applied sciences

MDPI

Article

Graph-Based Task Allocation for Multi-Agent Fleet Management: A Genetic Algorithm Approach with LLM Integration

Beril Yalcinkaya [1,2], Micael S. Couceiro [2,*], Salviano Soares [1] and António Valente [1,3]

[1] Engineering Department, School of Sciences and Technology, University of Trás-os-Montes and Alto Douro (UTAD), 5000-801 Vila Real, Portugal; beril@ingeniarius.pt (B.Y.); salblues@utad.pt (S.S.); avalente@utad.pt (A.V.)
[2] CORE R&D Department, Ingeniarius Lda, 4445-147 Alfena, Portugal
[3] Technology and Science, INESC TEC, 4200-465 Porto, Portugal
* Correspondence: micael@ingeniarius.pt

Abstract

Efficient task allocation and coordination are critical for heterogeneous multi-agent systems operating in dynamic field environments. This paper presents a closed-loop framework that integrates Large Language Models (LLMs) with graph-based optimisation to enable end-to-end task decomposition, allocation, and adaptive execution. High-level task scripts are initially parsed by an LLM into structured execution flows, which are transformed into Directed Acyclic Graphs (DAGs) capturing action-level dependencies. A Genetic Algorithm (GA) then optimises agent-to-task assignments by minimising makespan under capability and battery constraints. To ensure robustness, the framework incorporates an LLM-driven recovery module that enables localised replanning under execution failures without interrupting unaffected agents. System-level experiments in a high-fidelity agro-forestry simulation demonstrate a 37% increase ($p < 0.001$) in harvesting productivity and a 19% reduction in human idle time compared to manual baselines. Under mid-execution failures, the system maintains significantly higher performance, with replanning latencies averaging 24 s. The framework scales to large fleets (up to 1000 agents) and effectively enhances human–robot collaboration through structured, dependency-aware coordination.

Keywords: task allocation; multi-agent systems; genetic algorithm; directed acyclic graph; large language models; human–robot collaboration

1. Introduction

Task allocation is a fundamental challenge in multi-agent systems, particularly in environments where autonomous agents and human collaborators must coordinate to achieve complex, time-critical goals [1]. These hybrid systems are becoming increasingly prevalent in logistics [2], search and rescue [3], and precision agriculture [4], where agents with complementary capabilities must collaborate to complete tasks efficiently. In field robotics applications, such as agroforestry or construction, effective coordination requires not only initial allocation but also a continuous reasoning loop capable of handling resource constraints and operational uncertainties.

In many real-world environments, tasks are not atomic; they consist of multiple subtasks involving action-level dependencies that may need to be executed sequentially or distributed across a heterogeneous fleet [5]. For instance, in collaborative berry harvesting, tasks like picking, navigating, and transporting must be synchronised between human pickers and assisting drones. Each agent brings distinct advantages: humans excel at

Check for updates

Academic Editors: Kensuke Harada and Luigi Bibbò

Received: 23 March 2026
Revised: 9 April 2026
Accepted: 13 April 2026
Published: 15 April 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

delicate manipulation and complex terrain navigation, while drones provide high-speed logistics and data coverage. However, coordination becomes significantly more complex due to varying performance levels, battery limitations, and the intrinsic transport burden on humans.

Traditional task allocation frameworks often treat tasks as independent units, failing to capture the intricate temporal and logical dependencies between subtasks [6]. This often leads to suboptimal synchronisation, where agents remain idle while waiting for prerequisites, or mission failure when mid-execution uncertainties arise. To address these limitations, a more sophisticated approach is required, one that combines high-level semantic reasoning with robust optimisation to manage the entire decision–execution loop.

This work introduces a closed-loop, DAG-aware multi-agent coordination framework that tightly integrates LLM-based task understanding with optimisation-driven allocation and adaptive execution recovery. We leverage Directed Acyclic Graphs (DAGs) to explicitly capture sequential and parallel dependencies between subtask components, facilitating a clear representation of complex action-level workflows [7]. To extract this underlying structure from high-level task descriptions, we use LLMs to automatically infer execution steps and interdependencies from human-written task code. This semantic interpretation provides a robust foundation for modelling tasks as executable graphs.

For task allocation, we employ a Genetic Algorithm (GA) designed for its scalability and flexibility in solving multi-objective combinatorial problems [8]. Unlike rigid deterministic methods, the GA evaluates potential mappings based on the critical path of the DAG, incorporating penalties for battery violations and spatial inefficiencies. Beyond initial allocation, the framework integrates a monitoring and recovery layer. By maintaining a live execution context, the system can detect mid-execution failures and leverage the LLM's reasoning to synthesise recovery routines, ensuring mission continuity without a full system restart.

The performance of the integrated architecture is validated in a high-fidelity simulated agroforestry environment. The results demonstrate that our coordination logic significantly enhances human–robot synergy, yielding measurable gains in productivity and resilience.

While recent advancements such as TaskLAMA [9] and DAG-Plan [10] have demonstrated the potential of LLMs in extracting procedural knowledge from natural language, these frameworks primarily operate as open-loop planners. They focus on the semantic translation of instructions into static graphs but lack an integrated mechanism to map these plans onto a heterogeneous fleet with physical constraints (e.g., battery, spatial proximity) or to recover from mid-execution failures. The fundamental novelty of this work lies in the closed-loop synergy between LLM-driven semantic decomposition and Genetic Algorithm-based optimisation. Unlike previous works, our framework treats the generated DAG as a live, repairable execution context. By enabling the LLM to function as an adaptive supervisor that can locally replan a failing mission segment without interrupting unaffected agents, we bridge the gap between high-level reasoning and robust, multi-agent field execution.

This paper makes the following contributions:

1. An LLM-based method for extracting structured execution flows from human-written task code;
2. A DAG construction pipeline that captures action-level temporal and synchronisation dependencies;
3. A graph-aware Genetic Algorithm for scalable task allocation under capability, battery, and spatial constraints;
4. A closed-loop execution framework with LLM-driven, DAG-consistent recovery for maintaining mission continuity under failures.

2. State of the Art

In recent years, advancements in robotics technology have led to an increased deployment of robots across a variety of fields, including agriculture, search and rescue, logistics, and warehouse management. The tasks in these domains are often heterogeneous, meaning they vary in type and complexity, each with distinct requirements. Whether the robots involved are homogeneous or heterogeneous, it is essential to allocate tasks to the most suitable robots to ensure task completion in the most efficient manner possible, minimising both time and resource utilisation. This challenge is formalised in the Multi-Robot Task Allocation (MRTA) problem, which involves assigning tasks to multiple robots in a way that optimises performance and achieves a specified objective. The significance of this problem lies in its potential to improve operational efficiency, reduce task completion time, and enhance the overall performance of the system.

To better understand and categorise the complexity of the MRTA problem, Gerkey and Matarić introduced a widely adopted taxonomy that classifies MRTA scenarios based on three key dimensions [11]: robot capabilities, task requirements, and the nature of task assignments. First, with respect to robot capabilities, the taxonomy distinguishes between Single-Task (ST) robots, which can perform only one task at a time, and Multi-Task (MT) robots, which can handle several tasks simultaneously. Second, in terms of task requirements, it differentiates between Single-Robot (SR) tasks that require only one agent and Multi-Robot (MR) tasks that necessitate collaboration among multiple agents. Third, the assignment type is classified as either Instantaneous Assignment (IA), where tasks are allocated without consideration for future tasks, or Time-Extended Assignment (TA), where future tasks and constraints are considered to optimise the global allocation strategy.

Beyond taxonomy, MRTA solutions can also be distinguished by the organisational paradigm of the agent team, particularly centralised vs. decentralised approaches. In centralised systems, a single decision-making agent or controller has access to global knowledge and is responsible for allocating tasks to all other agents. This paradigm benefits from simplified coordination, consistent global optimisation, and reduced duplication of effort [12]. However, it suffers from well-known limitations, such as scalability bottlenecks, vulnerability to single-point failures, and lack of responsiveness in dynamic or large-scale environments. Centralised approaches are therefore best suited for smaller-scale, static environments where global state information is readily available. Conversely, decentralised task allocation distributes decision-making across multiple agents. Each agent may act autonomously or collaborate with others to reach a shared goal, often using mechanisms such as auctions, consensus algorithms, or evolutionary methods [13,14]. The main advantage of decentralised approaches is their robustness and scalability: the failure of one agent does not compromise the entire system, and new agents can be added with minimal reconfiguration. However, these systems face the challenge of coordination under uncertainty, where local decisions may not lead to globally optimal outcomes. As such, decentralised approaches often trade off optimality for increased flexibility, responsiveness, and fault tolerance, making them suitable for dynamic, partially observable, or large-scale settings.

The most extensively studied and applied strategies in MRTA are market-based approaches and optimisation-based approaches. Inspired by economic principles, the market-based approach models the task assignment problem as a resource allocation problem, where agents "bid" for tasks based on utility, cost, or suitability metrics. In a typical implementation, agents act as self-interested entities that compute their own valuations of tasks and participate in auctions or negotiations, with a central or distributed auctioneer managing the task distribution. Variants such as the Contract Net Protocol and Consensus-Based Bundle Algorithm (CBBA) [14] have been widely adopted in multi-robot systems for their ability to balance efficiency and autonomy. TraderBots [15] introduce a

framework where robots act as autonomous trading agents in a marketplace, negotiating for tasks based on utility. Similarly, M+ [16] and S + T [17] extend the market-based approach to enable coalition formation, allowing multiple agents to jointly perform tasks that require collaboration.

Market-based approaches offer notable advantages in multi-robot task allocation, including efficiency through local decision-making, robustness due to the absence of a single point of failure, and scalability as the system grows. They also handle dynamic environments well by supporting online task integration and flexible task reassignment. However, they come with limitations, such as the complexity of designing appropriate cost and utility functions, negotiation protocols, and potential communication overhead. In some scenarios, especially large-scale or well-structured problems, optimisation-based approaches can outperform market-based methods in solution quality and implementation simplicity [8]. Building upon this comparison, optimisation-based approaches have emerged as a strong alternative to market-based strategies in scenarios where solution quality and scalability are of paramount importance.

Optimisation-based methods for MRTA can be broadly categorised into deterministic and heuristic approaches. Deterministic methods, such as the Hungarian algorithm [18], branch-and-bound [19], and Mixed-Integer Linear Programming (MILP) [20], aim for exact solutions but struggle to scale due to high computational complexity. In contrast, metaheuristics like GAs [21], Particle Swarm Optimisation (PSO) [22], and Ant Colony Optimisation (ACO) [23] offer scalable, real-time solutions by trading off optimality for efficiency. Among these, GAs are especially robust and adaptable, yet most assume flat task structures and rely on handcrafted fitness functions, limiting their effectiveness in scenarios involving complex action dependencies or semantic constraints. This gap highlights the need for GA-based methods that can integrate structured task models, such as dependency graphs, with semantic task understanding.

In the context of task allocation, understanding the structure of the tasks themselves is crucial. Tasks can be classified based on their complexity, as proposed in [8]. Elemental (atomic) tasks are simple and cannot be divided into subtasks, requiring a single robot to execute them. Simple tasks, while breakable into subtasks, require all subtasks to be executed by the same robot. Compound tasks, on the other hand, involve decomposition into subtasks assigned to different robots or groups. Finally, complex tasks have multiple decompositions, with at least one feasible allocation involving multiple robots.

Simple tasks are rarely encountered in real-world multi-robot task allocation scenarios. Typically, tasks are structured with a global objective that is divided into subtasks. These decomposed subtasks often introduce a variety of dependencies that must be managed in task allocation and scheduling. For instance, subtasks may have internal dependencies, such as execution order, where one subtask cannot begin until another is completed. Additionally, conditional dependencies can arise, where the success or failure of one subtask determines whether subsequent subtasks proceed, are delayed, or altered. Furthermore, while parallel execution of subtasks can enhance efficiency, it introduces added complexity due to temporal dependencies and the need for synchronisation between tasks. In fact, Korsah, Dias, and Stentz extended Gerkey and Matarić's taxonomy to account for these types of constraints [24]. They identified four key dependency types:

- No Dependencies (ND): The cost of a given robot–task pair is independent of any other tasks or robots.
- In-Schedule Dependencies (ID): The cost of a robot–task pair depends on other tasks assigned to the same robot, creating intra-schedule dependencies.
- Cross-Schedule Dependencies (XD): The cost of a robot–task pair depends on the tasks assigned to other robots, resulting in inter-schedule dependencies.

- Complex Dependencies (CD): The cost of a robot–task pair depends on other robots' schedules, where task decomposition and allocation complexities create a combination of dependencies across robots.

These decompositions give rise to various types of dependencies: temporal (e.g., action A must precede action B), resource-based (e.g., limited battery or tools), and logical (e.g., conditional flows based on success or failure). To manage the complexity these dependencies introduce in multi-robot task allocation, DAGs are an effective tool for modelling action-level dependencies within higher-level tasks. In a DAG, actions (interpreted as low-level robotic behaviours) are represented as nodes, while edges define execution constraints such as ordering, conditions, or synchronisation. This structure facilitates clear representation of sequencing, concurrency, and control flow. Traditional methods, such as crowd-sourced approaches [25], often relied heavily on external data and were limited in scalability due to data availability and domain specificity.

While DAGs provide a powerful abstraction for capturing action dependencies, constructing them manually or from traditional pipelines can be labour-intensive and domain-specific. This is where LLMs have emerged as a transformative tool. Unlike traditional approaches, LLMs possess extensive commonsense and procedural knowledge acquired from large-scale pretraining on diverse real-world data. TaskLAMA demonstrated that LLMs can generate task graphs with accurate temporal dependencies by extracting structured flows from natural language instructions [9]. Similarly, DAG-Plan showed that LLMs can be used to efficiently generate collaborative task plans for mobile dual-arm robots by decomposing complex instructions into DAGs with clear temporal relationships [10]. While effective, DAG-Plan relies on reinforcement learning to execute the underlying robotic skills, which can become a bottleneck in tasks involving high complexity or variability.

Recent research has further explored the integration of LLMs with advanced graph-based reasoning techniques. Lin et al. [26] proposed Plan Like a Graph (PLaG), which translates natural language task sequences into graph structures with temporal constraints, enabling optimal scheduling. Likewise, Cao et al. [27] used LLMs to generate behaviour trees—another graph-based structure—for hierarchical robot control. Additionally, LiP-LLM [28] introduced a framework that integrates Linear Programming and Dependency Graphs with LLM outputs to solve multi-robot task planning problems. This approach first uses an LLM to extract action dependencies and preferences from natural language instructions, forming a dependency graph. Then, linear programming is applied to optimise the task allocation based on that graph structure. This combination demonstrates a promising hybrid model, where LLMs handle semantic understanding, while classical optimisation ensures rigorous task feasibility and efficiency.

Learning-based approaches, including deep learning and reinforcement learning (RL), have been explored in the context of MRTA for handling dynamic environments and large action spaces [29,30]. In addition, data-driven methods have been successfully applied in related domains such as fault diagnosis and system monitoring, where structured representations and graph-based learning enable effective reasoning over complex system states [31]. However, these methods often face limitations such as high data requirements, poor generalisation to novel scenarios, and difficulty incorporating task dependencies or domain-specific constraints, making them less suited for structured, dependency-aware MRTA problems.

Despite the broad application of market-based and optimisation-based methods in multi-robot task allocation (MRTA), each approach presents notable limitations. Market-based strategies, while decentralised and flexible, often rely on handcrafted utility functions and fail to capture complex action dependencies or inter-agent constraints. Deterministic optimisation methods, such as MILP, offer exact solutions but suffer from scalability issues

in large or dynamic environments. Metaheuristics, like GAs, provide scalability and adaptability, yet typically assume fixed task structures and overlook the semantic and temporal intricacies present in real-world robotic tasks. Concurrently, deep learning and reinforcement learning approaches demand extensive training data, generalise poorly to novel configurations, and often lack transparency in representing explicit action dependencies.

Recent efforts have begun to explore the integration of LLMs for task decomposition and DAG-based execution modelling. These approaches have demonstrated promising capabilities in extracting semantic structure from unstructured task descriptions or code. However, these approaches typically operate in isolation from constraint-aware MRTA formulations and are seldom integrated with optimisation strategies that can allocate tasks across heterogeneous agents while accounting for execution dependencies, resource limitations, and temporal constraints. This highlights a critical gap: the lack of an end-to-end framework that combines high-level semantic understanding with scalable, dependency-aware, and executable multi-agent planning. Furthermore, existing literature often treats task allocation as an initial planning phase, assuming perfect execution of assigned schedules. However, real-world field robotics are prone to dynamic failures and operational uncertainties. Most optimisation-based and LLM-integrated planners lack a continuous feedback loop that can detect mid-execution faults and adaptively re-allocate or repair task plans without human intervention. This highlights a critical need for resilient MRTA frameworks that not only handle semantic decomposition and initial allocation but also ensure mission continuity through autonomous recovery mechanisms. Individual components of task parsing or optimisation have been extensively studied. However, the seamless integration of LLM-driven task interpretation, DAG-based modelling, and GA-based optimisation into a cohesive, failure-resilient pipeline remains largely unexplored. Our work addresses this gap by providing a semantically grounded task allocation framework for heterogeneous fleets that ensures mission continuity even under mid-execution uncertainties.

As shown in Table 1, existing works either focus on LLM-based task decomposition or optimisation-based allocation, but do not integrate both within a unified execution framework. Moreover, failure recovery and closed-loop execution are largely absent in prior approaches.

Table 1. Comparison of related LLM-based task planning and allocation frameworks.

