FORMIGA: Framework for Optimised Robotic Management, Integration, Guidance, and Automation

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

[1*]CORE R&D Department, Ingeniarius, Lda., Rua Nossa Senhora da Conceição, 146, Alfena, 4445-147, Portugal.
[2]School of Sciences and Technology-Engineering Department, University of Trás-os-Montes and Alto Douro (UTAD), Vila Real, 4200-465, Portugal.
[3]3D Optical Metrology (3DOM), Bruno Kessler Foundation (FBK), Trento , Italy.

*Corresponding author(s). E-mail(s): beril@ingeniarius.pt;
Contributing authors: micael@ingeniarius.pt; salblues@utad.pt; avalente@utad.pt; remondino@fbk.eu;
†These authors contributed equally to this work.

Abstract

This paper present FORMIGA as an innovative solution for enhancing human-mediated robot autonomy under challenging field applications. FORMIGA emphasises operational management, seamless integration, and sophisticated automation. This paper surveys the state of the art on fleet management systems, including commercially available solutions, so as to establish a baseline around the topic. Afterwards, the paper delves into FORMIGA's features and its pivotal role within the FEROX project - an European project aimed at revolutionising wild berry picking using robots to improve working conditions and operational efficiency. Through simulations, we assess FORMIGA's impact, demonstrating its capability to adapt to varying situations and its potential to generate collaborative tasks using a large language model. Our findings underscore FORMIGA's contribution to the future of human-centric robotic services, marking a significant step towards realising flexible human-robot ecosystems in challenging domains.

Keywords: fleet management system, human-mediated robot autonomy, task coding, human-centric interface design, field robotics

1 Introduction

In an era where the fusion of technology and human expertise is paramount, the realm of robotics stands on the brink of a transformative leap, particularly within field domains, such as agriculture, forestry, construction, and others. The growing challenges faced by these domains, notably the imperative to enhance productivity and sustainability amidst constraints, such as labour shortages and environmental concerns, underscore the urgent need for innovative solutions. The advent of field robotics, characterised by intelligent machines capable of operating around the clock within self-coordinating fleets, heralds a new dawn [1]. These robots, equipped with advanced sensing, autonomy, and learning capabilities, promise not only to revolutionise the efficiency and cost-effectiveness of operations, but also to improve working conditions of humans, fostering a synergy between human workers and robotic counterparts.

The concept of human-robot teaming brings forth a nuanced perspective on the interaction between humans and robots, transcending the traditional view of robots as tools for routine tasks [2]. As we anticipate a future where robots assume more complex roles, the necessity for enhanced human-robot collaboration becomes evident [3]. However, the journey towards achieving seamless integration and autonomy in collaborative robotics is fraught with challenges, including the need to address autonomy exceptions that can disrupt operations, such as localisation, navigation or even electromechanical failures. The collaborative efforts of researchers, roboticists, and end-users are crucial in overcoming these hurdles, driving forward the design and deployment of robotic fleets that are not just technologically advanced, but also attuned to the nuanced requirements of human-robot collaboration in real-world settings [4].

Building on the foundational insights into the challenges and aspirations guiding the development of fleet management systems (FMS) within the robotics domain, we introduce a **F**ramework for **O**ptimised **R**obotic **M**anagement, **I**ntegration, **G**uidance, and **A**utomation, or FORMIGA for short[1]. FORMIGA is designed to navigate the complexities of real-world environments. FORMIGA's strategic inception and deployment within challenging field sectors epitomises the system's adaptability and potential. By offering a suite of features focused on optimised operational management, seamless integration, and intuitive user guidance, FORMIGA redefines the interface between human labour and robotic assistance.

