis et al., 2021).

Gatherers were the most sensitive to environmental resource conditions, as noted in previous studies on resource-driven multi-agent cooperation (Wang et al., 2020). Their highest average reward was in Test 5 (97.71), where the increased availability of apples and gold enabled consistent task completion without significant competition (Appendix A.5). Performance dropped sharply in congested or imbalanced setups. In Test 4 (gatherer-heavy), their average reward fell to -224.37, a result of overpopulation, redundancy, and poor defensive coverage (Appendix A.4). Similar underperformance appeared in Test 6 (-197.29) and Test 7 (-288.13), where faction density and role competition limited Gatherers' freedom of action (Appendix A.6, A.7). More stable performance was seen in Test 2 (18.97) and Test 1 (22.1), where HQ oversight supported more structured task distribution.

Peacekeepers showed the highest variation in performance across scenarios—a trait also noted in roles tied to reactive event-driven engagement (Leibo et al., 2017). They underperformed in cooperative, low-conflict environments like Tests 1 and 5, with average rewards of -43.73 and -68.78 respectively, due to limited opportunities for elimination or protection-based tasks (Appendix A.1, A.5). Their strength was clearest in competitive scenarios. For example, in Test 2, Faction 2's Peacekeepers scored an average of 60.55, aligning with increased defensive and combat opportunities (Appendix A.2). Lower or inconsistent scores in extended runs like Test 6 (4.63) and Test 7 (-52.39) were likely due to a lack of concentrated threats. As expected, Peacekeepers were most effective when deployed in volatile environments where conflict was frequent or escalating.

5.2.2 - Task Success Rates and Efficiency

Task success rates varied considerably depending on environmental complexity, factional competition, and team composition. In the controlled baseline setup (Test 1), agents achieved a 20.21% success rate (Appendix A.1), aided by static terrain and balanced roles. This remained the benchmark for coordinated performance. On the other hand, Test 3 (Peacekeeper-heavy) recorded

the lowest rate at just 4.21%, illustrating how overspecialisation can prevent task execution, especially when resource-focused roles are underrepresented (Appendix A.3).

Test 4, where Gatherers outnumbered Peacekeepers, fared only slightly better at 7.1% (Appendix A.4). This further supports the observation that role imbalance, regardless of which role dominates, consistently limits effectiveness. Even in resource-rich Test 5, success reached only 7.6%, reinforcing that environmental abundance alone cannot offset poor coordination (Appendix A.5).

The most recent test, Test 7, achieved a task success rate of 6.2%, despite its longer episode span and increased agent count. While the extended duration should allow for more adaptive behaviour, adding a third faction and greater agent density appeared to create coordination bottlenecks rather than improvements. This reinforces findings from earlier tests that adding complexity, without improving task assignment or inter-role coordination, tends to hinder rather than help (Appendix A.7).

Many tasks remained "ongoing" across all tests, often above 94%, indicating underlying systemic limitations. These may be attributed to:

- Unreachable or poorly selected task targets
- Agents failing to complete assignments due to policy deadlocks or looping behaviour
- Lack of dynamic task reassessment or cancellation logic
- Over-assignment to the same task site causes crowding and delay.

These issues suggest that low completion rates are not solely due to learning failures but also to shortcomings in environment setup or execution logic. This mirrors observations from Baker et al. (2019), where emergent multi-agent systems occasionally lock into inefficient behavioural loops. Addressing task lifecycle design—especially reassignment strategies and conflict resolution—could substantially improve success rates across all configurations.

5.2.3 - Strategic Behaviour and HQ Trends

One important observation made from the assessment was the evolution of HQ strategy decisions. In earlier trials like Test 1, HQs concentrated mostly on resource-collection, and did so often with COLLECT_GOLD and COLLECT_FOOD. However, in cases of higher intra-faction conflict (Tests 2, 3, and 6) they turned their attention to more defensive and aggressive options, such as DEFEND_HQ, RECRUIT_PEACEKEEPER, and ATTACK_THREATS. In Test 6, though, HQs issued DEFEND_HQ in excess of 30,000 times (see Appendix A.6), evidence to an specific evolution towards the increase in perceived enemy threat.

Test 5 provided a good case of context-based judgment. Headquarter of Faction 1 (HQ 1) would apply COLLECT_FOOD in 53.3% of decisions, which is comparable to the high food values and dense tree coverage of that environment (Appendix A.5). That is, the HQs were learning to adapt strategies to the layout and objectives of each scenario.

These results are consistent with prior efforts in multi-agent learning. For example, Baker et al. (2019) and Li et al. (2024) also notice that as environment difficulty increases, agent strategy will become more complex. As in Baker's study of hide-and-seek, where agents adopted strategies in response to changing conditions, HQs in this work evolved from naive collection tactics to context-aware, role-focused planning.

5.2.4 - Composition Balance and Team Coordination

Team composition had a major impact on performance across the tests. The most balanced setup—Test 1, with 2 Gatherers and 2 Peacekeepers per faction—delivered steady rewards across all roles and showed minimal overlap in task behaviour. It also achieved a task success rate of 20.21% (Appendix A.1), suggesting that role balance contributed to smoother coordination. By contrast, Test 4 featured a gatherer-heavy setup (3 Gatherers and 1 Peacekeeper per faction), which led to clear inefficiencies. Gatherers frequently clashed over the same resources, while the lone Peacekeeper couldn't effectively cover or support the team (Appendix A.4).