Feature	TaskLAMA	DAG-Plan	PLaG	LiP-LLM	Proposed Framework
Input	NL	NL	NL	NL	Python Scripts
Dependency	Static Graph	DAG	Temporal Graph	Dep. Graph	Dynamic DAG
Allocation	×	Limited	×	LP	GA
Heterogeneity	Limited	Single	Limited	Multi	Human–Robot
Constraints	×	×	×	Partial	Battery + Distance
Recovery	×	×	×	×	LLM Replan
Closed-loop	×	×	×	×	✓
Validation	Dataset	Sim	Dataset	Sim	HRC (FEROX)

×: Feature not supported or absent; ✓: Feature fully supported or present; GA: Genetic Algorithm; LP: Linear Programming; NL: Natural Language; Sim: Simulation.

In contrast, our framework combines semantic task parsing, dependency-aware DAG modelling, and optimisation-based allocation within a single pipeline, and introduces an LLM-driven recovery mechanism for adaptive replanning in human–robot collaborative scenarios.

3. Background: FORMIGA Framework and FEROX Use Case

This section provides essential context for our proposed task allocation approach by presenting the underlying system architecture and operational environment. We begin with an overview of the FORMIGA framework, focusing on its fleet manager subsystem, which facilitates coordinated task execution among multiple robots. This is followed by a description of the FEROX use case, a deployment scenario that illustrates how FORMIGA enables multi-agent collaboration. Together, these components establish the foundation for the methods and experiments presented in the subsequent sections.

3.1. FORMIGA Fleet Manager

FORMIGA (Framework for Optimised Robotic Management, Integration, Guidance, and Automation) is a centralised, Python-based (version 3.8.10), ROS-integrated framework designed to orchestrate the operations of heterogeneous multi-agent systems. Its core mission is to provide a flexible and modular solution for managing robotic and human agents in dynamic, real-world environments. In this work, FORMIGA serves as the infrastructure for integrating and evaluating our proposed task allocation methodology.

Building upon prior developments in the ROOSTER (https://github.com/ROOSTER-fleet-management, accessed on 12 April 2026) framework, FORMIGA extends the capabilities for defining, assigning, and executing collaborative tasks among fleets of autonomous agents. It supports dynamic task creation, context-aware agent selection, real-time resource monitoring, and seamless integration of external decision-making modules, including those powered by LLMs. These features make FORMIGA a suitable testbed for experimenting with scalable, intelligent task allocation strategies.

A key strength of FORMIGA lies in its modular architecture, composed of subsystems for task management, agent coordination, and fleet status monitoring. Tasks are expressed as Python functions, either written manually or automatically generated via LLMs. These functions are distributed to agents through the central fleet manager, which assigns them based on agent availability, proximity, and task requirements. This structure supports both single-agent and collaborative multi-agent tasks, enabling dynamic and flexible coordination.

The integration of LLMs into FORMIGA allows natural language task descriptions to be translated into executable Python code. This reduces the need for manual programming and enables rapid adaptation to new mission goals. Such capabilities are particularly valuable in field applications where adaptability, autonomy, and scalability are essential.

While previous versions of FORMIGA primarily assigned tasks based on agent availability and spatial proximity—assuming sufficient battery and ignoring task complexity—our work enhances this process by integrating a robust decision-making module. This module introduces LLM-driven task parsing, DAG-based dependency modelling, and a genetic algorithm for optimised, constraint-aware task allocation.

In this enhanced architecture, FORMIGA provides the runtime infrastructure, agent state monitoring, and task initiation pipeline. The LLM module parses human-written Python task code into structured execution flows; the DAG module transforms these flows into dependency graphs; and the GA module receives the DAG and current agent states (e.g., availability, battery, task load) as inputs to compute optimal task-to-agent assignments. We adopt the taxonomy introduced by Gerkey and Matarić [11] to characterise the capabilities of our system. Specifically, our implementation falls under the ST-MR-TA class: robots are Single-Task (ST), meaning that they execute one task at a time; tasks can require Multi-Robot (MR) collaboration through explicitly defined dependencies; and the allocation process is Time-Extended (TA), considering future constraints via DAG modelling and GA-based optimisation. As a result, FORMIGA now supports semantic reasoning,

temporal ordering, and battery-aware agent selection, enabling a more intelligent and context-sensitive coordination strategy for multi-agent systems. Moreover, by leveraging its modular monitoring subsystem, the framework provides the necessary hooks for real-time failure detection and adaptive replanning, ensuring mission continuity in dynamic field conditions.

3.2. Use Case: FEROX

The FEROX project (https://ferox.fbk.eu/, accessed on 12 April 2026) is a multidisciplinary initiative focused on improving the productivity and safety of wild berry and mushroom collectors operating in remote Nordic environments. Central to its vision is human–robot collaboration (HRC), where drones assist human workers through shared task execution (in Figure 1). This collaboration enhances both operational efficiency and system resilience in sparse, challenging terrains.



Figure 1. Preliminary real-world experiments executed in Rovaniemi (Finland). Two autonomously operating drones have been deployed to execute a series of tasks used to support berry picking operations, including the execution of specific tasks coordinated by FORMIGA, and to provide feedback to the pickers.

FEROX aims to deliver tangible benefits such as increased harvest yield, improved product quality, reduced physical strain on pickers, greater trust in robotic systems, and enhanced safety via autonomous support and monitoring. Human interaction with the robotic fleet is mediated through a mobile application that allows users to request services or respond to system-generated triggers. The drones provide navigational and logistical support, while task coordination and decision-making are managed by the FORMIGA fleet manager, which confirms and orchestrates robotic actions.

Beyond task allocation, FEROX incorporates several advanced research domains, including AI for yield mapping [32], human activity recognition [33], and studies in human factors and ethics [34].

A cornerstone of the project is the FEROX Simulator [35], an immersive, multi-agent simulation environment designed to model berry-picking scenarios. Built with ROS and Unity, the simulator combines physical simulation, task logic, and real-time HRC dynamics. Unity handles 3D physics and visualisation, while ROS manages drone behaviour and autonomous control (in Figure 2). The two systems are synchronised through the ROS-TCP Connector, allowing real-time feedback between virtual agents and their control logic.

Agents in FEROX are divided into:

- Human agents: Pickers harvest berries and interact with drones; Watchers oversee operations and validate key steps.
- Robotic agents: Lightweight drones (LWDs) explore and map the environment; heavy-lift drones (HWDs) handle transport, assistance, and navigation support.

Tasks are distributed and executed using the FORMIGA framework. Currently, these tasks are defined as atomic units comprising a sequence of actions to be performed by a single agent. Examples include reconnaissance, berry transport, or lost picker recovery. Task allocation relies on agent availability and proximity, with the nearest free drone

selected for execution. While effective for basic workflows, this strategy lacks the flexibility to manage more complex missions that require multiple agents, interdependent steps, or synchronisation between human and robotic roles.



Figure 2. General overview of the FEROX simulator.

In this work, we extend the task model by introducing composite tasks—higher-level goals decomposed into interrelated subtasks, each assigned to a different agent. Each subtask comprises a sequence of actions to be executed by its respective agent. These can span different agents and include logical dependencies and execution flows within subtasks. For instance, a single harvesting task may include subtasks such as environment exploration, coordinated pickup, and safety validation, each handled by different agents. Our approach enhances the expressiveness and adaptability of FORMIGA by enabling structured multi-agent task allocation and supervised execution, laying the foundation for scalable, resilient, and intelligent coordination in real-world HRC scenarios.

4. Methodology

Our methodology defines a structured pipeline for intelligent task allocation in heterogeneous multi-agent field operations, combining LLM-based task decomposition with optimisation-based scheduling, as illustrated in Figure 3. The process begins with task definitions written in a high-level Pythonic syntax, where each global task encapsulates a sequence of subtasks. These subtasks are individually extracted and passed to an LLM, which analyses the functional semantics of each subtask to identify execution-flow elements, such as conditions, loops, parallelism, and synchronisation points. The LLM returns a structured JSON representation for each subtask, including semantic labels, required actions, and execution conditions. The parsed outputs are then merged into a single global DAG, where nodes represent actions and edges encode precedence constraints (i.e., task dependencies). Estimated execution durations are stored as edge weights associated with the dependent nodes. This DAG serves as the input to a GA, which computes an optimised assignment of subtasks to available agents, aiming to balance workload, minimise overall execution time, and adhere to agent-specific constraints and capabilities.

4.1. Problem Formulation

To rigorously define the Multi-Robot Task Allocation (MRTA) problem, we formalise the inputs, outputs, and constraints using mathematical notation. Our objective is to determine an optimal assignment between a set of heterogeneous agents and a structured task graph with hierarchical decomposition.

- Agents ($\mathcal{A}$): Let $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ denote the set of m heterogeneous agents (e.g., humans and robots). Each agent a_i is characterised by its capability set C_i, battery capacity B_i, and spatial position P_i.

- Tasks and Subtasks ($\mathcal{T}, \mathcal{S}$): The global mission is decomposed into a set of high-level tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$. Each task t_j is further decomposed into a set of subtasks $\mathcal{S}_j = \{s_{j1}, s_{j2}, \ldots\}$.
  Each subtask $s \in \mathcal{S}$ corresponds to an executable sequence of low-level actions (e.g., `TakeOff → GoTo → LandIn`) that must be executed by a single agent to preserve execution consistency.
- Action-level DAG (G): The execution structure is represented as a Directed Acyclic Graph $G = (V, E)$, where nodes V correspond to atomic actions and edges E encode precedence and synchronisation constraints between actions. Subtasks correspond to contiguous subgraphs within G.
- Execution Time (D): For each subtask s assigned to agent a_i, the estimated duration $D(s, a_i)$ is computed as the aggregated execution time of its constituent actions, including both fixed action durations and travel costs.

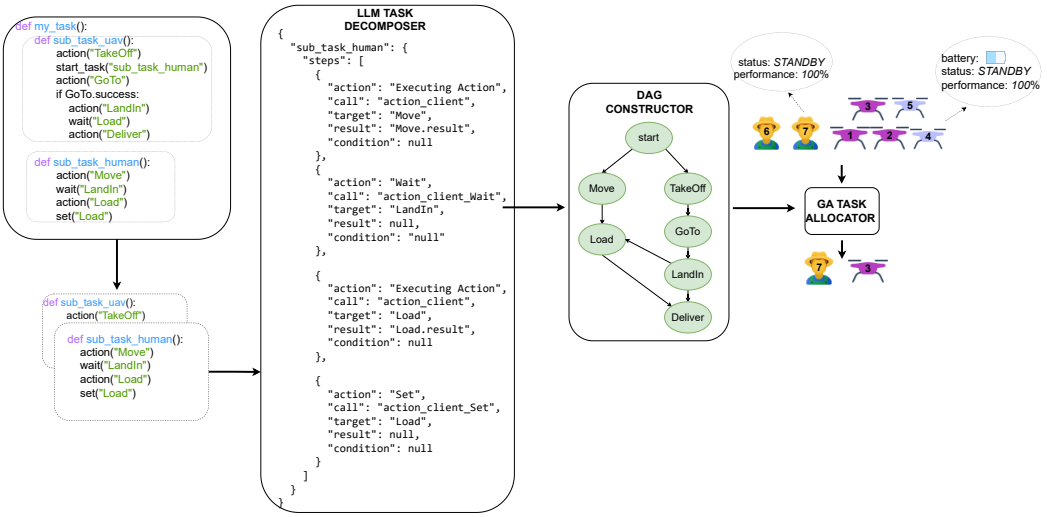


Figure 3. Overview of the proposed task allocation pipeline. High-level task scripts are parsed into structured representations using the DeepSeek R1 LLM. The outputs are merged into a global DAG that models action dependencies and execution flow. A Genetic Algorithm then computes an optimised multi-agent allocation strategy based on this graph. The numbers 1–7 represent the unique identification of the individual agents within the fleet.

Let $\mathcal{A}_s \subseteq \mathcal{A}$ denote the set of feasible agents for subtask s, determined by capability and resource constraints. Let $X_{i,s} \in \{0, 1\}$ be a binary decision variable indicating whether agent a_i is assigned to subtask s.

To ensure a feasible allocation, the following constraints must be satisfied:

1. Capability Constraint:
$$X_{i,s} = 1 \implies \text{Req}(s) \subseteq C_i \tag{1}$$
2. Uniqueness Constraint:
$$\sum_{a_i \in \mathcal{A}_s} X_{i,s} = 1, \quad \forall s \in \mathcal{S} \tag{2}$$
3. Battery Constraint:
$$\sum_{s \in \mathcal{S}} X_{i,s} \cdot D(s, a_i) \leq B_i, \quad \forall a_i \in \mathcal{A} \tag{3}$$
4. Precedence Constraint: Let $ST(v)$ denote the start time of action node $v \in V$. Then:
$$ST(v_k) \geq ST(v_j) + \text{dur}(v_j), \quad \forall (v_j, v_k) \in E \tag{4}$$

The primary objective is to minimise the overall mission makespan, approximated as the maximum workload assigned to any agent:

$$C_{\max} = \max_{a_i \in \mathcal{A}} \left(\sum_{s \in \mathcal{S}} X_{i,s} \cdot D(s, a_i) \right) \tag{5}$$

In addition to makespan minimisation, soft constraints such as energy efficiency and feasibility penalties are incorporated in the optimisation stage. This formulation aligns with the GA-based optimisation described in the following section, where allocation is performed at the subtask level while preserving action-level dependencies through the DAG.

4.2. Task Decomposition with LLM

Global tasks in our system are defined as Python functions, either manually written by developers or synthesised by an LLM. These task scripts follow a specific structural convention. Each task represents a collaborative mission involving multiple agents that must operate in parallel, often requiring synchronisation or inter-agent dependencies. For instance, one agent may need to wait until another completes a specific action before proceeding. To capture this structure, each agent's role within a collaborative task is explicitly defined as a subfunction within a global function. These subfunctions are intentionally designed to model distinct agent-specific actions, coordinated within a broader collaborative mission. This modular design enables the system to extract, interpret, and allocate subtasks effectively during the decomposition and planning stages.

An example of such a task structure is shown in Figure 4, where subfunctions encapsulate sequences of actions assigned to individual agents. To avoid ambiguity, we refer to these subfunctions as subtasks, and the lower-level calls within them such as A, B, C as actions, which represent concrete operations to be executed by the agents. These actions are invoked using the action_client function, a built-in ROS utility. This function dispatches execution goals to the relevant agent's action server, specified by AgentType and AgentName, and can include additional arguments, such as target coordinates or objects. To enforce sequential execution within a subtask, each action_client call is followed by a polling pattern (while action.result is None: pass) that ensures the next action only begins after the current one completes.



Figure 4. An example of a collaborative task structure in Python, where each subfunction corresponds to a specific subtask to be assigned to an individual agent.

The parallelism between subtasks is managed using the Python threading module, allowing multiple agent-specific tasks to run concurrently. However, these subtasks are not always independent; some may rely on specific steps in others before they can begin. To handle such dependencies, action_client_Set and action_client_Wait functions are used to coordinate the execution flow between threads. Specifically, action_client_Set signals that a particular action has completed, while action_client_Wait blocks the execution of a subtask until that signal is received. This mechanism allows developers to model complex inter-agent dependencies while maintaining clear and manageable task definitions.

To ensure clarity and modularity, all subtasks must be explicitly defined within the global task function. This design constraint not only promotes a structured representation of agent-specific behaviours, but also simplifies the subsequent decomposition pipeline. Rather than processing the entire global function at once, each subtask is individually passed to the LLM, allowing the model to focus solely on the logic and flow of a single agent's responsibilities. This separation avoids confusion from shared parameter names or intertwined control flows, and enables more accurate parsing of execution semantics. Once each subtask's logic is extracted by the LLM, synchronisation constructs such as action_client_Set and action_client_Wait, which indicate inter-task dependencies as explained above, can be handled deterministically in a post-processing step. This hybrid approach balances the strengths of LLM-based semantic parsing with deterministic rule-based dependency extraction.

With each subtask modularised, the next step involves transforming its procedural logic into a structured representation. This transformation is achieved via LLM prompting. Each subtask is used as input to construct a detailed prompt containing specific instructions that guide the LLM in generating a structured representation of the execution and logic flow in JSON format. This structured format serves as an intermediate representation that abstracts away implementation details while preserving the control logic, enabling subsequent components to reason symbolically about task dependencies, parallelism, and execution flow. The expected output follows this schema:

```
{
  "function_name": {
    "steps": [
      {
        "action": "Action keywords",
        "call": "Function/Method Name",
        "target": "Targeted Function, Thread",
        "result": "Result Variable",
        "condition": "Condition (if any)"
      },
      ...
    ]
  }
}
```

Each field in the generated JSON output corresponds to a key aspect of the function's execution logic. The action field defines the type of operation being performed, such as invoking an action, waiting for a result, setting or waiting for a synchronisation signal, or starting a thread. To ensure consistency and prevent the LLM from generating arbitrary or inconsistent action labels, a predefined set of allowed action keywords is enforced. Specifically, "Hold" represents a waiting state, typically used within a while loop to check for a result; "Executing Action" indicates a function call via action_client; "Threading" denotes the initiation of a new thread; "Wait" refers to a synchronisation point where the system waits for an event signal; and "Set" signifies the setting of an event signal. The call

field specifies the method or function being invoked, while the target field identifies the specific goal, thread, or event signal associated with the action. The result field captures the output of the operation, enabling downstream steps to use this data as input or for condition checking. Finally, the condition field encodes any logical constraint (e.g., if or while expressions) that determines whether a particular step should be executed. This structured format effectively captures the original function's control and data flow, enabling downstream components to reason about dependencies, concurrency, and conditional logic in multi-agent task planning scenarios.