2 Literature Review

The synthesis of human intellect and robotic capability is increasingly essential for the creation and delivery of effective services, especially under challenging domains. This underscores the indispensability of human-mediated systems in real-world environments, where robotics platforms are not merely autonomous entities, but integral components of a larger collaborative ecosystem [5]. The driving force behind this integration is the strategic amalgamation of human expertise with the computational prowess of robots, aimed at achieving tasks that neither could accomplish

[1] FORMIGA, meaning "ant" in Portuguese, embodies the system's qualities of strength, cooperation, and efficiency, mirroring the collaborative and resilient nature of ants in complex environments.

alone with the same efficiency, safety, and ecological sensitivity. Such a collaboration necessitates systems where robots perform under human supervision or intervention, bridging the gap between autonomous operation and human oversight [6]. This delicate balance of shared control and autonomy is crucial for tailoring robotic behaviour to meet domain-specific objectives, ensuring safety and acceptability among users. Next sections venture into the exploration of FMS within this context, identifying existing solutions and clarifying the challenges of designing and implementing these human-mediated robotic services for field applications.

2.1 Fleet Management Systems (FMS)

The literature indicates an urgent need for innovative approaches that not only facilitate effective communication and knowledge sharing among stakeholders, including roboticists, domain experts, and end-users, but also harmonise the conflicting requirements emerging from such interdisciplinary collaboration [7]. FMS constitute a pivotal facet, signifying the orchestrated governance of robotic entities collaboratively engaged in various tasks with humans. This orchestration encompasses intricate planning, allocation, and synchronisation, aimed at ensuring the seamless operation of agents (i.e., humans and robots) in diverse contexts, namely within field applications in which this work falls in. FMS serve as a linchpin for enhancing productivity, curtailing operational expenditures, and elevating mission success probabilities.

The key features of FMS for field robotics are (i) Task Formulation; (ii) Task Allocation; (iii) Coordination; (iv) Optimisation; (v) Adaptability; and (vi) Resilience. The state of the art of each of these key features are further explored next.

2.1.1 Task Formulation

Task formulation in robotics has traditionally relied on human expertise to define the necessary actions for robots [8]. This approach becomes increasingly complex in multi-robot and human-robot collaborative settings, where it requires accounting for both individual robot capabilities and the collective potential for joint task achievement [9]. The complexity further escalates when considering human skills, adaptability, and real-time decision-making in dynamic field environments.

Numerous architectures have been proposed to simplify this process, especially in human-robot assembly systems. Multi-criteria planning methods have been used to handle task allocation, incorporating factors such as ergonomics and productivity [9]. However, these systems often focus on controlled environments, limiting their applicability to real-world, dynamic operations. Recent advancements with AI, particularly Large Language Models (LLMs), have explored generating tasks in various formats, including flowcharts, behaviour trees, and programming language code [10–12]. LLMs leverage semantic knowledge and real-time perceptions, yet they primarily target structured environments like factories.

For instance, GenSim2 leverages multi-modal LLMs to create complex, scalable simulation tasks, enabling zero-shot sim-to-real transfer [11]. Other works, such as using LLMs to generate behaviour trees (BTs) for robot planning [12], improve task generation through iterative and human-in-the-loop feedback methods.

Additionally, frameworks like ProgPrompt use natural language and pseudo-code to generate Python-based robotic tasks [10], aligning generated tasks with robot APIs for real-world execution. Despite these advancements, few approaches integrate these capabilities into a comprehensive FMS, highlighting a need for task formulation methodologies that support human-robot collaboration in dynamic, real-world environments.

2.1.2 Task Allocation

Task allocation in FMS involves distributing tasks among robots based on their capabilities and constraints to ensure balanced workload and operational efficiency [13]. This process is inherently complex in multi-robot systems, particularly when dealing with heterogeneous robots with varying capabilities and field applications [14]. It consists of two main components: task assignment and coordination of execution, which are interdependent and essential for successful operations [15].

Different paradigms have been developed for task allocation, including centralised and decentralised approaches [16]. Centralised systems offer simplicity and cost savings but lack scalability and are prone to single points of failure. In contrast, decentralised systems improve robustness and flexibility, albeit sometimes at the cost of optimal solutions [17]. Market-based methods, using auction mechanisms, excel in adaptability and handling dynamic environments but can be complex when formalising cost functions [18]. Optimisation-based methods employ mathematical models for near-optimal solutions, efficiently handling constraints and noisy input data [19]. Although these methodologies address various aspects of task allocation, the integration into an FMS for dynamic, multi-robot collaboration in real-world applications remains a challenge.