Test 6 reused the same balanced 2–2 role split but extended the run duration. After a slow start, agents showed signs of adapting, with modest overall success (12.1%) and slightly positive Peacekeeper rewards—suggesting that coordination patterns may improve given more learning time (Appendix A.6). However, results from Test 7 were unexpectedly poor. Despite using the same 2–2 role mix and extended duration, the scenario saw only a 5.3% success rate, with 94.7% of tasks left incomplete. All roles—including HQ, Gatherers, and Peacekeepers—recorded negative rewards (Appendix A.7). The large number of agents and factions likely introduced excessive competition and overcrowding, which made it harder for the HQ to assign viable tasks and limited agent mobility. This highlights that even well-balanced teams can underperform if coordination breaks down or the environment becomes too congested. These findings are consistent with Wang et al. (2021), who noted that predefined role specialisation only works well when paired with effective coordination. Overall, the results point to heterogeneous teams with clear roles performing best when supported by strong strategy and enough space to operate.

5.2.5 - Comparative Insight and Adaptation

The MARL framework developed in this project, featuring a central HQ managing fixed-role agents—closely follows the hierarchical control structures seen in recent research. Studies like Zhou et al. (2021) and Li et al. (2024) have shown that command-based systems improve coordination and task management, particularly in fast-moving or competitive environments. In this system, the HQ adjusted its strategy based on changing conditions—for instance, shifting from resource gathering to defence, which suggests that hierarchical control can outperform flat, reactive models when complexity increases.

These findings also support Mu et al. (2025), who found that hierarchical task networks enhance efficiency by distributing decision-making across multiple levels. While agents in this project didn't communicate directly, the HQ's task assignments acted as an indirect form of coordination. This helped minimise redundancy and improved overall efficiency, especially in structured scenarios like Test 1 and resource-rich environments like Test 5. Even in high-conflict tests such as Test 7, the HQ kept agent

behaviour relatively coordinated, reinforcing the importance of central oversight. This ability to shift strategies as needed also mirrors the adaptive behaviours described by Baker et al. (2019), where agents evolved to match increasing environmental complexity.

Chapter 6: Conclusions and Recommendations

This effort developed and evaluated a Multi-Agent Reinforcement Learning (MARL) system for effective role-based teamwork in a dynamic, partially observable environment. It applied a hierarchical structure whereby a central HQ led Peacekeepers and Gatherers in order to mimic coordination and labour division (Mu et al., 2025; Zhou et al., 2021). Seven experiments were conducted to test the system in downtimes, team compositions and resource situations. Performance was context-dependent: HQ agents were strongest in structured and long scenarios, Gatherers in resource-rich maps, Peacekeepers in conflict-heavy trials confirming previous observations about task-specific performance (Leibo et al., 2017; Papoudakis et al., 2021). Teams with defined roles, more homogeneous construction, and functioning balance were most reliable.

There were also limitations. A lot of the tasks remained work-in-progress because of the suboptimal end conditions as well as delays in reevaluation (Shibata et al., 2022). Fixed roles in episodes caused reduced adaptability as also observed in [Iqbal and Sha, 2018] for decentralised MARL.

Recommendations for the Future

1. Task Lifecycle Management

   Agents sometimes got blocked in tasks because they were missing or the completion logic was delayed. The overall responsiveness of the system might also benefit from tweaks like task timeouts, revaluation mid-task or periodical validation checks (Shibata et al., 2022).

2. Dynamic Role Rebalancing

   Allowing agents to take on different roles halfway through an episode (e.g., dormant Peacekeepers collecting resources) could lead to increased adaptability and efficiency in dynamic environments (Wang et al., 2020).

3. Smarter spawning and pathfinder

   There were times where terrible spawn points and simple navigation hindered performance. Adding spawn validation and switching to more sophisticated pathfinders (e.g A*) may help in accomplishing goals more efficiently (Samvelyan et al., 2019).

4. Inter-Agent Communication

   Although agents can gain information about local allies through local state, they are still running with no global awareness of team status. This consonant with making lightweight communication— e.g., status broadcasts and message-passing—, which could help agents better coordinate among each other, and make it easier to agents prioritise more efficiently, as observed in similar multi-agent systems (Iqbal & Sha, 2018).

5. EXCHANGES BETWEEN FACTIONS AND NEGOTIATIONS

   Including rudimentary diplomacy (e.g. temporary alliances, trading, ceasefires) could make emergent behaviour more complex, and strategy more adaptable. Such dynamics have also been studied in massive-scale MARL experiments (Baker et al., 2020).

In conclusion, this work demonstrates that hierarchical, role-based MARL systems can exhibit coordinated and adaptive behaviour in dynamic, complex environments. The current results are promising, however, with enhancements to task logic, flexible role assignment, communication and strategic interaction, the framework is closer to practical application in domains such as autonomous logistics, exploration, defence, and swarm robotics.

Chapter 7: Project Reflection

Working on this project has been both rewarding and challenging. It is the most extensive codebase I have worked on, with many components custom-built to suit the project's multi-agent simulation demands. The iterative development cycle required frequent revisits to prior modules, often needing to refine the execution of critical elements.

Inspired by strategy games like Command and Conquer and The Battle for Middle-earth, the project prioritised AI-driven behaviour over visual fidelity. The task-based AI hierarchy is a significant strength: agents independently explore and report environmental findings to their HQ, allocating tasks based on faction status. The result is a dynamic, adaptive ecosystem that I'm proud of.

Technical challenges were significant. Debugging AI performance bottlenecks led to the discovery that initialising agents on the CPU before transferring them to the GPU caused lag. Direct GPU initialisation resolved the issue. Another major obstacle involved a neural network state mismatch error; resolving it required tracing through the state encoding process to uncover a miscalculated constant. This reinforced the value of structured debugging and consistent state definitions.

Balancing cooperation (within factions) and competition (between factions) proved incredibly complex. I approached this by applying game design principles, treating each agent as a specialised player within a strategic team.

Key takeaways include enhanced debugging, profiling, and problem-solving skills, especially within reinforcement learning and multi-agent systems.