To extract the structured execution logic from raw Python functions, we employ a few-shot prompting strategy using the DeepSeek-R1-Distill-Qwen-14B (https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-14B, accessed on 12 April 2026) model. The prompt includes a set of six carefully curated examples (six-shot), each demonstrating how specific function constructs (such as action_client calls, while loops, conditional branches, and threading.Thread) are to be translated into a structured JSON format. The prompt begins with a detailed explanation of the allowed action keywords, along with rules for extracting target names, handling conditions, and preserving execution order. These guidelines are followed by illustrative examples that cover a diverse range of patterns found in multi-agent task planning scripts, including sequential logic, parallel thread execution, and event-based synchronisation. This few-shot approach enables the model to generalise across unseen functions while maintaining consistency and interpretability in the output schema. A representative version of the exact prompt template used for LLM-based decomposition is provided in Appendix A to support reproducibility.

To ensure consistent and structurally valid outputs, LLM inference parameters were configured to minimise stochasticity. In particular, a low temperature setting ($T = 0.3$) was used to reduce variability in the generated outputs, while nucleus sampling ($\text{top}_p = 0.3$) was constrained to promote stable and reproducible DAG structures. This configuration prioritises deterministic parsing behaviour over generative diversity, which is critical for reliable downstream optimisation. Empirically, higher temperature values were found to introduce structural inconsistencies, such as missing dependencies and incorrect action ordering, which negatively impact DAG validity and, consequently, allocation feasibility.

Extracting a valid DAG from procedural task descriptions is a non-trivial reasoning task for LLMs, and errors may occur. We observe two common failure modes. First, the model may fail to capture implicit precedence relations, resulting in missing dependencies (false parallelism), where dependent subtasks are incorrectly treated as independent and may be scheduled concurrently. Second, in tasks involving repeated or structurally similar actions (e.g., multiple `GoTo` calls), the model may merge distinct actions into a single node, leading to action ambiguity and loss of structural fidelity.

These structural inconsistencies directly propagate to the optimisation stage. Since the GA relies on the DAG representation to estimate execution costs and enforce feasibility, missing dependencies may result in infeasible schedules, while merged nodes can lead to underestimation of task duration and distorted allocation decisions. This highlights the importance of accurate decomposition, as errors at this stage affect both the validity of the execution plan and the quality of the resulting agent assignments.

4.3. DAG Construction

Once each subtask has been modularised and transformed by the LLM into a structured JSON format, the next stage involves constructing a DAG that captures both logical and temporal dependencies across all agent actions. This DAG forms the structural backbone for downstream reasoning, allocation, and execution scheduling, ensuring that all execution constraints, both intra- and inter-subtask, are faithfully represented.

The construction process integrates three primary categories of dependencies: sequential action ordering within subtasks (Figure 5), inter-subtask thread initiation (Figure 6), and inter-thread synchronisation through event signalling (Figure 7).

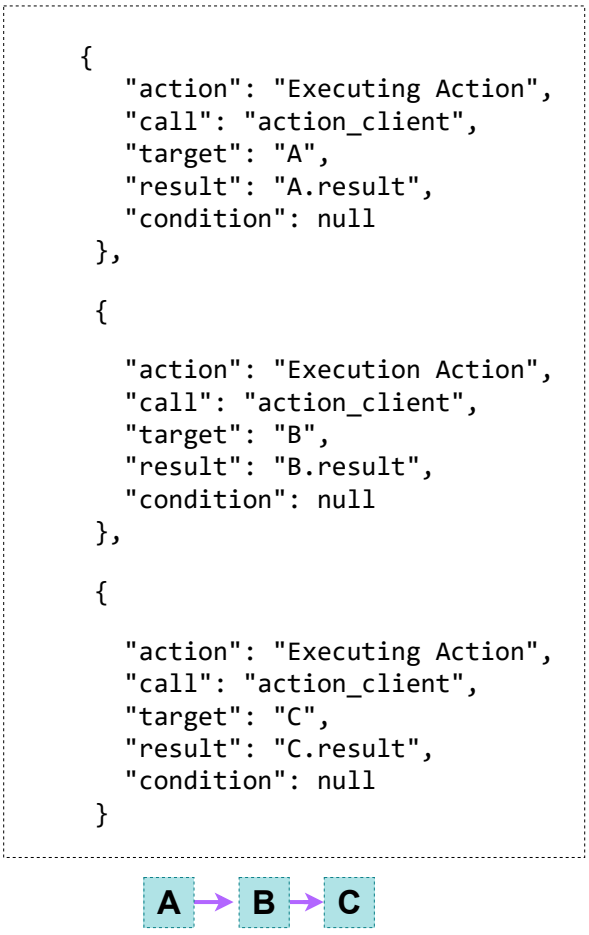


Figure 5. Subtask sequencing.



Figure 6. Inter-subtask thread initiation.

Subtask sequencing dependencies arise from the natural ordering of operations within a single subtask. Actions listed in sequence are modelled with directed edges that enforce strict execution order. In the presence of conditional branching (e.g., `if-else` structures), dependencies are added from the root condition to the first action of each branch, with each branch's actions connected linearly. The final action(s) of all branches are retained as merge points for downstream connections, thus preserving the control flow continuity beyond conditional structures.

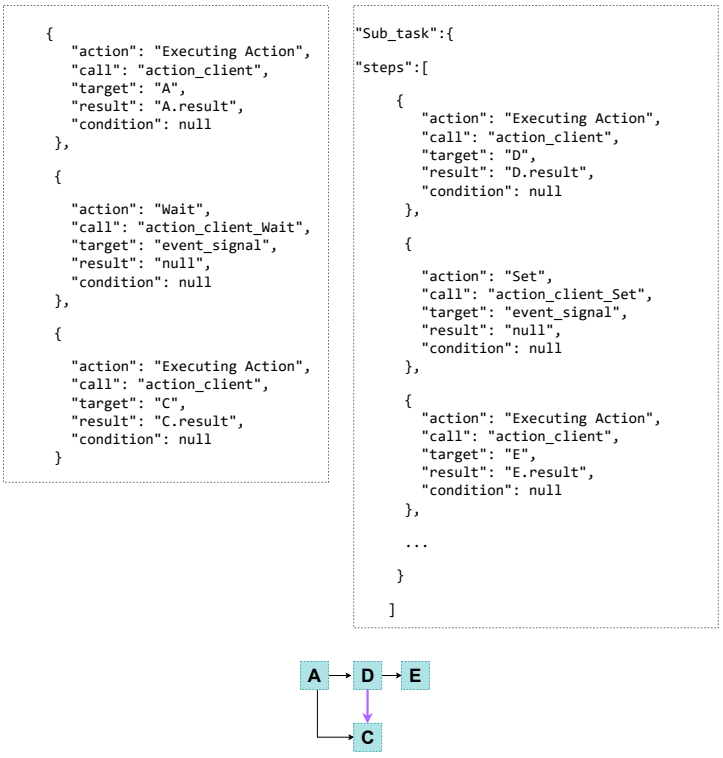


Figure 7. Inter-thread synchronisation.

Inter-subtask thread initiation occurs when a subtask explicitly spawns another as a concurrent execution thread. This is represented by symbolic references in the structured JSON, indicating the start of a new thread. In such cases, a dependency edge is created from the initiating action to the first action of the newly launched subtask. This models parallel execution contexts while preserving the initiating relationship between tasks.

Synchronisation dependencies are introduced to model temporal constraints across concurrent threads. These are expressed using event signalling semantics, where one thread sets an event and another waits for it. A dependency edge is added from the action preceding the event set to the action following the wait, ensuring that the waiting thread only proceeds once the required condition is met. This enables accurate modelling of coordination points in distributed or multi-agent workflows. The systematic procedure for integrating these disparate dependency types into a unified structure is formalised in Algorithm 1. This algorithm details the deterministic mapping of ordered action lists, conditional branches, and synchronisation pairs into the final action-level DAG (G).

Acyclicity is inherently guaranteed in our DAG construction due to the design of the task specification language and action server interaction model. Specifically, looping constructs, such as `for` or `while`, are avoided in task definitions. Instead, repetitive behaviours are handled internally by action servers, which manage their own state progression and return upon completion (e.g., after confirming arrival at a waypoint). As a result, the task code itself is strictly forward-executing, and the generated DAG remains free of cycles by construction.

The final DAG is assembled by aggregating all intra-subtask, inter-thread, and synchronisation dependencies into a single unified graph. The result is a coherent and semantically rich representation of the entire task plan, capturing both sequential and concurrent control flows. This graph serves as the input to the subsequent optimisation phase, where agent–task allocation and execution scheduling are performed.

Algorithm 1 Construction of the action-level DAG from LLM outputs

Require: Set of subtasks $\mathcal{S}$, structured JSON output for each subtask
Ensure: Action-level DAG $G = (V, E)$
Initialise $V \leftarrow \emptyset$, $E \leftarrow \emptyset$
for all subtasks $s \in \mathcal{S}$ **do**
  Parse ordered action list $A_s = [a_1, a_2, \ldots, a_k]$
  for all actions $a_r \in A_s$ **do**
    Add node a_r to V
  end for
  for $r = 1$ to $k - 1$ **do**
    Add sequential edge (a_r, a_{r+1}) to E
  end for
  if s contains conditional branches **then**
    Add branch edges from the condition node to the first action of each branch
    Add merge edges from terminal branch nodes to the next valid successor
  end if
end for
for all thread-start relations identified in JSON **do**
  Add edge from the initiating action to the first action of the spawned subtask
end for
for all matched Set/Wait synchronisation pairs **do**
  Add synchronisation edge from the action preceding `Set` to the action following `Wait`
end for
return $G = (V, E)$

4.4. GA for Multi-Agent Task Allocation

To solve the multi-agent task allocation problem in dynamic and heterogeneous environments, we propose an evolutionary optimisation approach grounded in GAs. The method is designed to address real-world constraints arising in robotic fleet management scenarios, involving diverse agents, such as drones and human operators. This heterogeneous setting introduces structural constraints into the allocation process that are not easily addressed by traditional optimisation methods.

As previously stated, the high-level mission is first decomposed into a sequence of interdependent subtasks, which are structurally represented as a DAG. Each subtask is annotated with its required agent type and estimated execution time, which is computed based on both the agent's capabilities (e.g., speed) and the contextual parameters of the action—such as the distance to be travelled in an operation. Each subtask's actions are interpreted to dynamically generate a list of feasible candidate agents, namely those that are not currently executing another task and have sufficient battery levels. Agent eligibility is determined by matching action requirements with agent capabilities. For instance, in our use case with the FEROX system, only LWDs possess the ability to perform exploration tasks, while HWDs are restricted from such operations. If a subtask includes the sequence TakeOff $\rightarrow$ GoTo $\rightarrow$ Explore $\rightarrow$ LandIn, the candidate pool is filtered to include only available LWDs that can perform all actions in the sequence.

For each candidate agent, the execution time of every action within a subtask is computed based on predefined action durations and proximity-based estimations. Certain actions, such as TakeOff or LandIn, have fixed durations (e.g., TakeOff takes approximately

3 s), while other movement-related actions, like GoTo, depend on the spatial distance between the agent's current location and the task location. This allows the system to prefer agents that are not only capable of performing the required actions, but are also closer to the task location, thereby minimising energy consumption and improving overall mission efficiency. These action-level durations are aggregated for each candidate agent to determine the total expected subtask execution time, which is then encoded in the task DAG as edge weights. These weights are later used by the genetic algorithm during optimisation to evaluate and select high-quality task-to-agent assignments.

We employ a GA to explore the search space of possible agent-to-subtask assignments. Each individual in the GA population represents a complete allocation, encoded as a list where each element denotes the agent assigned to a particular subtask $s \in \mathcal{S}$. The GA aims to find an allocation that minimises the overall mission makespan, adheres to battery and feasibility constraints, and considers proximity-based energy costs as a secondary but meaningful optimisation signal.

The fitness function F formulated in Equation (6), integrates multiple components that capture the key constraints and objectives of the allocation problem. The primary metric is the makespan $C_{\max}$, computed as the longest path through the DAG when execution times are adjusted based on agent capabilities. This ensures that agent–task compatibility and task dependencies are jointly optimised. To enforce feasibility, additional penalty terms are introduced: P_b, the battery penalty, which quantifies violations of battery constraints, and E, the proximity-aware energy cost, which captures total energy consumption across all agents. These penalties are scaled to discourage infeasible solutions early in the search process.

In addition, the energy cost component E serves as a tie-breaker when multiple agents are equally capable of executing a subtask within acceptable time and resource limits. For such cases, the algorithm prefers the agent with the shortest estimated path distance to the task's location. This cost is not used to override critical performance constraints but helps improve overall efficiency by reducing unnecessary travel and preserving energy reserves—an essential factor in field deployments of mobile robots.

$$F = C_{\max} + P_b + E \tag{6}$$

Each component of the fitness function F is calculated as follows:

- Makespan: The makespan $C_{\max}$ represents the total time taken by the slowest agent to complete all assigned subtasks, as formally defined in Equation (5), where $\mathcal{A}$ is the set of agents, $\mathcal{S}_a$ is the set of subtasks assigned to agent a, and each $D(s, a)$ is computed as the critical path length of subtask s's DAG, considering dependencies and edge weights.
- Battery Penalty: To enforce battery constraints, a penalty is applied whenever an agent's total assigned subtask duration exceeds its available battery capacity. For each agent a, the penalty is calculated as:

$$\text{Penalty}(a) = \max(0, T(a) - B(a)) \cdot \lambda_b, \tag{7}$$

where $T(a) = \sum_{s \in \mathcal{S}_a} D(s, a)$ is the total subtask duration assigned to agent a, $B(a)$ is the battery capacity of agent a, and λ_b is the battery penalty scaling factor.
The overall battery penalty P_b term for the fitness function F is then the sum of individual penalties:

$$P_b = \sum_{a \in \mathcal{A}} \text{Penalty}(a). \tag{8}$$

- Energy Cost: The energy cost E encourages the selection of agents that are spatially closer to the task locations, helping to minimise unnecessary travel. It is computed as

a soft cost proportional to the total task duration of each agent's assigned subtasks, with emphasis on spatial actions such as GoTo. Formally,

$$E = \sum_{a \in \mathcal{A}} T_{\text{est}}(a) \cdot \epsilon \tag{9}$$

where $T_{\text{est}}(a)$ is the sum of estimated execution times of all actions assigned to agent a, and ϵ is a scaling factor representing energy consumption per time unit.

The optimisation employs a GA implemented via the DEAP framework (https://github.com/DEAP/deap, accessed on 12 April 2026), with a population size of 100 individuals evolved over 300 generations. Each individual represents a candidate task-to-agent assignment, encoded as a list of agent indices. We use tournament selection (tournament size = 3) to balance exploration and exploitation, with crossover and mutation probabilities set to 0.7 and 0.2, respectively.

The selected GA parameters were determined based on preliminary sensitivity analysis to balance solution quality and computational cost. In particular, smaller population sizes were observed to lead to premature convergence, while larger populations increased runtime without significant improvements in allocation quality. Similarly, the number of generations was chosen to ensure stable convergence across different scenario configurations. The crossover and mutation probabilities follow standard evolutionary optimisation practices and were empirically validated to maintain sufficient diversity without destabilising feasible assignments.

The following custom operators ensure feasibility:

- Crossover (cx_unique): Exchanges assignment segments between parents while preserving agent–task validity.
- Mutation (mutate_unique): Randomly swaps agents for tasks with a 20% probability, constrained to eligible agents.

To accurately estimate the temporal cost of an agent traveling between two geographical coordinates, we employ the Haversine formula to compute the great-circle distance between two points on the Earth's surface. This approach accounts for the Earth's curvature and is more precise than simple Euclidean approximations for large-scale spatial distributions.

Given a source location with latitude and longitude (ϕ_1, λ_1) and a target location (ϕ_2, λ_2), where all angles are expressed in radians, the procedure is as follows:

1. Compute the differences in latitude and longitude:

$$\Delta\phi = \phi_2 - \phi_1, \quad \Delta\lambda = \lambda_2 - \lambda_1$$

2. Apply the Haversine formula to compute the central angle c between the two points:

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)$$

$$c = 2 \cdot \arctan 2\left(\sqrt{a}, \sqrt{1-a}\right)$$

3. Multiply the central angle c by the Earth's radius R to obtain the great-circle distance d:

$$d = R \cdot c$$

where $R = 6{,}366{,}707.0195$ m is the mean Earth radius used in this study.

4. Finally, given an agent's velocity v (in meters per second), the travel time t between the two points is computed as:

$$t = \frac{d}{v}$$

Figure 8 illustrates the overall workflow of the GA used for task-to-agent assignment. The algorithm begins by generating an initial population, where each individual represents a candidate solution—a mapping of tasks to agents. Each individual is evaluated using a multi-objective fitness function. Selection is performed using a tournament-based strategy to identify promising individuals for reproduction. A custom crossover operator is applied to preserve task–agent feasibility, while mutation introduces diversity by altering agent assignments within eligibility constraints. The population is iteratively evolved across multiple generations until convergence criteria—such as stagnation or maximum generation count—are met. Parameter values such as population size, number of generations, and selection strategy are adjustable based on scenario complexity, and are detailed in the experimental setup (Section 5.2).