2.1.3 Coordination

Coordination in FMS ensures synchronised operation among multiple robots to optimise performance and avoid conflicts or redundancies [20]. This is particularly challenging in environments with diverse robot capabilities and dynamic conditions. One of the earliest methods for robot coordination is the use of shared potential fields, which facilitate dynamic role assignment based on robot positions and target locations [21]. Through such methods, robots navigate without interfering with each other, enabling efficient collaborative objectives.

The CoMutaR framework extends this concept by concurrently tackling task distribution and coordination through coalition formation, considering factors like communication bandwidth, processing power, and robot positions [15]. By adopting an auction protocol, CoMutaR automatically forms robot teams capable of multitasking. Taxonomies for multi-robot coordination, such as those proposed by Verma & Ranga [22], classify coordination strategies into dimensions like static versus dynamic and centralised versus decentralised. These frameworks inform the design of FMS solutions that can handle the complexities of real-world applications and ensure efficient, human-centric robot collaboration.

2.1.4 Optimisation

Optimisation in FMS focuses on refining resource allocation and maximising operational efficiency while minimising conflicts [23]. Multi-objective optimisation strategies are particularly significant in dynamic environments, incorporating evolutionary algorithms [24] and market-based approaches [25] to address constraints like time, energy, and task prioritisation.

An example is the work by Tomidis et al., which uses multi-agent collaboration and auction-based methods to offer task assignments beyond mere compatibility, including factors like efficiency and resource utilisation [26]. Their approach demonstrates the importance of standardisation in modelling task information, making it adaptable to various domains. The integration of multi-objective optimisation within FMS is essential to manage the multifaceted challenges of real-world environments, ensuring harmonious robotic fleet operations.

2.1.5 Adaptability

Adaptability in FMS refers to the system's ability to respond to changing operational conditions and dynamic environments efficiently. It ensures the FMS maintains high performance levels despite variations in workload, environmental conditions, or unexpected challenges. Adaptive task planning, as demonstrated in multi-robot smart warehouses, dynamically adjusts assignments to improve system efficiency [27]. This approach emphasises centralised task management that adapts to robot locations and resource statuses.

From a distributed perspective, research into evolutionary computation and fuzzy logic has shown how multi-robot systems can evolve their task allocation and coordination in response to changing demands [28]. Adaptive workload allocation also plays a critical role in human-robot collaboration, distributing tasks efficiently between humans and robots to optimise performance and ensure cohesive operation [29]. By integrating these mechanisms, an advanced FMS can effectively navigate uncertainties, thereby going beyond repetitive tasks to meet the dynamic demands of modern robotic operations.

2.1.6 Resilience

Resilience in FMS is the system's ability to recover from failures and maintain operations under adverse conditions. It involves learning from disruptions, adapting to challenges, and evolving to respond better to complex environments [30]. This concept extends beyond robustness, requiring real-time adaptation and system-wide reorganisation.

Prorok et al. distinguish between robustness and resilience, emphasising the need for multi-robot systems to reconfigure capabilities in response to unexpected challenges [31]. They identify anticipatory policies and real-time adaptation as key resilience strategies. Additionally, addressing cybersecurity vulnerabilities, such as Sybil attacks, is crucial for maintaining network integrity in multi-robot systems [32]. Thus, resilience in FMS encompasses a system's ability to learn, adapt, and maintain functionality amidst disruptions, ensuring robustness in real-world environments.

2.1.7 Summary

FMS in robotics are crucial for effective multi-robot and human-robot collaboration, involving task formulation, allocation, coordination, optimisation, adaptability, and resilience. Despite numerous advancements, most existing solutions operate in isolated or controlled environments and fail to integrate these components into a unified framework. This paper emphasises the necessity of a comprehensive integration to create robust FMS capable of handling real-world applications. Within this work, special attention is given to task formulation as a core feature of the FORMIGA system, leveraging AI-generated task definitions to enhance flexibility and adaptability. While other features also play significant roles in FORMIGA's design, in particular task allocation and coordination, the current focus is on establishing a solid foundation, laying the groundwork for a more holistic and integrated FMS in future iterations.