If extended, I would improve inter-agent communication and introduce diplomacy mechanics to deepen faction-level strategy, paving the way for future research in autonomous systems and strategic AI simulation.

References

GeeksforGeeks (2024) MultiAgent Reinforcement Learning in AI. https://www.geeksforgeeks.org/multi-agent-reinforcement-learning-in-ai/ (Accessed: April 9, 2025).

Canese, L., Cardarilli, G.C., Di Nunzio, L., Fazzolari, R., Giardino, D., Re, M. and Spanò, S. (2021) 'Multi-agent reinforcement learning: A review of challenges and applications', Applied Sciences, 11(11), p. 4948. Available at: https://doi.org/10.3390/app11114948 (Accessed: 1 April 2025).

Tran, K.-T., Dao, D., Nguyen, M.-D., Pham, Q.-V., O'Sullivan, B. and Nguyen, H.D. (2025) Multi-agent collaboration mechanisms: A survey of LLMs. arXiv preprint arXiv:2501.06322v1. Available at: https://arxiv.org/abs/2501.06322 (Accessed: 1 April 2025).

Zhao, Y., Ju, L. and Hernández-Orallo, J. (2024) 'Team formation through an assessor: choosing MARL agents in pursuit-evasion games'. doi:10.1007/s40747-023-01336-5. (Accessed: 1 April 2025).

Littman, M.L. (1994) 'Markov games as a framework for multi-agent reinforcement learning', Proceedings of the Eleventh International Conference on Machine Learning, pp. 157–163. Available at: https://courses.cs.duke.edu/spring07/cps296.3/littman94markov.pdf (Accessed: 1 April 2025).

Wong, A. et al. (2021) Deep Multiagent Reinforcement Learning: Challenges and Directions. https://arxiv.org/abs/2106.15691. (Accessed: 1 April 2025).

Lowe, R. *et al.* (2017) 'Multi-Agent Actor-Critic for mixed Cooperative-Competitive environments,' *arXiv (Cornell University)* [Preprint]. https://doi.org/10.48550/arxiv.1706.02275.

Iqbal, S. and Sha, F. (2018) 'Actor-Attention-Critic for Multi-Agent reinforcement learning,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1810.02912.

Rashid, T. et al. (2018) 'QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1803.11485.

Schulman, J. et al. (2017) Proximal Policy optimization Algorithms. https://arxiv.org/abs/1707.06347 (Accessed: 1 April 2025).

Mnih, V. et al. (2015b) 'Human-level control through deep reinforcement learning,' Nature, 518(7540), pp. 529–533. https://doi.org/10.1038/nature14236.

Mu, X. *et al.* (2025) 'Hierarchical task network-enhanced multi-agent reinforcement learning: toward efficient cooperative strategies,' *Neural Networks*, 107254. https://doi.org/10.1016/j.neunet.2025.107254.

Albrecht, S.V., Christianos, F. and Schäfer, L. (2024) *Multi-agent reinforcement learning: Foundations and modern approaches*. Cambridge, MA: MIT Press. Available at: https://www.marl-book.com/ (Accessed: 1 April 2025).

Zhu, T. *et al.* (2024) 'HyperComm: Hypergraph-based communication in multi-agent reinforcement learning,' *Neural Networks*, 178, p. 106432. https://doi.org/10.1016/j.neunet.2024.106432 (Accessed: May 7, 2025).

Sukhbaatar, S., Szlam, A. and Fergus, R. (2016) 'Learning Multiagent Communication with Backpropagation,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1605.07736 (Accessed: May 7, 2025).

Singh, A., Jain, T. and Sukhbaatar, S. (2018) 'Learning when to Communicate at Scale in Multiagent Cooperative and Competitive Tasks,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1812.09755 (Accessed: May 7, 2025).

Foerster, J.N. et al. (2016) 'Learning to Communicate with Deep Multi-Agent Reinforcement Learning,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1605.06676 (Accessed: May 7, 2025)..

Yu, P. *et al.* (2023) 'Enhancing Multi-Agent Coordination through Common Operating Picture Integration,' *arXiv (Cornell University)* [Preprint]. https://doi.org/10.48550/arxiv.2311.04740 (Accessed: May 7, 2025).

Liu, B. et al. (2021) 'Coach-Player Multi-Agent Reinforcement Learning for dynamic team composition,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2105.08692 (Accessed: May 7, 2025).

Shibata, K. et al. (2023) 'Learning Locally, Communicating Globally: Reinforcement learning of multi-robot task allocation for cooperative transport,' IFAC-PapersOnLine, 56(2), pp. 11436–11443. https://doi.org/10.1016/j.ifacol.2023.10.431 (Accessed: May 7, 2025).

Wang, J. et al. (2020) 'Shapley Q-Value: A local reward approach to solving global reward games,' Proceedings of the AAAI Conference on Artificial Intelligence, 34(05), pp. 7285–7292. https://doi.org/10.1609/aaai.v34i05.6220 (Accessed: May 7, 2025).

Leibo, J.Z. *et al.* (2017) 'Multi-agent reinforcement learning in sequential social dilemmas,' *arXiv (Cornell University)* [Preprint]. https://doi.org/10.48550/arxiv.1702.03037 (Accessed: May 7, 2025)..

Wang, T. et al. (2020) 'ROMA: Multi-Agent Reinforcement Learning with Emergent Roles,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2003.08039 (Accessed: May 7, 2025)..

Albrecht, S.V. and Stone, P. (2018) 'Autonomous agents modelling other agents: A comprehensive survey and open problems,' Artificial Intelligence, 258, pp. 66–95. https://doi.org/10.1016/j.artint.2018.01.002.