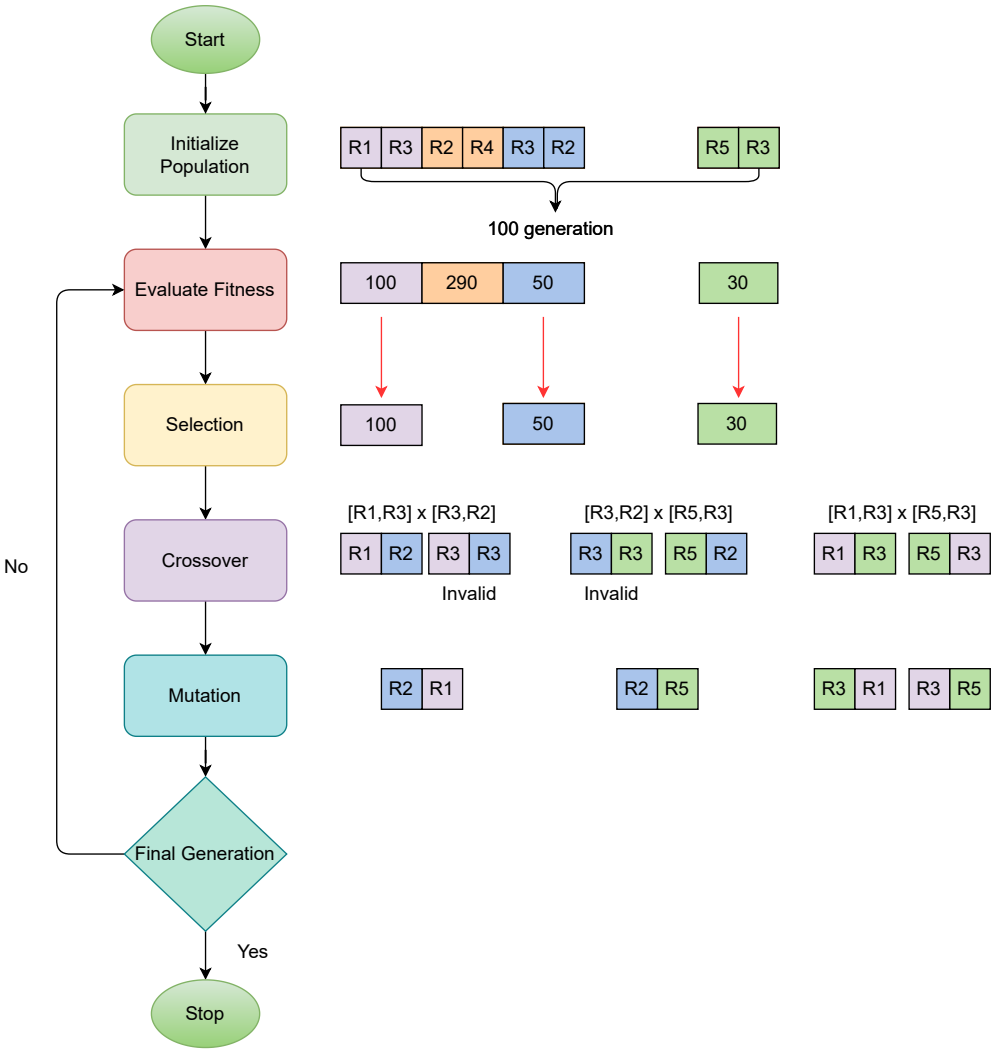


Figure 8. Workflow of the genetic algorithm used for task-to-agent assignment.

4.5. System Integration and Adaptive Resilience

The proposed architecture operates as a closed-loop supervisor, where task allocation, monitored execution, and recovery are tightly integrated into a unified decision–execution loop. While the GA computes the initial optimal assignment over the task DAG, the

system's robustness in dynamic environments is achieved through an LLM-driven recovery module that enables adaptive, localised replanning during execution.

4.5.1. Failure Monitoring and Context Gathering

During execution, the FMS employs a uniform monitoring pattern for all agents. Each subtask is wrapped with three control signals: (i) *stop_event*, enabling graceful halting when a recovery plan is available; (ii) *failure_event*, triggered upon action failure (e.g., `FAILED`/`ABORTED` states from ROS action servers); and (iii) *action_interrupt*, allowing immediate preemption when required.

Upon failure, the Replan module extracts a structured execution context, including the current task code, agent states (e.g., position, availability, and progress), and the live DAG. Crucially, the DAG explicitly encodes completed, active, and pending nodes, enabling the system to preserve already executed actions and constrain recovery to only the affected subgraph.

4.5.2. Semantic Recovery via LLM Reasoning

The collected context is encoded into a structured JSON prompt and provided to the LLM, which is framed as an adaptive multi-agent planner. Rather than generating a plan directly, the model is guided to first reason about the failure scope—determining whether it is localised or requires coordinated replanning—while respecting task dependencies encoded in the DAG. A representative version of the exact prompt template is provided in Appendix B. The LLM produces a structured recovery plan consisting of: (i) updated task code defining revised action sequences, and (ii) metadata directives such as *agents_to_stop*, *agents_to_add*, and *new_sync_events*.

The generated plan is validated and integrated into the execution layer through DAG-aware reconciliation, ensuring that completed actions are preserved and only the affected subgraph is modified. To guarantee safe execution, a selective control policy is applied: unaffected agents continue execution, while impacted agents are either gracefully stopped or immediately interrupted depending on synchronisation constraints and resource conflicts. In human–robot collaboration scenarios, conservative policies are enforced, prioritising non-interruption of human tasks and allowing near-complete actions (e.g., above 80% progress) to finish before recovery is applied.

Finally, the recovery code undergoes schema validation and dry-run compilation prior to execution. If inconsistencies are detected, the system re-prompts the LLM with a compact error report, ensuring that the injected recovery routines are both executable and semantically consistent with the system state.

4.5.3. Computational Complexity of the Pipeline

The computational cost of the proposed pipeline can be analysed in three stages: LLM-based decomposition, DAG construction, and GA-based allocation.

LLM decomposition: Let S denote the number of subtasks and L the average token length of each subtask prompt. Since each subtask is processed independently, the decomposition cost scales approximately linearly with the number of subtasks, i.e., $O(S \cdot L)$ at the prompting level, excluding hardware-dependent model inference constants.

DAG construction: Let $|V|$ and $|E|$ denote the number of action nodes and edges in the final DAG. The deterministic graph construction phase requires a single pass over the structured JSON outputs and synchronisation pairs, yielding a complexity of $O(|V| + |E|)$.

GA-based Allocation and Scalability: The complexity of the optimisation stage is approximately $O(G \cdot P \cdot S)$, where G is the number of generations, P is the population size, and S is the number of subtasks. Additionally, the initialisation phase involves candidate-

agent filtering and duration estimation with a cost of $O(M \cdot S)$, where M is the number of agents.

The scalability of the framework with respect to larger agent populations (up to $M = 1000$) is supported by the subtask-level abstraction. Instead of assigning agents to individual atomic actions, the optimisation is performed over a reduced set of subtasks ($S \ll N$, where N is the number of action-level nodes in the DAG). In practice, S remains relatively small (e.g., $10 \leq S \leq 50$), which bounds the size of the optimisation search space.

As the number of agents increases, the size of the candidate pool for each subtask grows, which improves the likelihood of selecting better (e.g., closer or more capable) agents. This is reflected in the relatively stable assignment accuracy across different fleet sizes.

However, the computational cost of the optimisation increases with the number of agents due to the expanded candidate evaluation space. As shown in the experiments, while the framework maintains solution quality, the runtime increases with larger agent populations.

5. Experiments

To evaluate the proposed system's effectiveness in task representation and allocation, we conducted a series of five experiments, each targeting a distinct aspect of the pipeline. The first experiment investigates the ability of LLMs to convert Python-based task code into structured execution flows in the form of JSON objects. We assessed multiple model configurations using tasks of varying complexity. The second experiment focuses on the GA component, analysing how different population sizes and energy cost weight parameters influence agent selection diversity and convergence. The third experiment builds on the GA results to evaluate scalability, testing the robustness and performance of the system as the number of available robotic agents increases from 10 to 1000. The fourth experiment—a system-level assessment—investigates the behaviour of the complete integrated loop, quantifying human–robot synergy and mission resilience under mid-execution failures. Finally, the fifth experiment provides a comparative benchmark, evaluating the role of dependency-aware modelling by contrasting our GA-based approach against three alternative strategies: Particle Swarm Optimisation, Greedy local allocation, and a Sequential Single-Item auction baseline. This experiment specifically quantifies the performance leap achieved through global DAG reasoning compared to local decision-making under resource constraints.

Together, these experiments provide a comprehensive assessment of both the reasoning and optimisation components of our multi-agent task allocation framework.

5.1. From Task Code to Execution DAG

The objective of this experiment is to evaluate the ability of LLMs to extract structured execution flows from Python-based task definitions used in multi-agent robotic systems. Specifically, we aim to convert raw task code including elements such as sequential actions, multi-threading, synchronisation primitives, and conditional logic into a DAG that captures action dependencies.

This evaluation serves two purposes: (1) to determine whether LLMs can replace custom parsers or rule-based systems in understanding procedural logic in robotics workflows, and (2) to explore the impact of model scale and prompt design on accuracy and generalisability. We test both subtask-based and full-task-based prompting strategies using two models of different sizes (7B and 14B), assessing their ability to infer correct dependencies under varying levels of complexity.

5.1.1. Experimental Tasks

To thoroughly assess the LLMs' performance in dependency extraction, we designed a set of three representative tasks with increasing complexity. Each task embodies flow patterns, such as sequential actions, parallel threads, synchronisation events, and conditional branches. These tasks are designed to be executed within the FEROX simulator, involving both drone and human agents, enabling comprehensive evaluation of multi-agent task coordination and synchronisation in a controlled, realistic environment.

Task 1: Basic Parallel Execution with Two Independent Agents

The first task is designed to evaluate the model's ability to accurately identify parallel execution flows involving multiple agents. In this scenario, two robotic agents (drones) independently perform delivery operations: each drone takes off, navigates to a designated location, and lands to complete its delivery. Specifically, two drones independently execute a sequence of three actions: *TakeOff*, *GoTo*, and *LandIn*. These actions are encapsulated within two distinct functions—`first` and `second`—which operate concurrently via Python threading. The `first` function runs in a separate thread and performs the actions sequentially for the first agent. Conversely, the `second` function executes sequentially on the main thread for the second agent. Notably, there is no explicit synchronisation or communication between these two concurrent threads. Both threads originate from a conceptual *Start* node, representing the initiation point of the overall task execution. The task structure is depicted as a DAG in Figure 9, illustrating the independent parallel workflows. This experimental setup assesses the model's capability to discern such parallel procedural flows within robotic operations.

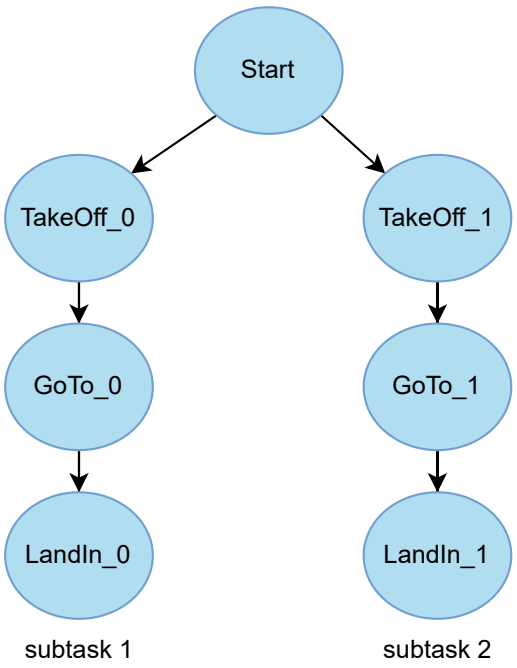


Figure 9. DAG representation of Task 1.

Task 2: Sequential Synchronisation Between Robot and Human Agents

Task 2 presents a cooperative scenario involving two different agent types,a human and a robot, with interleaved and synchronised actions that simulate a real-world agricultural operation. The workflow begins with the human agent performing a *Picking* action on the main thread, representing the manual harvesting of berries. Concurrently, a new thread is spawned to execute the robot agent's flight sequence, consisting of three sequential actions: *TakeOff*, *GoTo*, and *LandIn*, modelling the drone arriving at the harvesting site.

Importantly, the robot's *TakeOff* action depends explicitly on the completion of the human's *Picking* action, ensuring that the drone does not depart before the berries are ready to be loaded. After the drone lands at the picking site, the human proceeds with a *Loader* action, representing the loading of harvested berries onto the drone. This loading step explicitly waits for the drone's *LandIn* action to complete, reflecting the real-world constraint that loading cannot occur until the drone has safely landed. The dependencies, illustrated in Figure 10, highlight essential synchronisation points between the human and robot agents that the LLM must accurately capture to ensure correct task execution.

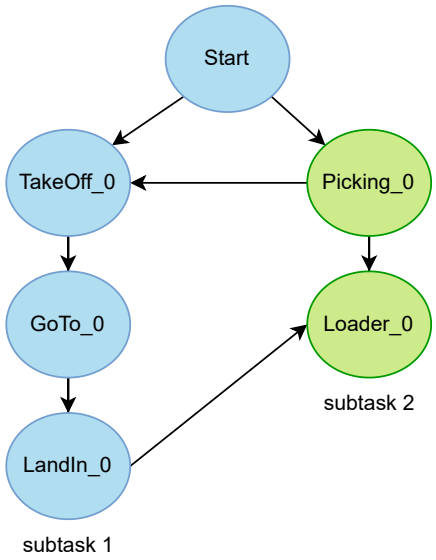


Figure 10. DAG representation of Task 2. Blue nodes denote actions performed by robotic agents, while green nodes represent actions executed by the human agent.

Task 3: Multi-Agent Task with Nested Synchronisation and Conditional Branching

The third task represents the most complex scenario in this study, involving three agents, two drones and one human, cooperating in a structured workflow with nested synchronisation and conditional branching, reflecting a realistic agricultural operation.

Initially, the primary drone undertakes an asynchronous *Explore* mission, flying through multiple GNSS waypoints to survey the berry field. This exploration runs in a separate thread and signals its completion through a synchronisation primitive, *action_client_Set(explore_complete)*. Meanwhile, the human agent navigates (*MoveTo*) to the picking site.

After arriving, the human agent waits for the drone's exploration to complete (*action_client_Wait(explore_complete)*) before commencing the *Picking* action, harvesting berries from the site. The outcome of this picking action determines subsequent steps via conditional branching: if the picking is successful (*clientPicking.get_result() == True*), a secondary drone is launched in a new thread to perform a delivery sequence consisting of *TakeOff*, *GoTo*, and *LandIn*, representing the drone arriving to collect the harvested berries.

The human then waits for this secondary drone to land before performing a *Loader* action, loading the berries onto the drone for transport. Conversely, if the picking fails, the human immediately initiates a fallback *MoveTo* action, moving to an alternative location without involving the secondary drone.

This task embodies a complex interplay of concurrency as illustrated in Figure 11, synchronisation, and conditional logic: the exploration drone surveys the field while the human approaches the site; the human's picking depends on exploration completion; the delivery drone's mission depends on successful picking; and the human's loading depends

on the delivery drone's arrival. Failure triggers a fallback human action, bypassing the delivery drone.

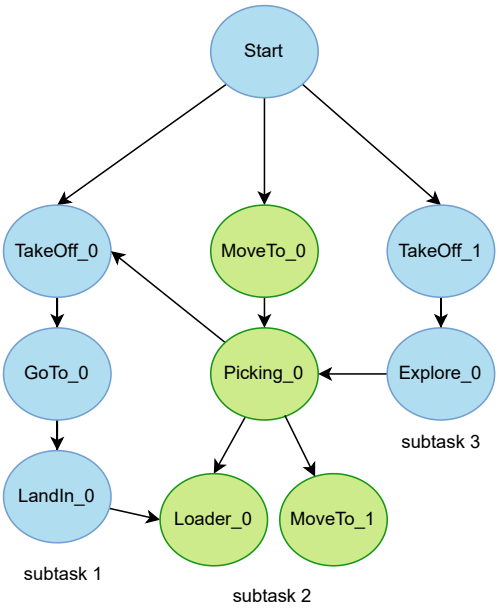


Figure 11. DAG representation of Task 3.

5.1.2. DAG Generation Setup

To evaluate the capability of LLMs in extracting execution logic from procedural task definitions, we conducted six distinct experiments. These experiments vary along two primary dimensions: the granularity of input (subtask-level versus full-task-level) and the model size (7B versus 14B parameters). Each experimental configuration was applied uniformly across three tasks to enable consistent and comparative analysis.

The experimental configurations are defined as follows:

1. Subtask-JSON-7B: Subtask-level input processed by the 7B parameter model to generate structured execution flow in JSON format.
2. Subtask-JSON-14B: Subtask-level input processed by the 14B parameter model to generate structured execution flow in JSON format.
3. Fulltask-JSON-7B: Full-task input processed by the 7B parameter model to generate structured execution flow in JSON format.
4. Fulltask-JSON-14B: Full-task input processed by the 14B parameter model to generate structured execution flow in JSON format.
5. Fulltask-DAG-7B: Full-task input processed by the 7B parameter model to directly generate a task execution DAG, represented as a list of edges within a JSON-formatted output.
6. Fulltask-DAG-14B: Full-task input processed by the 14B parameter model to directly generate a task execution DAG, represented as a list of edges within a JSON-formatted output.

In the *Subtask-JSON-7B* and *Subtask-JSON-14B* configurations, individual atomic Python functions (subtasks) were provided to the models separately. The models were prompted to generate a structured JSON output representing the internal execution flow of each subtask. These JSON outputs were then merged and transformed into DAGs using a deterministic rule-based post-processing algorithm, as described in Section 4.

In the *Fulltask-JSON-7B* and *Fulltask-JSON-14B* configurations, the complete Python function, including nested subtasks, conditionals, concurrency constructs (e.g., threading), and synchronisation mechanisms, was supplied to the models in a single prompt. The models were instructed to output a structured JSON representation of the full execution

flow. As with the subtask cases, these structured outputs were subsequently converted into DAGs using the same rule-based transformation pipeline.

In contrast, the *Fulltask-DAG-7B* and *Fulltask-DAG-14B* configurations aimed to assess whether models could bypass intermediate semantic representations and directly produce the task execution DAG. In these settings, the full procedural function was given as input, and the models were prompted to return a flat representation of the DAG—consisting of uniquely named nodes and an explicit list of directed edges capturing task dependencies—embedded within a JSON-formatted output.

All experiments employed the DeepSeek-R1-Distill-Qwen models at two different parameter scales (7B and 14B). Inference parameters were held constant across all tests, using a temperature setting of 0.3 to encourage deterministic generation and a top-p value of 0.3 to limit randomness in sampling.