Baker, B. *et al.* (2019) 'Emergent tool use from Multi-Agent Autocurricula,' *arXiv (Cornell University)* [Preprint]. https://doi.org/10.48550/arxiv.1909.07528.

Abadi, M. et al. (2016) TensorFlow: Large-Scale machine learning on heterogeneous distributed systems. https://arxiv.org/abs/1603.04467.

Papoudakis, G. et al. (2020) 'Benchmarking Multi-Agent deep reinforcement learning algorithms in cooperative tasks,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2006.07869.

Appendices

Section A: Evaluation Test Results

This section presents detailed performance results and metrics for the seven test scenarios described in Chapter 5. Each subsection contains agent rewards, task distributions, HQ strategies, and key behavioural trends per configuration.

Appendix A.1 – Test 1: Static Baseline

Table A1.1 – Agent Rewards

Role	Average Reward
Gatherer	22.1
Hq	14.22
Peacekeeper	-43.73

Table A1.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	5848	90.4%
Success	620	9.6%

Table A1.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	7	0.1%
explore	151	2.3%
forage	0	0.0%
gather	2905	44.9%
mine	0	0.0%
move_to	3405	52.6%

Table A1.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	24572	79.3%
Gatherer	none	5630	18.2%
Gatherer	gather	778	2.5%
Gatherer	eliminate	0	0.0%

Gatherer	explore	0	0.0%
Peacekeeper	move_to	15733	53.0%
Peacekeeper	none	12138	40.9%
Peacekeeper	eliminate	1809	6.1%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A1.5 – Action Frequencies by Role

Role	Action	Count	Percentage
Gatherer	move_left	4385	14.2%
Gatherer	move_down	4378	14.1%
Gatherer	move_up	4186	13.5%
Gatherer	explore	4033	13.0%
Gatherer	forage_apple	3953	12.8%
Gatherer	move_right	3818	12.3%
Gatherer	heal_with_apple	3145	10.2%
Gatherer	mine_gold	3082	9.9%
Peacekeeper	move_left	4565	15.4%
Peacekeeper	move_right	4554	15.3%
Peacekeeper	move_up	4483	15.1%
Peacekeeper	move_down	4217	14.2%
Peacekeeper	explore	4179	14.1%
Peacekeeper	patrol	2823	9.5%
Peacekeeper	eliminate_threat	2699	9.1%
Peacekeeper	heal_with_apple	2160	7.3%

Table A1.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	3375	19.9%
ATTACK_THREATS	1384	8.2%
COLLECT_GOLD	3302	19.5%
COLLECT_FOOD	938	5.5%
RECRUIT_GATHERER	1278	7.5%
RECRUIT_PEACEKEEPER	4832	28.5%
NO_PRIORITY	1847	10.9%

Table A1.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	8.46
Faction1	Hq	18.54
Faction1	Peacekeeper	7.06
Faction2	Gatherer	35.74
Faction2	Hq	9.91
Faction2	Peacekeeper	-94.53

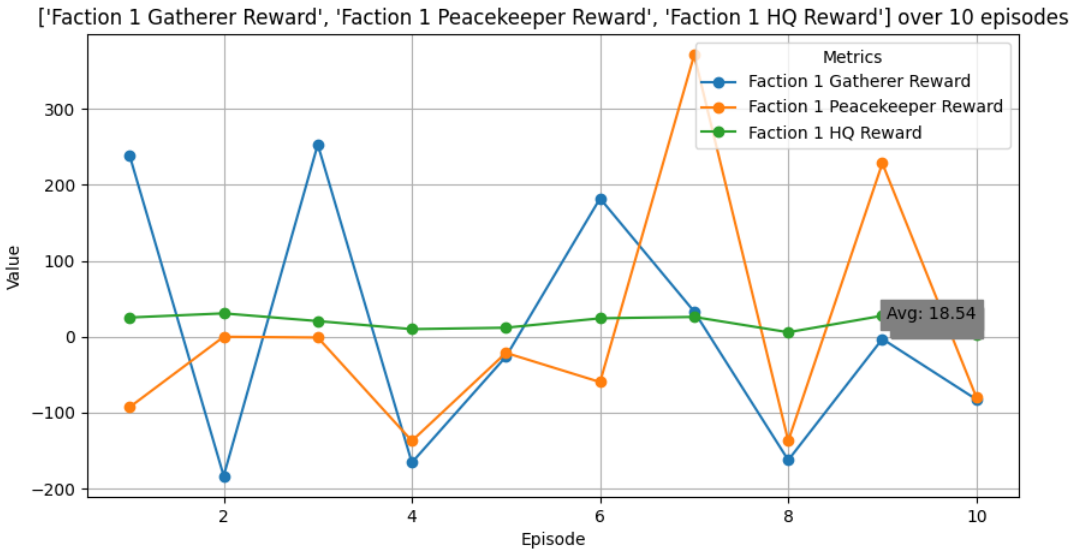
Table A1.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	346	4955	6.98%
Faction2	274	1513	18.11%

Table A1.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	3325	39.2%
Faction1	ATTACK_THREATS	1141	13.5%
Faction1	COLLECT_GOLD	100	1.2%
Faction1	COLLECT_FOOD	200	2.4%
Faction1	RECRUIT_GATHERER	50	0.6%
Faction1	RECRUIT_PEACEKEEPER	2986	35.2%
Faction1	NO_PRIORITY	676	8.0%
Faction2	DEFEND_HQ	50	0.6%
Faction2	ATTACK_THREATS	243	2.9%
Faction2	COLLECT_GOLD	3202	37.8%
Faction2	COLLECT_FOOD	738	8.7%
Faction2	RECRUIT_GATHERER	1228	14.5%
Faction2	RECRUIT_PEACEKEEPER	1846	21.8%
Faction2	NO_PRIORITY	1171	13.8%

(Figure A1.10 – Faction 1 Reward trend graph)



Appendix A.2 – Test 2: Increased faction competition

Table A2.1 – Agent Rewards

Role	Average Reward
Gatherer	18.97
Hq	13.1
Peacekeeper	7.58

Table A2.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	6576	90.0%
Success	732	10.0%

Table A2.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	11	0.2%
explore	165	2.3%
forage	0	0.0%
gather	3537	48.4%
mine	0	0.0%
move_to	3595	49.2%

Table A2.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	27947	91.0%
Gatherer	none	2185	7.1%
Gatherer	gather	577	1.9%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	none	14985	53.0%
Peacekeeper	move_to	12137	43.0%
Peacekeeper	eliminate	1133	4.0%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A2.5 – Action Frequencies by Role

Role	Action	Count	Percentage
Gatherer	forage_apple	4152	13.5%
Gatherer	explore	4009	13.1%
Gatherer	move_right	4000	13.0%
Gatherer	move_up	3857	12.6%
Gatherer	move_left	3783	12.3%
Gatherer	mine_gold	3682	12.0%
Gatherer	heal_with_apple	3629	11.8%
Gatherer	move_down	3597	11.7%
Peacekeeper	explore	4728	16.7%
Peacekeeper	move_left	4384	15.5%
Peacekeeper	move_up	4070	14.4%
Peacekeeper	move_right	3993	14.1%
Peacekeeper	move_down	3876	13.7%
Peacekeeper	eliminate_threat	2702	9.6%
Peacekeeper	patrol	2488	8.8%
Peacekeeper	heal_with_apple	2014	7.1%

Table A2.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	2521	14.7%
ATTACK_THREATS	1135	6.6%
COLLECT_GOLD	1295	7.5%
COLLECT_FOOD	1355	7.9%
RECRUIT_GATHERER	4588	26.7%
RECRUIT_PEACEKEEPER	4062	23.6%
NO_PRIORITY	2246	13.1%

Table A2.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	-13.41
Faction1	Hq	11.7
Faction1	Peacekeeper	-43.83
Faction2	Gatherer	-28.16
Faction2	Hq	15.13
Faction2	Peacekeeper	60.55
Faction3	Gatherer	98.47
Faction3	Hq	12.46
Faction3	Peacekeeper	6.03

Table A2.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)

Faction1	158	908	17.40%
Faction2	153	2859	5.35%
Faction3	421	3541	11.89%

Table A2.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	164	2.9%
Faction1	ATTACK_THREATS	88	1.5%
Faction1	COLLECT_GOLD	0	0.0%
Faction1	COLLECT_FOOD	533	9.3%
Faction1	RECRUIT_GATHERER	1983	34.6%
Faction1	RECRUIT_PEACEKEEPER	2603	45.4%
Faction1	NO_PRIORITY	363	6.3%
Faction2	DEFEND_HQ	1628	28.4%
Faction2	ATTACK_THREATS	492	8.6%
Faction2	COLLECT_GOLD	0	0.0%
Faction2	COLLECT_FOOD	200	3.5%
Faction2	RECRUIT_GATHERER	1843	32.1%
Faction2	RECRUIT_PEACEKEEPER	1459	25.4%
Faction2	NO_PRIORITY	112	2.0%
Faction3	DEFEND_HQ	729	12.7%
Faction3	ATTACK_THREATS	555	9.7%
Faction3	COLLECT_GOLD	1295	22.6%
Faction3	COLLECT_FOOD	622	10.8%
Faction3	RECRUIT_GATHERER	762	13.3%
Faction3	RECRUIT_PEACEKEEPER	0	0.0%
Faction3	NO_PRIORITY	1771	30.9%
Faction1	DEFEND_HQ	164	2.9%
Faction1	ATTACK_THREATS	88	1.5%
Faction1	COLLECT_GOLD	0	0.0%
Faction1	COLLECT_FOOD	533	9.3%
Faction1	RECRUIT_GATHERER	1983	34.6%
Faction1	RECRUIT_PEACEKEEPER	2603	45.4%
Faction1	NO_PRIORITY	363	6.3%

(Figure A2.1 – Reward trend graph)

Appendix A.3 – Test 3: Peacekeeper majority

Table A3.1 – Agent Rewards

Role	Average Reward
Gatherer	-60.42
Hq	9.23
Peacekeeper	-25.67

Table A3.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	9383	95.7%
Success	425	4.3%

Table A3.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	34	0.3%
explore	209	2.1%
forage	0	0.0%
gather	1134	11.6%
mine	0	0.0%
move_to	8431	86.0%

Table A3.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	26006	88.1%
Gatherer	none	2507	8.5%
Gatherer	gather	994	3.4%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	move_to	44863	64.2%
Peacekeeper	none	20089	28.7%
Peacekeeper	eliminate	4955	7.1%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A3.5 – Action Frequencies by Role

Role	Action	Count	Percentage

Gatherer	move_down	4058	13.8%
Gatherer	move_up	3867	13.1%
Gatherer	move_left	3811	12.9%
Gatherer	move_right	3779	12.8%
Gatherer	explore	3769	12.8%
Gatherer	heal_with_apple	3423	11.6%
Gatherer	mine_gold	3412	11.6%
Gatherer	forage_apple	3388	11.5%
Peacekeeper	move_right	12617	18.0%
Peacekeeper	heal_with_apple	10698	15.3%
Peacekeeper	move_down	10117	14.5%
Peacekeeper	explore	8273	11.8%
Peacekeeper	move_left	7728	11.1%
Peacekeeper	move_up	7719	11.0%
Peacekeeper	patrol	6879	9.8%
Peacekeeper	eliminate_threat	5876	8.4%