Two distinct prompting strategies were used, corresponding to input granularity. For the subtask- and fulltask-JSON configurations, prompts guided the model to emit a detailed nested JSON format, including execution types, such as *Executing Action*, *Threading*, *Wait*, *Hold*, and *Set*, along with contextual metadata (e.g., called functions, conditions, and variables). These prompts included a formal schema description and six illustrative examples demonstrating varied execution flows, including sequential logic, branching, and concurrency.

For the fulltask-DAG configurations, prompts instructed the model to infer and directly output a task DAG. The expected output was a flat list of nodes with unique labels and directed edges specifying execution precedence. These prompts emphasised correct interpretation of parallelism, synchronisation, and subtask boundaries. Four curated examples were included to guide the model's understanding of DAG structure, dependencies, and concurrency semantics.

5.1.3. DAG Structural Fidelity and Accuracy Metrics

To assess the quality of the structured execution flows produced by our system, we evaluated the generated DAGs against manually annotated ground-truth representations. Each DAG consists of directed edges that represent dependencies between actions, either within a subtask (intra-subtask edges) or across subtasks (inter-subtask dependencies). Since these edges are later assigned time-based weights for task scheduling and agent allocation, the presence or absence of even a single edge can significantly impact downstream decision-making. Therefore, accurate edge reconstruction is essential.

A naive edge-level accuracy metric defined as the ratio of correctly predicted edges to the total number of edges in the ground truth may give a misleading impression of quality. For instance, consider the following ground-truth DAG:

```
{("start", "A"), ("A", "B"), ("start", "C"), ("C", "D"), ("A", "C")}
```

This structure includes two subtasks (edges `A` $\rightarrow$ `B` and `C` $\rightarrow$ `D`) and one inter-subtask dependency (`A` $\rightarrow$ `C`). If the system predicts:

```
{("start", "A"), ("A", "B"), ("start", "C"), ("A", "C")}
```

Then four out of five edges are correctly identified, suggesting an 80% edge-level accuracy. However, the missing edge `("C", "D")` renders Subtask 2 structurally incomplete and invalid. This omission is critical, as the corresponding action `D` may carry significant execution time. Its absence could result in an underestimated makespan or an incorrect agent assignment.

To more appropriately capture the impact of missing edges, we adopt a *component-level evaluation* strategy. Instead of evaluating edge-level accuracy in isolation, we structure the evaluation around logical components of the task:

- Subtasks: Each subtask is evaluated based on the correctness of its internal (intra-subtask) dependencies;
- Inter-subtask dependencies: If present, all edges representing coordination between different subtasks are grouped together and treated as an additional evaluation unit.

Let n be the number of subtasks, and let I be an indicator variable representing the presence of inter-subtask dependencies:

$$I = \begin{cases} 1 & \text{if inter-subtask dependencies exist} \\ 0 & \text{otherwise} \end{cases}$$

Each subtask and the collection of inter-subtask dependencies (if any) is considered a distinct evaluation component. A component receives a score of:

- 1, if all required edges in that component are correctly identified;
- 0, if any edge in that component is missing.

The final evaluation score is computed as:

$$\text{Score} = \frac{\sum_{i=1}^{n} S_i + I \cdot D}{n + I} \tag{10}$$

where S_i denotes the correctness score for subtask i, and D is the score for the inter-subtask dependency component.

This evaluation scheme ensures that the omission of any critical edge, whether within a subtask or between subtasks, meaningfully affects the final score. Given that the DAG is used for calculating the longest path in downstream task allocation, such omissions can significantly impact execution time estimation and agent assignment. Therefore, this method provides a balanced and task-relevant assessment of structural accuracy.

Returning to the earlier example, with $n = 2$ subtasks and inter-subtask dependencies present ($I = 1$), the scores are:

$$S_1 = 1, \quad S_2 = 0, \quad D = 1$$

Applying the formula:

$$\text{Score} = \frac{S_1 + S_2 + I \cdot D}{n + I} = \frac{1 + 0 + 1 \times 1}{2 + 1} = \frac{2}{3} \approx 66.7\% \tag{11}$$

This method better reflects the functional usability of the predicted DAG in practice. Unlike edge-level accuracy, which treats each edge equally and independently, this approach penalises structural breakages that compromise task execution. Given that each missing edge can affect the critical path in scheduling, this structure-aware evaluation provides a more realistic and task-relevant performance measure.

5.1.4. Assessment of LLM-Based Task Decomposition and DAG Accuracy

In this experiment, we evaluated four configurations of language models on their ability to convert Pythonic task code into structured execution flows represented as JSON objects and corresponding DAGs. We used three representative tasks of increasing complexity to assess both 7B and 14B versions of the DeepSeek-R1-Distill-Qwen model in two modes: Fulltask, where the entire task was provided as a single input, and Subtask, where each subfunction was processed independently.

The results, summarised in Table 2, reveal significant differences in performance across these configurations. The most reliable and scalable approach was Subtask-JSON-14B, which consistently produced accurate and well-structured outputs across all tasks.

Table 2. LLM Execution DAG Generation Results (Correct Parts Extracted/Total Parts). Parts refer to structurally significant elements such as correctly labeled action nodes and their dependencies.

Model/Mode	Task 1 (2 Parts)	Task 2 (3 Parts)	Task 3 (4 Parts)
Fulltask-DAG-7B	0/2 (0%)	0/3 (0%)	0/4 (0%)
Fulltask-DAG-14B	0/2 (0%)	2/3 (66%)	0/4 (0%)
Fulltask-JSON-7B	0/2 (0%)	0/3 (0%)	0/4 (0%)
Fulltask-JSON-14B	2/2 (100%)	3/3 (100%)	3/4 (75%)
Subtask-JSON-7B	2/2 (100%)	3/3 (100%)	0/4 (0%)
Subtask-JSON-14B	2/2 (100%)	3/3 (100%)	4/4 (100%)

Task 1 (2 Subtasks, No Dependencies)

This simplest task consisted of two independent subtasks with no conditional or threaded execution. Both subtask-level JSON extraction models, *Subtask-JSON-7B* and *Subtask-JSON-14B*, succeeded in this setting, correctly parsing each function and producing accurate execution flows and DAG outputs in JSON format.

The *Fulltask-JSON-14B* model also achieved perfect accuracy on Task 1, successfully outputting correct structured JSON despite handling the full task input. However, the *Fulltask-JSON-7B* model failed to produce any correct outputs.

For the DAG generation modes, *Fulltask-DAG-7B* failed entirely, entering a repetitive reasoning loop focused on label disambiguation without producing final results. *Fulltask-DAG-14B* showed some reasoning ability and correctly inferred label naming conventions (e.g., `TakeOff_0`), but ultimately did not generate any valid DAG output.

Task 2 (2 Subtasks, Dependencies and Threads)

In this moderately complex task involving two subtasks with simple threading, both subtask-level JSON extraction configurations, *Subtask-JSON-7B* and *Subtask-JSON-14B*, again performed perfectly. They correctly extracted sequential dependencies, handled concurrency, and produced consistent, disambiguated action labels.

The *Fulltask-JSON-14B* model also succeeded fully on this task, accurately capturing all subtasks and their dependencies in the JSON output.

Conversely, *Fulltask-JSON-7B* and *Fulltask-DAG-7B* failed entirely, repeating previous issues with structural reasoning and label confusion.

Fulltask-DAG-14B showed partial success: it understood the overall task structure, separated threads correctly, and named actions intelligently. However, its final DAG output was incomplete, missing critical dependencies such as from the "picking" subtask to the "loader," which led to an inaccurate execution graph despite correct intermediate reasoning steps.

Task 3 (3 Subtasks, Conditional Logic, Dependencies and Threads)

Task 3 posed the highest challenge, combining conditional branching, subtasks with complex interdependencies, and parallel execution threads. Among all configurations, only *Subtask-JSON-14B* successfully generated a fully correct result, correctly modelling thread separation, conditional logic and dependency chains. The resulting JSON and DAG were complete and faithful to the intended task structure.

Subtask-JSON-7B showed mostly accurate execution flow extraction but failed due to a mislabeling error in the JSON structure related to a conditional branch. This single error prevented correct parsing of the final DAG output, despite other parts of the flow being correct. This highlights a key limitation of smaller models: while capable of handling local segments, they struggle to maintain global consistency in complex conditional and threaded structures.

All *Fulltask-DAG* configurations failed to produce correct outputs. Specifically, *Fulltask-DAG-7B* generated fundamentally flawed graphs by failing to disambiguate repeated actions—labeling multiple `TakeOff` actions identically—which caused collapse of distinct dependencies into single nodes, as illustrated in Figure 12. Additionally, hallucinated nodes such as "Human", "Robot", "Wait", and "Set" appeared alongside invalid edges (e.g., `GoTo` → `Picking`), undermining DAG correctness.

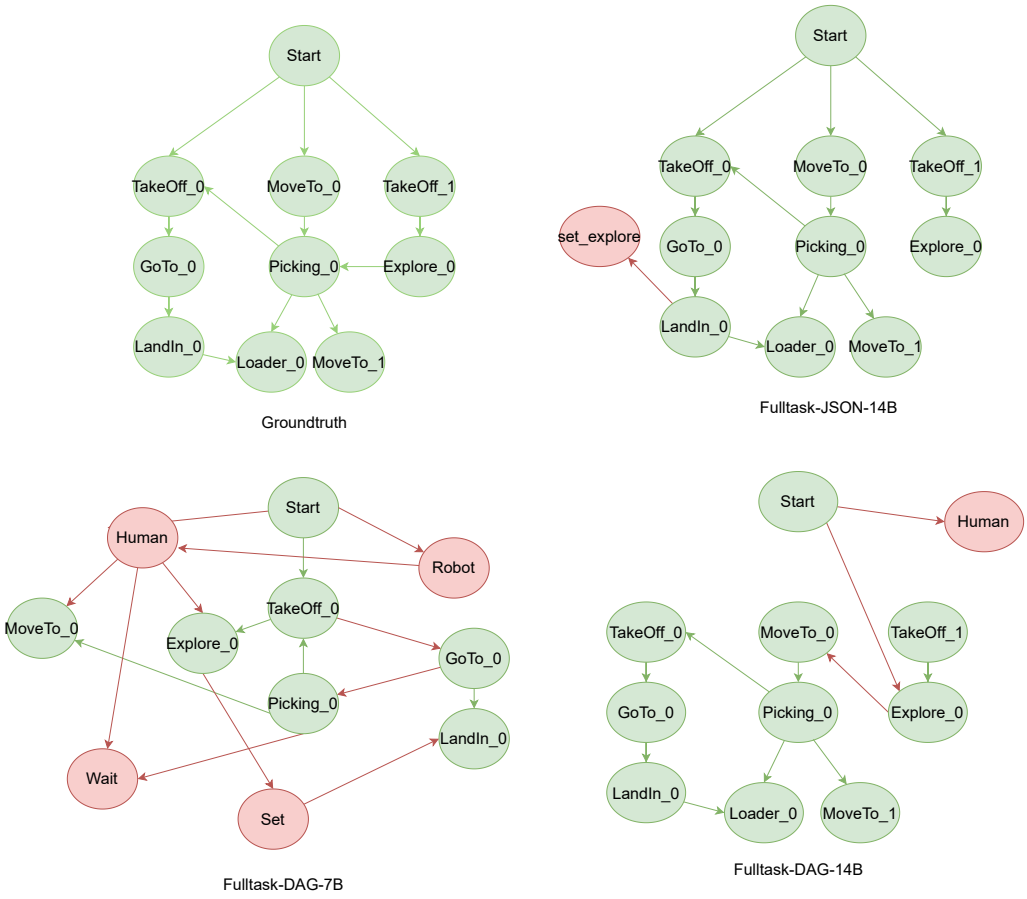


Figure 12. DAG visualisation of the outputs from Fulltask-DAGs and Fulltask-JSON-14B for Task 3. Green nodes and arrows indicate correct actions and dependencies, while red elements represent hallucinated outputs.

Fulltask-DAG-14B demonstrated a somewhat better grasp of overall structure, successfully identifying some threads and applying disambiguated naming (e.g., `TakeOff_0`, `TakeOff_1`). However, its final output remained incomplete and inconsistent, omitting critical edges (e.g., from `Start` to subtasks), introducing hallucinated nodes like "Human" and including invalid transitions such as `Explore` → `GoTo`. Even if one were to overlook these deficiencies, critical task actions and their timing contributions would be missing, making the DAG unreliable.

Fulltask-JSON-7B and *Fulltask-JSON-14B* performed better than their DAG counterparts at the full-task level. While *Fulltask-JSON-14B* correctly identified all subtasks, it missed the dependency from "explore" to "Picking" and erroneously included a control point as a node (`set_explore`). In contrast, *Fulltask-JSON-7B* failed to produce any meaningful JSON outputs for this task.

Discussion

The experimental results clearly demonstrate that subtask-based prompting—especially when paired with a larger model like *Subtask-JSON-14B*—offers superior robustness, ac-

curacy, and scalability in generating execution DAGs for multi-agent robotic tasks. This configuration consistently produced the most reliable outputs across all task complexities, successfully modelling conditional logic, parallel execution, and inter-subtask dependencies with high structural fidelity. These findings support the hypothesis that decomposing complex procedures into localised reasoning units improves model performance while enabling post hoc integration for global correctness.

In contrast, full-task configurations exhibited significant limitations. While *Fulltask-JSON-14B* showed partial success in identifying task structure, it frequently introduced structural errors such as hallucinated control nodes (e.g., `set_explore`) and missing dependencies (e.g., `explore` → `picking`), especially under nested or interdependent logic. *Fulltask-JSON-7B* failed to produce valid outputs, underscoring the difficulty smaller models face in maintaining coherence over long input sequences. Similarly, the *Fulltask-DAG* configurations were largely unsuccessful. The 7B model repeatedly fell into reasoning loops or hallucinated elements, while the 14B model—despite detecting some parallelism and naming conventions—could not translate its reasoning into complete DAGs. These shortcomings suggest a persistent gap between internal task comprehension and structured output generation in full-task settings, potentially addressable via improved prompting strategies or architectural refinements.

These structural inaccuracies have a direct and critical downstream impact on the task allocation phase. Since the GA relies on the DAG's edge weights and connectivity to calculate the critical path, a missing dependency edge (e.g., failing to link *Picking* to *Loading*) leads to blind parallelism. In such cases, the GA treats sequential subtasks as parallelisable, resulting in a mathematically optimal but physically impossible makespan. During execution, this translates to resource conflicts or deadlocks—such as a human agent being assigned to load berries before they have been harvested. Furthermore, collapsed nodes, where multiple unique actions are merged into one due to label confusion, cause a significant underestimation of mission duration and potentially lead to over-allocating a single agent to tasks that require multiple resources.

Beyond output fidelity, subtask-based processing offers key practical benefits. Its modular design facilitates independent subtask development, simplifies debugging, and allows selective updates without reprocessing the entire task. It also supports parallel inference and reuse of verified components across different task graphs, promoting scalability and maintainability. Taken together, these advantages position subtask-based prompting—not only as the most reliable method for current systems—but also as a foundation for future development in reliable and interpretable execution-flow generation for robotic applications.

5.2. Task Allocation

This experiment evaluates the effectiveness of our GA-based task allocation method in a realistic simulation environment, given a known and correct execution DAG for each task. We aim to assess how the algorithm performs in assigning agents within a heterogeneous multi-agent system—composed of human workers and drones—based on agent capabilities, energy constraints, and task requirements. By testing multiple tasks with varying levels of subtask interdependencies, we examine how the structure of the execution graph influences the selection of appropriate agent types and the overall allocation efficiency.

5.2.1. Allocation Scenario Setup

The experiments are conducted in the FEROX simulator where a heterogeneous team of 2 human agents and 13 drones operates in a shared environment (see Figure 13). Each drone is initialised with a specific battery level and located at different distances from two predefined task targets. This heterogeneous setup is intended to test whether the

proposed GA-based method can correctly match the ground-truth allocation to subtasks by considering both their type (human or drone) and dynamic constraints, such as energy availability and travel distance.



Figure 13. Drones and humans are distributed across the forestry scenario in the FEROX Simulator.

The human agents are assumed to have full energy availability and unrestricted mobility. In contrast, drones vary significantly in both remaining energy and proximity to the two target locations (denoted as T_1 and T_2). These factors are expected to influence the GA's allocation decisions, especially under different task dependency structures and energy cost penalties. Table 3 summarises the battery levels and distances of each drone from the two targets used in the experiments.

Table 3. Battery levels and distances to the targets for drone and human agents.

Robot	Battery (%)	Distance to T_1 (m)	Distance to T_2 (m)
D1	3	106.10	46.22
D2	22	120.34	53.88
D3	16	112.35	111.32
D4	1	48.49	137.11
D5	70	110.44	77.22
D6	30	101.99	165.15
D7	45	102.25	52.82
D8	35	103.39	109.39
D9	24	77.97	179.67
D10	80	70.11	113.82
D11	95	110.50	111.84
D12	75	140.09	80.65
D13	20	126.22	138.95
H1	100	117.12	32.33
H2	100	51.72	97.57

To evaluate how different task dependency structures influence agent selection, we define three task variants with varying subtask interdependencies, shown in Figure 14. All tasks consist of three subtasks and are executed using the same environment setup described above.

- **Task 1: Independent Subtasks.** This task serves as a baseline and is identical to the task used in the LLM experiment. Each subtask is independent and can be executed concurrently without any ordering constraints.