Table A3.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	9679	30.3%
ATTACK_THREATS	2077	6.5%
COLLECT_GOLD	3521	11.0%
COLLECT_FOOD	725	2.3%
RECRUIT_GATHERER	8192	25.6%
RECRUIT_PEACEKEEPER	250	0.8%
NO_PRIORITY	7500	23.5%

Table A3.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	-78.2
Faction1	Hq	-1.34
Faction1	Peacekeeper	-60.9
Faction2	Gatherer	-47.59
Faction2	Hq	12.15
Faction2	Peacekeeper	-59.25
Faction3	Gatherer	-55.46
Faction3	Hq	16.87
Faction3	Peacekeeper	43.13

Table A3.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	119	451	26.39%
Faction2	148	3081	4.80%

Faction3	158	6276	2.52%

Table A3.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	65	0.6%
Faction1	ATTACK_THREATS	0	0.0%
Faction1	COLLECT_GOLD	715	6.7%
Faction1	COLLECT_FOOD	191	1.8%
Faction1	RECRUIT_GATHERER	4477	42.0%
Faction1	RECRUIT_PEACEKEEPER	200	1.9%
Faction1	NO_PRIORITY	5000	47.0%
Faction2	DEFEND_HQ	3165	29.7%
Faction2	ATTACK_THREATS	1927	18.1%
Faction2	COLLECT_GOLD	265	2.5%
Faction2	COLLECT_FOOD	150	1.4%
Faction2	RECRUIT_GATHERER	2591	24.3%
Faction2	RECRUIT_PEACEKEEPER	50	0.5%
Faction2	NO_PRIORITY	2500	23.5%
Faction3	DEFEND_HQ	6449	60.6%
Faction3	ATTACK_THREATS	150	1.4%
Faction3	COLLECT_GOLD	2541	23.9%
Faction3	COLLECT_FOOD	384	3.6%
Faction3	RECRUIT_GATHERER	1124	10.6%
Faction3	RECRUIT_PEACEKEEPER	0	0.0%
Faction3	NO_PRIORITY	0	0.0%

(Figure A3.1 – Reward trend graph)

Appendix A.4 – Test 4: Gatherer majority

Table A4.1 – Agent Rewards

Role	Average Reward
Gatherer	-224.37
Hq	3.65
Peacekeeper	-80.75

Table A4.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	9567	92.9%
Success	732	7.1%

Table A4.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	8	0.1%
explore	507	4.9%
forage	0	0.0%
gather	1647	16.0%
mine	0	0.0%
move_to	8137	79.0%

Table A4.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	147047	80.1%
Gatherer	none	33656	18.3%
Gatherer	gather	2778	1.5%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	none	32582	50.5%
Peacekeeper	move_to	25123	38.9%
Peacekeeper	eliminate	6818	10.6%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A4.5 – Action Frequencies by Role

Role	Action	Count	Percentage

Gatherer	move_down	26542	14.5%
Gatherer	move_right	25930	14.1%
Gatherer	move_up	25872	14.1%
Gatherer	move_left	25630	14.0%
Gatherer	explore	23363	12.7%
Gatherer	heal_with_apple	19157	10.4%
Gatherer	mine_gold	18804	10.2%
Gatherer	forage_apple	18183	9.9%
Peacekeeper	move_right	9857	15.3%
Peacekeeper	move_left	9360	14.5%
Peacekeeper	explore	8903	13.8%
Peacekeeper	move_down	8891	13.8%
Peacekeeper	patrol	8570	13.3%
Peacekeeper	move_up	8516	13.2%
Peacekeeper	eliminate_threat	5993	9.3%
Peacekeeper	heal_with_apple	4433	6.9%

Table A4.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	6350	9.9%
ATTACK_THREATS	5850	9.1%
COLLECT_GOLD	50	0.1%
COLLECT_FOOD	14595	22.8%
RECRUIT_GATHERER	14083	22.0%
RECRUIT_PEACEKEEPER	14583	22.8%
NO_PRIORITY	8431	13.2%

Table A4.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	-227.29
Faction1	Hq	4.88
Faction1	Peacekeeper	-138.05
Faction2	Gatherer	-203.98
Faction2	Hq	-0.0
Faction2	Peacekeeper	-175.5
Faction3	Gatherer	-241.85
Faction3	Hq	6.08
Faction3	Peacekeeper	71.32

Table A4.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	251	4415	5.69%
Faction2	194	2382	8.14%

Faction3	287	3502	8.20%

Table A4.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	2050	9.6%
Faction1	ATTACK_THREATS	2450	11.5%
Faction1	COLLECT_GOLD	50	0.2%
Faction1	COLLECT_FOOD	5383	25.3%
Faction1	RECRUIT_GATHERER	2700	12.7%
Faction1	RECRUIT_PEACEKEEPER	5250	24.6%
Faction1	NO_PRIORITY	3431	16.1%
Faction2	DEFEND_HQ	1800	8.4%
Faction2	ATTACK_THREATS	150	0.7%
Faction2	COLLECT_GOLD	0	0.0%
Faction2	RECRUIT_GATHERER	11000	51.6%
Faction2	RECRUIT_PEACEKEEPER	83	0.4%
Faction2	NO_PRIORITY	0	0.0%
Faction3	DEFEND_HQ	2500	11.7%
Faction3	ATTACK_THREATS	3250	15.2%
Faction3	COLLECT_GOLD	0	0.0%
Faction3	COLLECT_FOOD	931	4.4%
Faction3	RECRUIT_GATHERER	383	1.8%
Faction3	RECRUIT_PEACEKEEPER	9250	43.4%
Faction3	NO_PRIORITY	5000	23.5%

(Figure A4.1 – Reward trend graph)

Appendix A.5 – Test 5: Resource-heavy test

Resource-heavy configuration: This test increased overall resource availability by raising the apple tree density to 0.15 and the gold zone probability to 0.1. The base quantity of apples and gold was set to 10. Agent purchase cost was doubled to 10 units for balancing purposes.

Table A5.1 – Agent Rewards

Role	Average Reward
Gatherer	97.71
Hq	16.99
Peacekeeper	-68.78

Table A5.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	66388	92.4%
Success	5423	7.6%

Table A5.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	164	0.2%
explore	1049	1.5%
forage	0	0.0%
gather	42644	59.4%
mine	0	0.0%
move_to	27954	38.9%

Table A5.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	293912	80.5%
Gatherer	none	55820	15.3%
Gatherer	gather	15586	4.3%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	move_to	153047	48.5%
Peacekeeper	none	144670	45.8%
Peacekeeper	eliminate	18044	5.7%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A5.5 – Action Frequencies by Role

Role	Action	Count	Percentage
Gatherer	move_left	51429	14.1%

Gatherer	explore	49915	13.7%
Gatherer	move_right	48332	13.2%
Gatherer	move_down	48286	13.2%
Gatherer	move_up	48184	13.2%
Gatherer	forage_apple	40656	11.1%
Gatherer	mine_gold	39473	10.8%
Gatherer	heal_with_apple	39043	10.7%
Peacekeeper	move_right	49026	15.5%
Peacekeeper	move_down	47530	15.1%
Peacekeeper	move_left	44900	14.2%
Peacekeeper	move_up	44665	14.1%
Peacekeeper	explore	43535	13.8%
Peacekeeper	patrol	33526	10.6%
Peacekeeper	eliminate_threat	31461	10.0%
Peacekeeper	heal_with_apple	21118	6.7%

Table A5.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	35778	18.1%
ATTACK_THREATS	25724	13.0%
COLLECT_GOLD	13335	6.7%
COLLECT_FOOD	57405	29.1%
RECRUIT_GATHERER	20134	10.2%
RECRUIT_PEACEKEEPER	26234	13.3%
NO_PRIORITY	18991	9.6%

Table A5.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	33.95
Faction1	Hq	12.22
Faction1	Peacekeeper	-118.68
Faction2	Gatherer	213.17
Faction2	Hq	13.57
Faction2	Peacekeeper	-127.3
Faction3	Gatherer	46.0
Faction3	Hq	25.2
Faction3	Peacekeeper	39.64

Table A5.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	1394	16050	8.69%
Faction2	2346	22878	10.25%
Faction3	1683	32883	5.12%

Table A5.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	8825	13.4%
Faction1	ATTACK_THREATS	3034	4.6%
Faction1	COLLECT_GOLD	3612	5.5%
Faction1	COLLECT_FOOD	35097	53.3%
Faction1	RECRUIT_GATHERER	4258	6.5%
Faction1	RECRUIT_PEACEKEEPER	8908	13.5%
Faction1	NO_PRIORITY	2133	3.2%
Faction2	DEFEND_HQ	3306	5.0%
Faction2	ATTACK_THREATS	14660	22.3%
Faction2	COLLECT_GOLD	6578	10.0%
Faction2	COLLECT_FOOD	17565	26.7%
Faction2	RECRUIT_GATHERER	285	0.4%
Faction2	RECRUIT_PEACEKEEPER	13463	20.4%
Faction2	NO_PRIORITY	10010	15.2%
Faction3	DEFEND_HQ	23647	35.9%
Faction3	ATTACK_THREATS	8030	12.2%
Faction3	COLLECT_GOLD	3145	4.8%
Faction3	COLLECT_FOOD	4743	7.2%
Faction3	RECRUIT_GATHERER	15591	23.7%
Faction3	RECRUIT_PEACEKEEPER	3863	5.9%
Faction3	NO_PRIORITY	6848	10.4%

(Figure A5.1 – Reward trend graph)

Appendix A.6 – Test 6: Default settings, longer-run

Table A6.1 – Agent Rewards

Role	Average Reward
Gatherer	-197.29
Hq	5.29
Peacekeeper	4.63

Table A6.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	32275	95.3%
Success	1595	4.7%

Table A6.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	32	0.1%
explore	946	2.8%
forage	0	0.0%
gather	3798	11.2%
mine	0	0.0%
move_to	29094	85.9%

Table A6.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	246980	90.1%
Gatherer	none	22110	8.1%
Gatherer	gather	5106	1.9%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	none	146379	61.8%
Peacekeeper	move_to	78970	33.4%
Peacekeeper	eliminate	11413	4.8%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A6.5 – Action Frequencies by Role

Role	Action	Count	Percentage
Gatherer	move_left	37428	13.7%
Gatherer	move_right	36797	13.4%
Gatherer	explore	35826	13.1%
Gatherer	move_up	35459	12.9%
Gatherer	move_down	35083	12.8%

Gatherer	forage_apple	31742	11.6%
Gatherer	mine_gold	31713	11.6%
Gatherer	heal_with_apple	30148	11.0%
Peacekeeper	move_up	45795	19.3%
Peacekeeper	move_left	38539	16.3%
Peacekeeper	move_right	35106	14.8%
Peacekeeper	move_down	34218	14.5%
Peacekeeper	explore	33767	14.3%
Peacekeeper	eliminate_threat	17606	7.4%
Peacekeeper	patrol	16437	6.9%
Peacekeeper	heal_with_apple	15294	6.5%

Table A6.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	30168	21.2%
ATTACK_THREATS	8932	6.3%
COLLECT_GOLD	11306	7.9%
COLLECT_FOOD	3580	2.5%
RECRUIT_GATHERER	43792	30.8%
RECRUIT_PEACEKEEPER	29909	21.0%
NO_PRIORITY	14561	10.2%