- **Task 2: Interdependent Subtasks.** This variant introduces a single directed dependency between two of the subtasks in Task 1, thereby enforcing a partial execution order. This setup is used to test how even minimal interdependency affects agent allocation, especially under energy and distance constraints.
- **Task 3: Mixed-Agent with Interdependencies.** This task is a variant of Task 3 from the LLM experiment, where the final `MoveTo` action after the `Picking` action has been removed. The resulting structure includes three subtasks with stronger interdependencies, requiring more coordinated execution. Unlike the previous variants, this task explicitly involves heterogeneous agent types: one human and two drones must be assigned, increasing the complexity of the allocation. This setup evaluates the algorithm's ability to correctly handle both dependency-aware scheduling and agent-type suitability within a more constrained and interlinked execution flow.

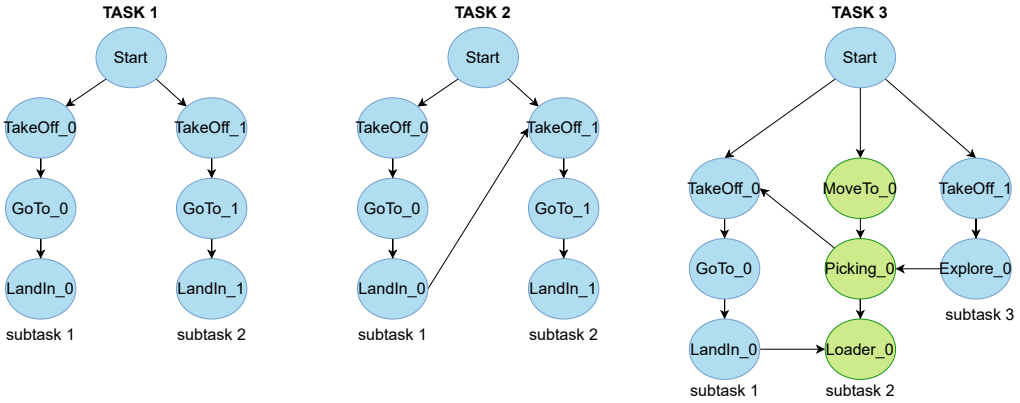


Figure 14. DAG representations of the three task variants.

Each task variant is executed under nine different GA configurations, created by combining the following parameters:

- Population Sizes: 30, 50, and 100.
- Energy Cost Coefficients: 0.05, 0.1, and 0.5.

For each configuration and task, the GA was run 30 times to ensure consistency across stochastic runs. This resulted in a total of 810 runs across all task types and parameter settings. The objective of this setup is to observe how varying population sizes and energy-cost weights influence the quality of agent–task assignments, particularly in the presence of inter-task dependencies and agent heterogeneity.

5.2.2. Allocation Optimality and Accuracy Metrics

The effectiveness of the GA scheduler is assessed using two orthogonal criteria over multiple independent runs of every task–configuration combination.

Assignment Correctness: For each subtask, the ground-truth agent was known *a priori*, manually determined based on the minimal-cost solution considering agent capabilities and distances. After each GA run, we recorded the agent actually selected for that subtask and classified the run as *correct* only if the complete set of selected agents exactly matches the ground truth. The ratio of correct runs to total runs therefore measures the GA's ability to converge to the optimal allocation under stochastic evolution.

Execution Time: The wall-clock time required for the GA to terminate is captured for every run. This metric reflects the computational cost of the optimisation process and is later analysed with respect to GA population size and energy-cost weighting.

After every run, we derived (i) assignment correctness and (ii) the distribution of GA execution times. These metrics enable a comprehensive analysis of the GA's solution

quality and computational efficiency as they vary with task complexity, population size, and energy-cost weighting.

5.2.3. Assessment of Allocation Performance

We evaluated the impact of different GA configurations—specifically population size and energy cost weighting—on agent selection consistency and diversity across three tasks in the FEROX simulator. The agent selection frequencies under varying settings are presented in Figures 15–17.

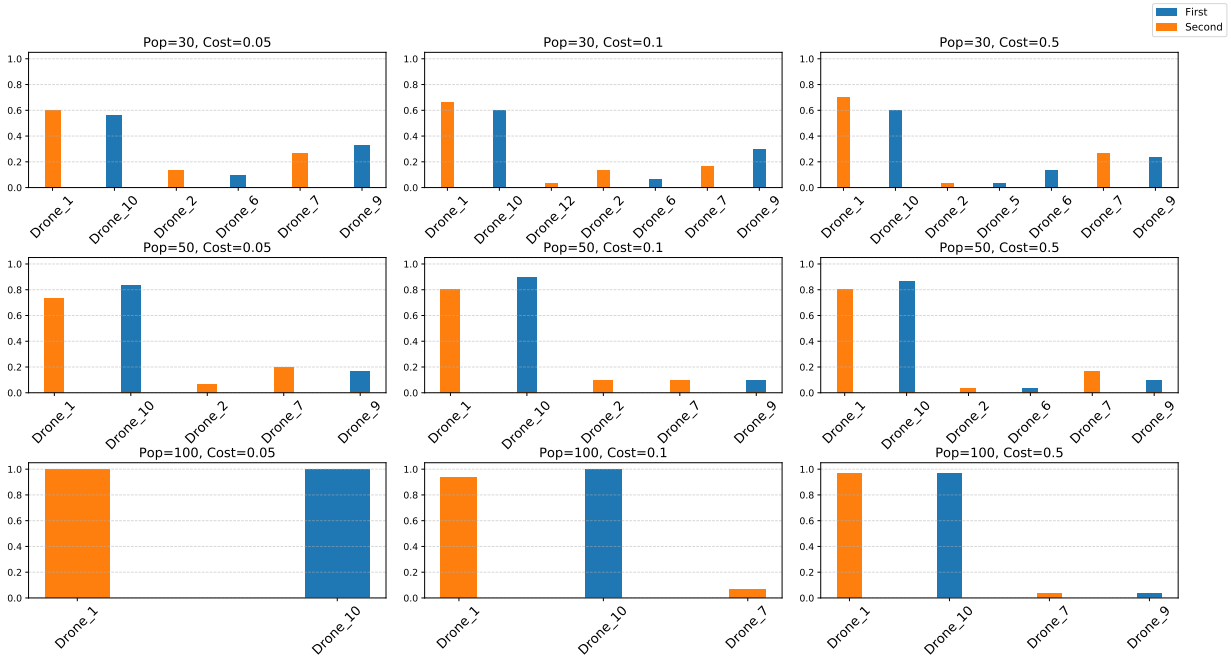


Figure 15. Agent selection for Task 1.

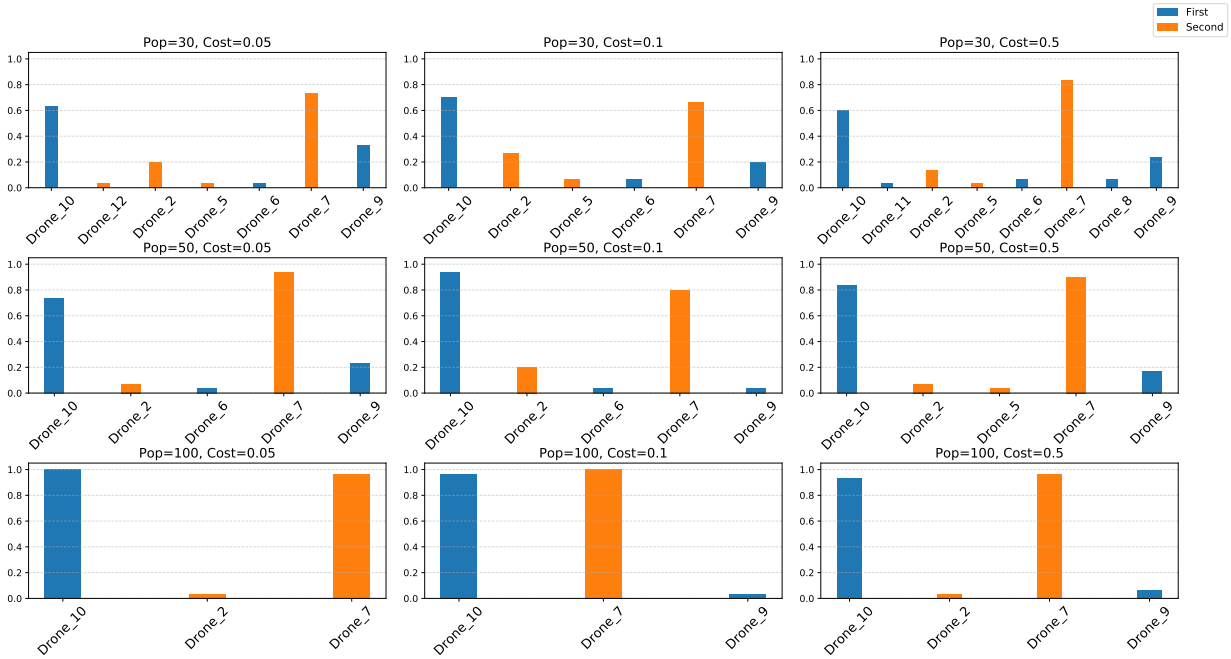


Figure 16. Agent selection for Task 2.

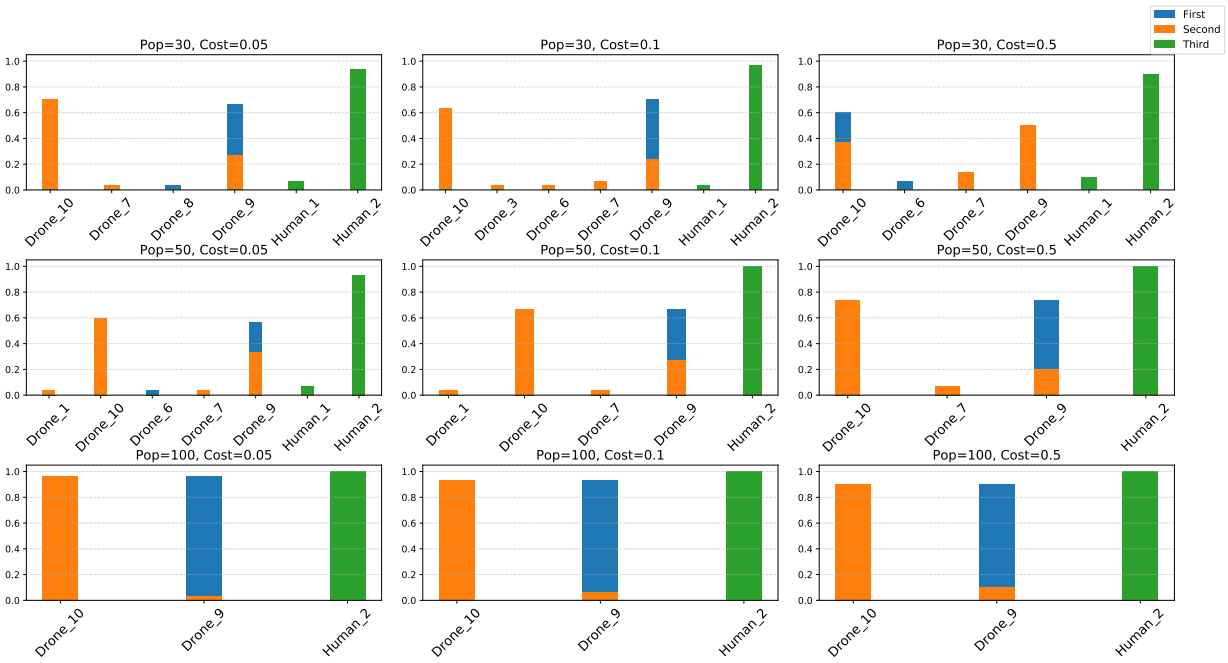


Figure 17. Agent selection for Task 3.

Task 1

Across all configurations, Drone_10 consistently emerged as the most frequently selected first agent. At the smallest population size (30), it was chosen in 56.67% of runs, with some variation from Drone_6 and Drone_9. As the population size increased, convergence strengthened: 83.33% at population size 50 and 100% at a population size of 100 (cost weight = 0.05 and 0.1).

A similar pattern was observed for the second agent, where Drone_1 achieved 100% selection at population size 100 and cost weight 0.05.

Notably, although Drone_4 was physically the closest to the target location, it was never selected due to inadequate battery capacity. This highlights the importance of integrating resource constraints—such as battery level—into the task allocation mechanism, as spatial proximity alone does not guarantee feasibility under real-world conditions.

Task 2

This task introduced a sequential dependency: the second agent could only proceed after the first completed its mission. This constraint made battery sufficiency and wait tolerance critical factors.

In contrast to Task 1, Drone_1 became nonviable due to its low battery level (3%) and the additional wait time. Instead, Drone_10 remained the dominant first agent, while Drone_7 replaced Drone_1 as the preferred second agent, likely due to its higher battery capacity and readiness for delayed execution.

At population size 100 and cost weight 0.1, Drone_10 and Drone_7 were selected in 96.67% and 100% of runs, respectively. This preference persisted even at a higher cost weight (0.5), showing the algorithm's robustness to moderate penalisation.

Task 3

The GA was executed under varying configurations of population size (30, 50, 100) and energy cost weighting (0.05, 0.1, 0.5) to evaluate agent selection frequencies for the two robotic agents assigned in Task 3, as described in Section 5.2.1.

Across all configurations, Drone_9 was predominantly selected as the first robotic agent responsible for the exploration subtask, with selection frequencies ranging from

approximately 56% to 97%. This preference reflects its advantageous position as the second closest drone to the target area, despite a limited battery level (24%). It is important to note that the duration of the exploration task itself was neglected in the temporal dependency modelling; only the drone's arrival time at the target was considered for scheduling purposes.

Conversely, Drone_10 was most frequently chosen as the second robotic agent, tasked with post-picking loading operations. The selection frequency of Drone_10 as the second robot varied between 36% and 97%, consistent with its higher battery capacity (80%) and proximity, enabling it to undertake the longer-duration dependent subtasks that follow exploration and picking.

Regarding human assignment, Human_2 was consistently selected, attributable to its closer proximity to the target area compared to other available human agents. This selection aligns with the goal of minimising movement time and overall task duration.

Discussion

Overall, the proposed approach effectively assigns multiple agents—both humans and robots—based on a combination of key factors, including battery capacity, proximity, and task dependencies. The algorithm demonstrates a strong ability to respect resource constraints, particularly battery levels, ensuring that agents with insufficient energy are excluded even if they are spatially closer to task locations. This multi-agent allocation approach explicitly accounts for agent type (human vs. robot) by assigning subtasks accordingly, reflecting their differing capabilities and availability.

Population size was a significant factor influencing the diversity and stability of agent selection. Smaller populations led to more variability in the chosen agents, while larger populations (especially Pop = 100) resulted in consistent convergence toward the most suitable candidates across runs. In contrast, the weighting of energy cost penalties had a comparatively subtler effect on agent diversity; even as penalisation increased, the algorithm consistently preferred the most viable agents. This indicates a well-shaped fitness landscape that reliably guides the search toward feasible and effective solutions regardless of moderate changes in cost weighting.

Figure 18 presents runtime distributions. Larger populations led to increased runtimes, from a median of 1.4–2.1 s at a population size of 30 to over 3 s at 100. Meanwhile, energy cost weight had minimal impact on execution time. This reflects a predictable trade-off: larger populations improve solution quality at the cost of longer (but still acceptable) computation time.

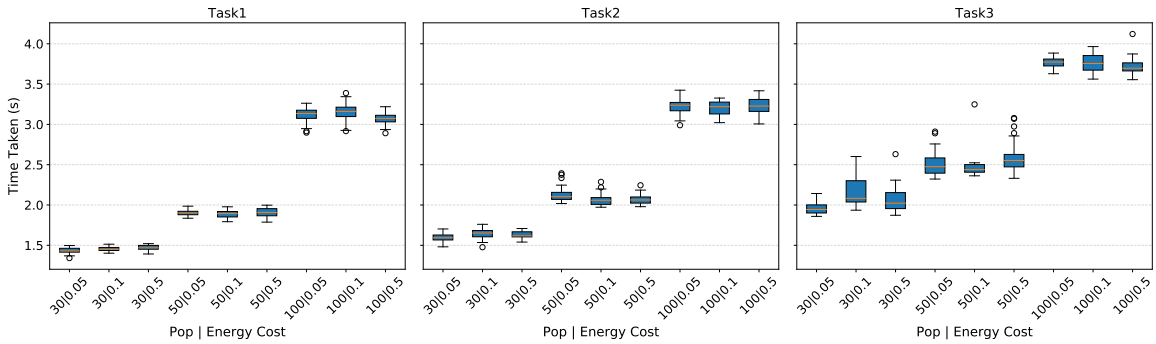


Figure 18. Time Distributions Across GA Configurations. The orange horizontal lines within the boxes represent the median execution time, while the open circles denote outliers.

5.3. Scalability

The scalability test builds upon the previous GA experiments by applying the same setup to Task 1 (independent subtasks) under increasing team sizes of 10, 100, and

1000 robots. This test evaluates the GA's performance and robustness as the number of agents increases significantly.

5.3.1. Scalability Test Setup

All robots are assigned identical battery levels to eliminate energy variation as a confounding factor. Among them, the first 10 robots are strategically positioned: Drone_1 is initialised as the closest robot to Target 1, and Drone_2 as the closest robot to Target 2, allowing us to track whether the GA continues to prioritise optimal selections under increased agent counts. The remaining robots are distributed randomly around the targets to ensure they are not initially close, as shown in Figure 19. This design introduces spatial diversity, mimicking real-world deployment scenarios.

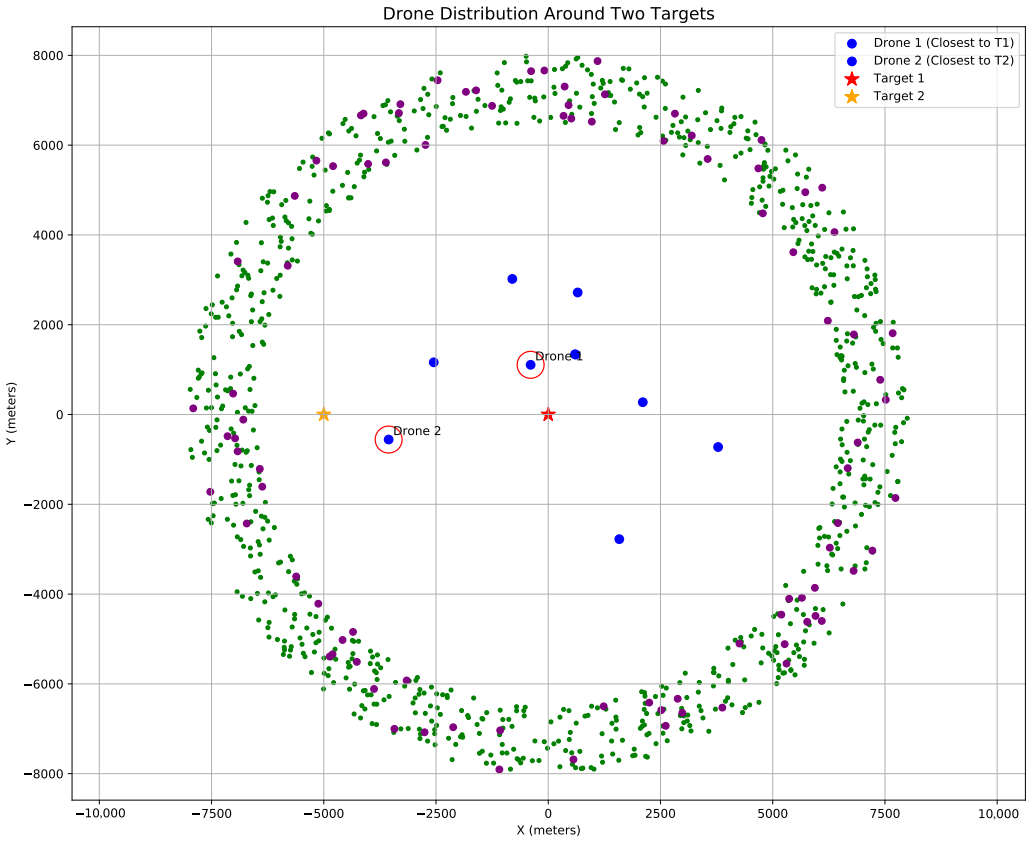


Figure 19. Agent distribution for the scalability experiment. The first 10 agents are shown in blue, additional agents up to 100 are shown in purple, and the remaining agents up to 1000 are shown in green.

Due to computational and real-time constraints, the Unity-based FEROX simulator was not used in this experiment. A distributed ROS-based simulation was used in place of the Unity environment. In this setup, robot states (e.g., battery levels, GNSS positions) are published by a standalone Python script, while a separate subscriber runs the GA algorithm. This design avoids Unity's computational overhead, particularly in the 1000-agent scenarios.

To maintain comparable assignment accuracy as the number of agents increased, we scaled the GA's population size and number of generations. This adjustment enables the GA to effectively navigate the expanded search space and helps assess its scalability in terms of both solution quality and runtime performance.

5.3.2. Scalability Performance Metrics

The evaluation follows the same methodology detailed in Section 5.2.2, concentrating exclusively on Task 1. We assess the GA's performance as the fleet size scales from 10 to 100 and 1000 robots, while increasing the population size and the number of generations to sustain comparable solution quality. Performance metrics include Assignment Correctness and GA execution time.

5.3.3. Assessment of Scalability

Figure 20 summarises the scalability test results, comparing GA execution time and assignment accuracy across 10-, 100-, and 1000-agent configurations. For each team size, the GA was executed 30 times. We report the median execution time and the assignment accuracy. Accuracy is defined as the average percentage of runs in which Drone_1 and Drone_2, the closest robots to their respective targets, were correctly selected for the tasks.

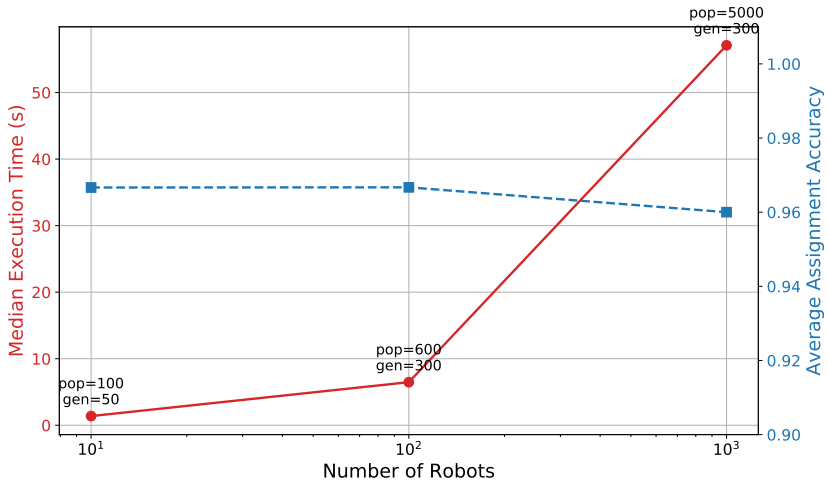


Figure 20. Median execution time (in seconds) and assignment accuracy (percentage of correct agent selection) for Task 1 across increasing fleet sizes. Results averaged over 30 independent GA runs. The solid red line represents the median execution time, while the dashed blue line represents the average assignment accuracy.

The results indicate that the proposed approach maintains a high level of assignment accuracy (95%) even as the number of available robots increases by two orders of magnitude. This demonstrates that the framework remains robust as the size of the agent pool increases.

Importantly, the execution time increases from 1.38 s for 10 agents to 57.09 s for 1000 agents, reflecting the additional computational cost of evaluating a larger candidate pool.

These results provide empirical evidence of the framework's capability to manage large-scale agent populations. To further clarify these findings, we provide the following analysis:

Runtime Complexity: As the number of candidate agents increases, the observed median execution time (57.09 s for 1000 agents) is consistent with the derived complexity of $O(G \cdot P \cdot S \cdot M)$ (Section 4.5.3). This indicates that the optimisation remains tractable and avoids combinatorial explosion.

Memory Considerations: The memory footprint is dominated by intermediate data structures such as the agent–subtask eligibility matrix of size $O(M \times S)$. In practice, since the number of subtasks remains limited, this results in a modest memory requirement that is suitable for standard deployment environments.

Approximation Behaviour: Despite the increase in the candidate agent pool, the assignment accuracy remains stable (95–96%) across all fleet sizes. This suggests that larger

agent pools increase the likelihood of including near-optimal assignments within the search space, enabling the GA to consistently identify high-quality solutions without requiring a proportional increase in optimisation effort.

These findings underscore the GA's robustness in large-scale task allocation, while also highlighting the importance of future work on algorithmic optimisation or parallelisation to support real-time deployment in massive multi-agent systems.

5.4. System-Level Integration and Resilience Assessment

While the previous experiments validated the individual performance of the LLM-based decomposition and the GA-based allocation, this experiment assesses the behaviour of the complete integrated architecture. The goal is to evaluate the unified decision–execution loop during an interactive human–robot mission within the high-fidelity FEROX Simulator.

5.4.1. Experimental Setup and Scenarios

The system-level evaluation exercises task allocation, monitored execution, and recovery in an end-to-end mission. The simulator provides a ground-truth observation proxy for human activities (e.g., walking, picking, loading) to determine action compliance. A collaborative *PickupAndLoad* workflow is implemented, where a human picker harvests berries and transfers the load to a drone for transport to a landing port.

The experiments were conducted using a gamified harvesting mission. Participants controlled an avatar to collect berries under three controlled conditions, with $n = 30$ trials per condition:

1. Human-only Baseline: Manual searching, harvesting, and transport.
2. Drone-assisted (Nominal): Coordinated execution where the human harvests and the drone transports.
3. Drone-assisted with Failure: A mid-transport drone failure was injected after the loading phase to trigger the adaptive recovery pipeline.

The following assumptions were made:

- Human locomotion is load-dependent; velocity decreases as the carried berry count increases.
- All agents, both human and robotic, operate as single-task executors.
- Communication delays between the simulator and ROS middleware are negligible relative to task durations.

Failures are introduced as controlled mechanical faults in the drone-assisted scenario. Specifically, in the failure scenario ($n = 30$), a single mid-execution failure is injected per trial after the loading phase, resulting in a 100% failure rate within this condition.

The Human-only Baseline (manual condition) serves as a non-recovery baseline, where no robotic assistance or adaptive reallocation is available. This allows us to quantify the impact of both task allocation and recovery mechanisms by comparing system performance under (i) no assistance, (ii) nominal coordination, and (iii) failure with recovery.

5.4.2. Performance Metrics

Four primary metrics were used to quantify system-level synergy and responsiveness:

1. Harvesting Throughput (berries/min): Total berries collected divided by total run duration.
2. Idle Ratio (%): The proportion of time the human is not actively picking (includes navigation, waiting, and loading).
3. Agent Assignment Time (s): Latency from the task request to the GA-based agent binding.
4. Replanning Time (s): Latency from failure detection to the injection of a semantically valid recovery plan.

5.4.3. Results

The results of the 90 experimental trials were analysed using a MANOVA, which indicated a statistically significant overall group effect on the combined performance outcomes (Wilks' $\Lambda = 0.252$, $F(2, 86) = 127.35$, $p < 0.001$). Follow-up univariate ANOVAs confirmed strong effects on both throughput ($F(2, 87) = 129.13$, $p < 0.001$) and idle ratio ($F(2, 87) = 123.86$, $p < 0.001$).

Impact of Assistance on Productivity: As summarised in Table 4, the Drone-assisted (Nominal) condition showed a substantial advantage. Throughput increased from 337.06 ± 43.79 berries/min in the manual baseline to 461.98 ± 16.71 berries/min with robotic support—a 37% improvement. This difference is statistically significant ($F(2, 87) = 129.13$, $p < 0.001$). This gain is directly attributable to the reduction in human idle time, as the idle ratio dropped from 49% to 30%.

Table 4. Descriptive statistics for system-level performance across experimental conditions ($n = 30$ per condition). Idle ratio is reported as a proportion of time not spent picking (multiply by 100 to obtain percentages).

Condition	*n*	Berries/Min (Mean ± SD)	Idle Ratio (Median [Q1, Q3])
Manual	30	337.06 ± 43.79	0.49 [0.45, 0.55]
Drone (No Fail)	30	461.98 ± 16.71	0.30 [0.29, 0.32]
Drone (Failure)	30	415.48 ± 24.12	0.38 [0.36, 0.41]

Resilience under Failure: In the failure scenario, the system maintained mission continuity despite the transport failure. Although the idle ratio increased to 38% due to the necessary repair interaction, the throughput (415.48 ± 24.12 berries/min) remained significantly higher than the manual baseline.

System Responsiveness: Table 5 highlights the timing overheads. The Replanning Time (24.07 ± 5.58 s) was significantly faster than the initial Agent Assignment Time (38.08 ± 9.44 s), confirming the architecture's ability to restore feasibility within interactive timeframes.

Table 5. System responsiveness metrics: Agent Assignment Time ($N = 60$) and Replanning Time ($N = 30$).

Metric	Mean (s)	Std (s)
Agent Assignment Time	38.08	9.44
Replanning Time	24.07	5.58

5.4.4. Discussion

A central finding is that productivity gains are driven by explicit coordination semantics rather than mere robotic presence. The DAG-based runtime enforces human–robot couplings (e.g., rendezvous and payload exchange) as executable constraints, eliminating non-productive human intervals and allowing the operator to focus on harvesting. Consequently, the observed 37% increase in throughput stems from structured synchronisation and improved allocation of human effort.

The effectiveness of the architecture is closely tied to its responsiveness. The observed latencies (38 s for initial assignment and 24 s for recovery) are well-suited for logistics-heavy field robotics, where task durations dominate planning overhead. However, these latencies may limit applicability in fast-paced collaborative manipulation scenarios. This highlights a trade-off: the current reasoning complexity is appropriate for long-horizon missions, while tighter real-time domains would require faster optimisation backends or approximate planning strategies.

The failure experiments serve as a stress test for intent preservation. By framing recovery as a reconciliation with the current DAG state rather than a full restart, the system avoids discarding the physical and cognitive effort already invested by both agents. This design supports mission continuity and maintains human–robot collaboration efficiency under mid-execution disturbances.

Despite the positive results, several experimental limitations remain. First, the use of simulator-provided labels enables controlled evaluation but does not capture real-world sensing noise and perception uncertainty. Second, the evaluation focuses on a specific class of failures (transport-related mechanical faults) and does not yet consider richer fault models or environmental variability, such as obstacles, communication delays, or sensor inconsistencies, which would require more detailed state representations. In addition, a direct comparison with a failure scenario without recovery is not included in this study. In such cases, a failure would result in task interruption and require full task reinitialisation. Future work will investigate controlled comparisons with explicit no-recovery baselines.

Beyond experimental constraints, the integrated system is subject to several structural limitations. The framework assumes that the initial DAG generated by the LLM is semantically valid; missing dependencies may lead to execution inconsistencies or deadlocks if recovery cannot identify a feasible continuation point. Moreover, although replanning ($\approx$24 s) is faster than initial assignment, it still introduces a non-negligible pause. In highly dynamic environments, this latency may lead to brittleness if the system state evolves faster than the optimisation process. Finally, the system is currently validated on explicit mechanical failures and human-mediated repairs, while its robustness against more subtle failure modes—such as sensor drift or intermittent communication loss—remains to be evaluated.

Although the framework is evaluated in an agroforestry scenario using the FEROX simulator, its architecture is designed to be domain-independent. This is enabled by a standardised interface for agent and action integration, as introduced in our prior work [36]. Agents are abstracted through a unified namespace structure (e.g., `agent_type/agent_name/status`), while actions are exposed via a consistent interface (e.g., `agent_type/agent_name/action/action_name`). This abstraction decouples task definitions from domain-specific implementations, allowing the framework to be extended to domains such as construction, logistics, or inspection by exposing domain-specific actions without modifying the core planning or allocation pipeline.

Human–robot collaboration also introduces important safety and ethical considerations. Task allocation must ensure that human operators are not assigned unsafe or cognitively excessive workloads, while robot behaviour must remain predictable, interpretable, and interruptible. In the current system, conservative execution policies are adopted, such as avoiding interruption of ongoing human actions and preserving synchronisation constraints defined in the task DAG. However, the framework does not yet provide formal safety guarantees, and incorporating human-aware planning, safety constraints, and compliance with domain-specific standards remains an important direction for future work.

5.4.5. Comparative Evaluation and Dependency Analysis

To evaluate the role of dependency-aware modelling and the choice of optimisation strategy, we conducted a comparative benchmark across four methods: GA, PSO, Greedy, and SSI.

We consider two task configurations: (i) sequential subtasks with explicit dependencies (Task 2 in Section 5.2), and (ii) independent parallel subtasks (Task 1 in Section 5.2) with 15 LWDs. In both configurations, Drone_11 and Drone_12 are the closest agents to the task targets, making them locally optimal candidates based on travel time. However, Drone_12 is subject to a battery constraint that limits its ability to complete all assigned actions.

This setup creates a controlled trade-off between local optimality and global feasibility. While distance-based strategies favour Drone_12, dependency-aware allocation must consider whether the agent can complete the full task sequence without violating resource constraints. This distinction becomes critical in the dependency-constrained scenario, where suboptimal early assignments propagate through the task sequence and significantly affect overall makespan.

Following this setup, we compare four allocation strategies: (i) a Genetic Algorithm as the proposed method, (ii) Particle Swarm Optimisation (PSO) as an alternative metaheuristic baseline, (iii) a greedy allocation strategy, and (iv) a Sequential Single-Item (SSI) auction-based method.

Each method is evaluated over 30 independent runs for both task configurations. We report the average makespan and computation time for each method in Table 6.

Table 6. Performance Comparison of Task Allocation Methods under Different Dependency Structures and Resource Constraints (Averaged over 30 trials).

Scenario	Method	Makespan (s)	Comp. Time (s)	Optimal Conv. Rate (%)
Task 1 (Independent)	GA (Ours)	11.68	3.82	83%
	PSO	12.18	2.22	87%
	Greedy	11.35	2.29	100%
	SSI	11.35	2.23	100%
Task 2 (Dependency-Constrained)	GA (Ours)	53.67	3.59	96%
	PSO	53.94	2.32	90%
	Greedy	3077.44 †	2.42	0%
	SSI	3077.44 †	2.35	0%

† Values represent penalty-inflated makespan caused by mission failure (battery exhaustion of Drone_12).

In the dependency-constrained scenario (Task 2), both greedy and SSI consistently select Drone_12 and Drone_11 based solely on proximity. However, Drone_12 does not have sufficient battery capacity to complete the full task sequence under dependency constraints. As a result, these allocations incur large penalties in the fitness evaluation, leading to significantly inflated makespan values ≈3077 s. This demonstrates that local decision-making strategies fail to account for feasibility under resource and temporal dependencies.

In contrast, both GA and PSO correctly avoid infeasible assignments by incorporating dependency-aware evaluation. The GA demonstrates high reliability by converging to the optimal allocation (Drone_11 and Drone_2) in 29 out of 30 independent runs (96.7% consistency). In the single instance where it deviated, the algorithm settled on a near-optimal feasible alternative, reflecting the inherent stochastic nature of evolutionary search while maintaining operational safety. PSO achieves similar near-optimal performance but occasionally selects Drone_7 as the second agent, which represents the next-best feasible alternative. This behaviour reflects the stochastic nature of PSO, leading to slightly lower consistency compared to GA.