Table A6.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	-196.39
Faction1	Hq	8.03
Faction1	Peacekeeper	-0.76
Faction2	Gatherer	-198.2
Faction2	Hq	2.55
Faction2	Peacekeeper	10.01

Table A6.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	821	15231	5.39%
Faction2	774	18639	4.15%

Table A6.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	11436	16.1%
Faction1	ATTACK_THREATS	8447	11.9%
Faction1	COLLECT_GOLD	1455	2.0%
Faction1	COLLECT_FOOD	372	0.5%

Faction1	RECRUIT_GATHERER	27924	39.3%
Faction1	RECRUIT_PEACEKEEPER	16116	22.7%
Faction1	NO_PRIORITY	5374	7.6%
Faction2	DEFEND_HQ	18732	26.3%
Faction2	ATTACK_THREATS	485	0.7%
Faction2	COLLECT_GOLD	9851	13.9%
Faction2	COLLECT_FOOD	3208	4.5%
Faction2	RECRUIT_GATHERER	15868	22.3%
Faction2	RECRUIT_PEACEKEEPER	13793	19.4%
Faction2	NO_PRIORITY	9187	12.9%

(Figure A6.1 – Reward trend graph)

Appendix A.7 – Test 7: Longer run, increased factions, default settings

Table A7.1 – Agent Rewards

Role	Average Reward
Gatherer	-288.13
Hq	-7.84
Peacekeeper	-52.39

Table A7.2 – Task Result Distribution

Result	Count	Percentage
Failure	0	0.0%
Ongoing	46607	94.7%
Success	2613	5.3%

Table A7.3 – Task Type Distribution

Task Type	Count	Percentage
defend	0	0.0%
eliminate	135	0.3%
explore	1595	3.2%
forage	0	0.0%
gather	5554	11.3%
mine	0	0.0%
move_to	41936	85.2%

Table A7.4 – Task Types by Role

Role	Task Type	Count	Percentage
Gatherer	move_to	458267	79.1%
Gatherer	none	102445	17.7%
Gatherer	gather	18469	3.2%
Gatherer	eliminate	0	0.0%
Gatherer	explore	0	0.0%
Peacekeeper	move_to	254645	57.0%
Peacekeeper	none	147040	32.9%
Peacekeeper	eliminate	45165	10.1%
Peacekeeper	gather	0	0.0%
Peacekeeper	explore	0	0.0%

Table A7.5 – Action Frequencies by Role

Role	Action	Count	Percentage
Gatherer	move_right	84056	14.5%
Gatherer	move_up	79897	13.8%
Gatherer	move_down	78798	13.6%
Gatherer	move_left	78273	13.5%
Gatherer	explore	76283	13.2%

Gatherer	forage_apple	61668	10.6%
Gatherer	mine_gold	60741	10.5%
Gatherer	heal_with_apple	59465	10.3%
Peacekeeper	move_up	95351	21.3%
Peacekeeper	explore	64419	14.4%
Peacekeeper	move_left	62861	14.1%
Peacekeeper	move_right	55816	12.5%
Peacekeeper	move_down	51464	11.5%
Peacekeeper	patrol	41952	9.4%
Peacekeeper	heal_with_apple	37894	8.5%
Peacekeeper	eliminate_threat	37093	8.3%

Table A7.6 – HQ Strategy Distribution

Strategy	Count	Percentage
DEFEND_HQ	47804	14.4%
ATTACK_THREATS	47874	14.4%
COLLECT_GOLD	46039	13.9%
COLLECT_FOOD	24758	7.5%
RECRUIT_GATHERER	74424	22.4%
RECRUIT_PEACEKEEPER	19264	5.8%
NO_PRIORITY	72090	21.7%

Table A7.7 – Reward by Faction and Role

Faction	Role	Average Reward
Faction1	Gatherer	-291.24
Faction1	Hq	-6.46
Faction1	Peacekeeper	-245.34
Faction2	Gatherer	-227.68
Faction2	Hq	-6.83
Faction2	Peacekeeper	-102.12

Table A7.8 – Task Success Rate by Faction

Faction	Successful Tasks	Total Tasks	Success Rate (%)
Faction1	747	7016	10.65%
Faction2	750	12168	6.16%
Faction3	1116	30036	3.72%

Table A7.9 – HQ Strategy Usage by Faction

Faction	Strategy	Count	Percentage
Faction1	DEFEND_HQ	13307	12.0%
Faction1	ATTACK_THREATS	10819	9.8%
Faction1	COLLECT_GOLD	25325	22.9%

Faction1	COLLECT_FOOD	14924	13.5%
Faction1	RECRUIT_GATHERER	28643	25.9%
Faction1	RECRUIT_PEACEKEEPER	10073	9.1%
Faction1	NO_PRIORITY	7660	6.9%
Faction2	DEFEND_HQ	20272	18.3%
Faction2	ATTACK_THREATS	13616	12.3%
Faction2	COLLECT_GOLD	20314	18.3%
Faction2	COLLECT_FOOD	8577	7.7%
Faction2	RECRUIT_GATHERER	6976	6.3%
Faction2	RECRUIT_PEACEKEEPER	8346	7.5%
Faction2	NO_PRIORITY	32650	29.5%
Faction3	DEFEND_HQ	14225	12.8%
Faction3	ATTACK_THREATS	23439	21.2%
Faction3	COLLECT_GOLD	400	0.4%
Faction3	COLLECT_FOOD	1257	1.1%
Faction3	RECRUIT_GATHERER	38805	35.0%
Faction3	RECRUIT_PEACEKEEPER	845	0.8%
Faction3	NO_PRIORITY	31780	28.7%

(Figure A7.1 – Reward trend graph)

Figures

Tables