In the dependency-free scenario (Task 1), the absence of ordering constraints allows Drone_12 to complete its assigned subtask within its battery limits. Since the total mission duration for each agent is reduced compared to the cumulative sequential load, Drone_12's limited battery becomes sufficient for its specific subtask, rendering the locally optimal choice feasible. As a result, multiple agent combinations, including Drone_12, Drone_2, and Drone_7, become feasible. Consequently, both GA and PSO exhibit reduced consistency, as the fitness landscape contains multiple near-optimal solutions. In this setting, greedy and SSI also perform competitively, as local distance-based decisions are sufficient when global dependencies are absent.

In terms of computational performance, PSO consistently achieves lower execution times compared to GA, due to its simpler update mechanism and faster convergence per iteration. However, this speed advantage comes at the cost of lower solution stability in dependency-constrained scenarios.

Overall, these results highlight that the primary performance gain stems from dependency-aware reasoning rather than the choice of optimiser alone. Without modelling dependencies, different allocation paradigms (greedy and auction-based) converge to identical but infeasible solutions. Conversely, when dependencies are properly captured, both GA and PSO are able to produce feasible allocations, with GA providing more reliable convergence and PSO offering faster computation.

6. Conclusions

This study introduced a cognitive architecture that integrates LLMs with a graph-based genetic algorithm to address task allocation and resilient execution in heterogeneous multi-agent systems. Through comprehensive experiments, we demonstrated that subtask-based prompting consistently produces accurate and interpretable execution DAGs, providing a modular foundation for real-world robotics workflows. In the allocation stage, the GA-based allocator effectively balanced makespan minimisation with strict resource constraints, sustaining over 95% assignment accuracy in large-scale scenarios with up to 1000 agents.

Beyond algorithmic performance, system-level assessments in a high-fidelity simulator validated three key operational findings:

1. Human–Robot Synergy: The architecture successfully optimised joint workflows, resulting in a 37% increase in harvesting throughput and a 19% reduction in human idle time compared to manual baselines.
2. Computational Resilience: The integrated loop demonstrated the ability to detect and recover from mid-execution failures. By using a semantic recovery mechanism to repair the structured execution context, the system maintained mission continuity with an average replanning time of 24 s, avoiding the need for full mission restarts.
3. Operational Trade-offs: While the architecture introduces an initial latency of approximately 38 s for complex reasoning, the sustained efficiency gains and the ability to preserve mission intent under uncertainty rapidly offset this initial overhead.

Overall, these results provide strong evidence that coupling structured task reasoning with adaptive allocation and monitoring creates a robust framework for multi-agent coordination. Future work will focus on further optimising runtime through parallel evaluation and hybrid search strategies to enable responsive performance in tighter real-time collaboration domains.

Author Contributions: Conceptualisation, B.Y. and M.S.C.; methodology, B.Y. and M.S.C.; software, B.Y.; validation, B.Y.; formal analysis, B.Y.; investigation, B.Y. and M.S.C.; data curation, B.Y.; writing—original draft preparation, B.Y.; writing—review and editing, M.S.C.; visualisation, B.Y.; supervision, M.S.C., S.S. and A.V.; project administration, B.Y.; funding acquisition, B.Y., S.S. and A.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partly funded by the FEROX EU project, which has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101070440. This work was also partially funded by FCT—Fundação para a Ciência e a Tecnologia (FCT) I.P. (2022.13815.BDANA), through national funds, within the scope of the UIDB/00127/2020 project (IEETA/UA).

Institutional Review Board Statement: Formal ethics approval was not required for this study as the research was classified as a non-interventional study involving low-risk human interaction with a simulated environment. Under Portuguese national law, mandatory ethics committee approval

applies to clinical research governed by Law No. 21/2014 of 16 April (as amended) and, for clinical trials of medicinal products, Law No. 9/2026 of 6 March. This study did not fall under the scope of these regulations as it did not involve clinical trials, invasive procedures, or the collection of sensitive biometric data. Participants interacted solely with a gamified simulation, and the analysis was restricted to anonymized avatar movement data (spatial coordinates and timestamps). Data protection and privacy were maintained in strict accordance with Regulation (EU) 2016/679 (GDPR) and Portuguese Law No. 58/2019 of 8 August.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy restrictions related to the FEROX project's internal data policies.

Acknowledgments: The authors would like to thank the FEROX project consortium for their support. During the preparation of this manuscript, the authors used ChatGPT (version 5.4) for the purposes of language editing and structural refinement of the text. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: Authors Beril Yalcinkaya and Micael S. Couceiro were employed by the company Ingeniarius Lda. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Appendix A. Representative Prompt Template for LLM-Based Task Decomposition

The following prompt template was used to guide the LLM in converting Python task functions into a structured JSON representation of execution flow.

```
You are a code analyzer. Your task is to convert the given Python function
into a structured JSON format representing its execution flow. The JSON
should capture all function calls, conditions, synchronization points, and
execution sequences.

Output format:
{
  "function_name": {
    "steps": [
      {
        "action": "Action keyword",
        "call": "Function/Method Name",
        "target": "Targeted Function, Event, or Thread",
        "result": "Result Variable",
        "condition": "Condition (if any)"
      }
    ]
  }
}

Allowed actions:
- "Executing Action": used when calling a task/action function
- "Wait": used when waiting on an event or synchronization point
- "Set": used when setting an event signal
- "Threading": used when starting a new thread
- "Hold": used when explicitly waiting in polling loops

Instructions:
1. Preserve the execution order exactly.
2. Explicitly represent synchronization primitives.
3. Capture conditional logic when present.
4. Do not invent implicit actions.
```

```
5. Output only the JSON structure.

Examples:
[Example 1: sequential execution]
[Example 2: synchronization with set()]
[Example 3: waiting and conditional execution]

Task:
Now, analyze the following Python function and generate the corresponding
JSON output. Ensure that the output follows the schema exactly.
```

Appendix B. Representative Prompt Template for LLM-Based Recovery Planning

The following prompt template summarizes the structured prompting strategy used for adaptive recovery generation after execution failures.

System Prompt.

```
You are an adaptive robotic task planner. Your task is to analyze a
multi-agent task failure and generate a Python-based recovery plan.

You must reason over:
- the current task code,
- the failure message,
- agent states, capabilities, and positions,
- the initial and current task DAGs,
- inter-agent dependencies and synchronization events.

Your recovery plan may include:
- revised agent-specific recovery functions,
- stop or interrupt decisions for ongoing agents,
- delegation to other available agents,
- new synchronization primitives when required.

Constraints:
1. Use only the actions listed in each agent's capabilities.
2. Do not invent new agents, capabilities, or actions.
3. Keep each action inside the correct agent's subtask.
4. Continue from the failure point; do not repeat completed actions unless
    necessary.
5. Preserve or correctly update synchronization events.
6. If an interrupted agent is reused in recovery, ensure its remaining
   current_dag actions are completed afterward.
7. Return a valid JSON object only.

Required JSON fields:
- task_code
- failed_agents_to_stop
- other_agents_to_stop
- stopped_agents_with_interrupted_action
- agents_to_add
- new_sync_events
- agents_in_recovery
```

User Prompt.

```
Here is the input JSON describing the current task code, failure, agent
states, capabilities, positions, initial_dag, current_dag, and dependencies.

Before generating recovery code, reason about:
1. Whether the failure is local or requires multi-agent coordination.
2. Whether the strategy should retry, revise, or reassign.
3. Which agents should be stopped, interrupted, or added.
4. Which completed actions must not be repeated.
5. Which synchronization events must be preserved or added.

Then return the recovery plan as a valid JSON object following the
specified schema.
```

Representative Input Example.

```
{
  "task_code": "def example(): ...",
  "failure": "LoadPallets: The robot is stuck because the target itself is
      inside an obstacle.",
  "agents": [
    {
      "name": "loader",
      "type": "ugv1",
      "status": "Occupied",
      "progress": {"LoadPallets": 20},
      "position": [0, 0, 10],
      "capabilities": ["GoTo", "LoadPallets", "UnloadPallets"],
      "initial_dag": [["GoTo", "LoadPallets"], ["LoadPallets", "
          UnloadPallets"]],
      "current_dag": [["LoadPallets", "UnloadPallets"]]
    },
    {
      "name": "User_1",
      "type": "human",
      "status": "Idle",
      "progress": {},
      "position": [0, 0, 15],
      "capabilities": ["HumanIteraction"],
      "initial_dag": [],
      "current_dag": []
    }
  ],
  "dependencies": []
}
```

Representative Output Example.

```
{
  "task_code": "def loader_recovery(): ... \n def User_1_recovery(): ...",
  "failed_agents_to_stop": ["loader"],
  "other_agents_to_stop": [],
  "stopped_agents_with_interrupted_action": [],
  "agents_to_add": ["User_1"],
  "new_sync_events": ["thread_human"],
  "agents_in_recovery": ["loader", "User_1"]
}
```

References

1. Skaltsis, G.M.; Shin, H.S.; Tsourdos, A. A survey of task allocation techniques in MAS. In Proceedings of the 2021 International Conference on Unmanned Aircraft Systems (ICUAS), Athens, Greece, 15–18 June 2021; pp. 488–497. [CrossRef]
2. Chen, Z.; Alonso-Mora, J.; Bai, X.; Harabor, D.D.; Stuckey, P.J. Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. *IEEE Robot. Autom. Lett.* **2021**, *6*, 5816–5823. [CrossRef]
3. Queralta, J.P.; Taipalmaa, J.; Pullinen, B.C.; Sarker, V.K.; Gia, T.N.; Tenhunen, H.; Gabbouj, M.; Raitoharju, J.; Westerlund, T. Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision. *IEEE Access* **2020**, *8*, 191617–191643. [CrossRef]
4. Lytridis, C.; Kaburlasos, V.G.; Pachidis, T.; Manios, M.; Vrochidou, E.; Kalampokas, T.; Chatzistamatis, S. An Overview of Cooperative Robotics in Agriculture. *Agronomy* **2021**, *11*, 1818. [CrossRef]
5. Kannan, S.S.; Venkatesh, V.L.N.; Min, B.C. SMART-LLM: Smart Multi-Agent Robot Task Planning using Large Language Models. In Proceedings of the 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Abu Dhabi, United Arab Emirates, 14–18 October 2024; pp. 12140–12147. [CrossRef]
6. Bischoff, E.; Meyer, F.; Inga, J.; Hohmann, S. Multi-Robot Task Allocation and Scheduling Considering Cooperative Tasks and Precedence Constraints. In Proceedings of the 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Toronto, ON, Canada, 11–14 October 2020; pp. 3949–3956. [CrossRef]
7. Nilsson, A.; Bonander, C.; Strömberg, U.; Björk, J. A directed acyclic graph for interactions. *Int. J. Epidemiol.* **2021**, *50*, 613–619. [CrossRef] [PubMed]
8. Chakraa, H.; Guérin, F.; Leclercq, E.; Lefebvre, D. Optimization techniques for Multi-Robot Task Allocation problems: Review on the state-of-the-art. *Robot. Auton. Syst.* **2023**, *168*, 104492. [CrossRef]
9. Yuan, Q.; Kazemi, M.; Xu, X.; Noble, I.; Imbrasaite, V.; Ramachandran, D. TaskLAMA: Probing the complex task understanding of language models. In *AAAI'24: AAAI Conference on Artificial Intelligence*; AAAI Press: Washington, DC, USA, 2024; Volume 38, pp. 19468–19476.
10. Gao, Z.; Mu, Y.; Qu, J.; Hu, M.; Guo, L.; Luo, P.; Lu, Y. Dag-plan: Generating directed acyclic dependency graphs for dual-arm cooperative planning. *arXiv* **2024**, arXiv:2406.09953.
11. Gerkey, B.P.; Matarić, M.J. A formal analysis and taxonomy of task allocation in multi-robot systems. *Int. J. Robot. Res.* **2004**, *23*, 939–954. [CrossRef]
12. Liu, C.; Kroll, A. A centralized multi-robot task allocation for industrial plant inspection by using a* and genetic algorithms. In *Artificial Intelligence and Soft Computing*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 466–474.
13. Giordani, S.; Lujak, M.; Martinelli, F. A distributed algorithm for the multi-robot task allocation problem. In *Trends in Applied Intelligent Systems*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 721–730.
14. Choi, H.L.; Brunet, L.; How, J.P. Consensus-Based Decentralized Auctions for Robust Task Allocation. *IEEE Trans. Robot.* **2009**, *25*, 912–926. [CrossRef]
15. Dias, M.B. *Traderbots: A New Paradigm for Robust and Efficient Multirobot Coordination in Dynamic Environments*; Carnegie Mellon University: Pittsburgh, PA, USA, 2004.
16. Botelho, S.; Alami, R. M+: A scheme for multi-robot cooperation through negotiated task allocation and achievement. In Proceedings of the 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), Detroit, MI, USA, 10–15 May 1999; Volume 2, pp. 1234–1239. [CrossRef]
17. Viguria, A.; Maza, I.; Ollero, A. S+T: An algorithm for distributed multirobot task allocation based on services for improving robot cooperation. In Proceedings of the 2008 IEEE International Conference on Robotics and Automation, Pasadena, CA, USA, 19–23 May 2008; pp. 3163–3168. [CrossRef]
18. Cai, K. Warehouse automation by logistic robotic networks: A cyber-physical control approach. *Front. Inf. Technol. Electron. Eng.* **2020**, *21*, 693–704. [CrossRef]
19. Martin, J.G.; Frejo, J.R.D.; García, R.A.; Camacho, E.F. Multi-robot task allocation problem with multiple nonlinear criteria using branch and bound and genetic algorithms. *Intell. Serv. Robot.* **2021**, *14*, 707–727. [CrossRef]
20. Ghassemi, P.; Chowdhury, S. Multi-robot task allocation in disaster response: Addressing dynamic tasks with deadlines and robots with range and payload constraints. *Robot. Auton. Syst.* **2022**, *147*, 103905. [CrossRef]
21. Li, Z.; Li, X. Genetic algorithm for task allocation and path planning of multi-robot system. *J. Math. Sci.* **2016**, *4*, 34–38.
22. Chen, J.; Wang, J.; Xiao, Q.; Chen, C. A multi-robot task allocation method based on multi-objective optimization. In Proceedings of the 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV), Singapore, 18–21 November 2018; pp. 1868–1873.
23. Wang, J.; Gu, Y.; Li, X. Multi-robot task allocation based on ant colony algorithm. *J. Comput.* **2012**, *7*, 2160–2167. [CrossRef]
24. Korsah, G.A.; Stentz, A.; Dias, M.B. A comprehensive taxonomy for multi-robot task allocation. *Int. J. Robot. Res.* **2013**, *32*, 1495–1512. [CrossRef]

25. Zhou, S.; Zhang, L.; Yang, Y.; Lyu, Q.; Yin, P.; Callison-Burch, C.; Neubig, G. Show me more details: Discovering hierarchies of procedures from semi-structured web data. *arXiv* **2022**, arXiv:2203.07264. [CrossRef]
26. Lin, F.; La Malfa, E.; Hofmann, V.; Yang, E.M.; Cohn, A.; Pierrehumbert, J.B. Graph-enhanced large language models in asynchronous plan reasoning. *arXiv* **2024**, arXiv:2402.02805. [CrossRef]
27. Cao, Y.; Lee, C. Robot behavior-tree-based task generation with large language models. *arXiv* **2023**, arXiv:2302.12927.
28. Obata, K.; Aoki, T.; Horii, T.; Taniguchi, T.; Nagai, T. LiP-LLM: Integrating Linear Programming and dependency graph with Large Language Models for multi-robot task planning. *IEEE Robot. Autom. Lett.* **2024**, *10*, 1122–1129. [CrossRef]
29. Li, M.; Wang, Z.; Li, K.; Liao, X.; Hone, K.; Liu, X. Task allocation on layered multiagent systems: When evolutionary many-objective optimization meets deep Q-learning. *IEEE Trans. Evol. Comput.* **2021**, *25*, 842–855. [CrossRef]
30. Zhao, N.; Liang, Y.C.; Niyato, D.; Pei, Y.; Wu, M.; Jiang, Y. Deep reinforcement learning for user association and resource allocation in heterogeneous cellular networks. *IEEE Trans. Wirel. Commun.* **2019**, *18*, 5141–5152. [CrossRef]
31. Zhang, H.; Zhao, H.; Fu, Y.; Ma, J.; Xiang, Y. The class labels and spatial information based fault diagnosis of air handling unit via combining kernel Fischer discriminant analysis with an improved graph convolutional neural network. *Measurement* **2026**, *257*, 118622. [CrossRef]
32. Technologies of Vision (TeV) @ FBK. WildBe (Revision 010f5f0). 2024. Available online: https://huggingface.co/datasets/FBK-TeV/WildBe (accessed on 12 April 2026).
33. Yalcinkaya, B.; Couceiro, M.S.; Pina, L.; Soares, S.; Valente, A.; Remondino, F. Towards Enhanced Human Activity Recognition for Real-World Human-Robot Collaboration. In Proceedings of the 2024 IEEE International Conference on Robotics and Automation (ICRA), Yokohama, Japan, 13–17 May 2024.
34. Fletcher, S.; Oostveen, A.; Chippendale, P.; Couceiro, M.; Turtiainen, M.; Ballester, L. Developing Unmanned Aerial Robotics to Support Wild Berry Harvesting in Finland: Human Factors, Standards And Ethics. In Proceedings of the ICRES 2023, Utrecht, The Netherlands. 17–18 July 2023; pp. 109–119.
35. Yalçinkaya, B.; Couceiro, M.; Soares, S.F.S.P.; Valente, A. Unlocking the Potential of Human-Robot Synergy under Advanced Industrial Applications: The FEROX Simulator. In *European Robotics Forum 2024*; Springer: Cham, Switzerland, 2024.
36. Yalcinkaya, B.; Couceiro, M.S.; Soares, S.; Valente, A. FORMIGA: A fleet management framework for sustainable human–robot collaboration in field robotics. *Front. Robot. AI* **2025**, *12*, 1706910. [CrossRef] [PubMed]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.