2.2 FMS Frameworks

Although no existing framework comprehensively encompasses all the features outlined in the previous section, several noteworthy attempts have emerged, both as open-source initiatives and commercial offerings. We will briefly explore the most significant of these frameworks in the following sections.

2.2.1 NVIDIA Isaac ROS Mission Dispatch

Mission Dispatch and Client[2] within NVIDIA's Isaac establish a standardised, open-source framework for efficient task allocation and tracking between fleet management systems and ROS 2 robots. This system operates through VDA5050, an open communication standard tailored for robot fleets, ensuring seamless wireless messaging via MQTT, a lightweight protocol ideal for Internet of Things (IoT) applications.

Mission Dispatch, available as a containerised micro-service for download from NGC or as open-source code on NVIDIA's Isaac GitHub repository, offers flexibility for integration into various fleet management systems. Its compatibility with other open-source ROS 2 clients, including the VDA5050 Connector developed by OTTO Motors and InOrbit, underscores the commitment to interoperability and innovation in fleet management. Mission Client, designed to align with ROS 2 Humble, further simplifies task assignment and tracking by being readily available as a package in the NVIDIA Isaac ROS GitHub repository. This comprehensive framework streamlines fleet operations, allowing robot manufacturers to focus on their unique differentiators while promoting innovation within the robotics community.

2.2.2 ROOSTER

ROOSTER[3], a ROS-based open-source project, is designed as a versatile fleet management solution that excels in task allocation, scheduling, and autonomous navigation. At its core, the Fleet Manager application serves as a centralised control hub, offering a comprehensive view of the current fleet and job statuses while enabling seamless

[2] https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_mission_client
[3] https://github.com/ROOSTER-fleet-management

order placement. Users can efficiently manage tasks and orders, with features that allow for sorting and filtering based on priority and other criteria.

Complementing the Fleet Manager, the Mobile Executor (MEx) Sentinel operates in the background, meticulously tracking other MEx within the fleet. This component not only monitors MEx IDs and statuses but also facilitates dynamic task assignments and unassignments. Additionally, ROOSTER's Job Manager intelligently processes incoming orders, creating jobs with associated tasks and efficiently allocating them based on priority and proximity. These features collectively enhance fleet coordination, making ROOSTER a valuable asset for optimising heterogeneous fleet operations.

2.2.3 Open-RMF

Open-RMF[4] represents a powerful fleet management solution, rooted in ROS 2, designed to facilitate seamless coordination across heterogeneous fleets of robotic systems. Its strength lies in its collection of reusable and scalable libraries and tools that enhance interoperability and resource optimisation within diverse robotic environments. RMF utilises standardised communication protocols to efficiently manage critical resources, including robots, lifts, doors, and passageways. The RMF Core component adds intelligence to the system by optimising resource allocation and proactively preventing conflicts over shared resources.

RMF's feature-rich ecosystem extends to various utilities and tools, such as demos showcasing its capabilities in different environments. The Traffic Editor empowers users to create traffic patterns and annotate floor plans for use in RMF, enhancing virtual simulation environments. Free Fleet, an open-source robot fleet management system, caters to robot developers who seek an open-source fleet manager for their projects. Additionally, RMF offers visualisation and control tools like the RMF Schedule Visualiser and RMF Web UI, ensuring operators have user-friendly interfaces to interact with the system.

For simulation enthusiasts, RMF provides simulation plugins available in Gazebo and Ignition, along with freely distributable simulation assets to accelerate simulation efforts. The task planner algorithm takes into account various factors, including agent priorities, proximity, capabilities, and skills, while also considering agent constraints such as battery levels.

2.2.4 InOrbit

InOrbit[5] is a cloud-based robot operations platform (RobOps) designed to enhance the efficiency of deploying and managing robot fleets across various operational complexities. It addresses key challenges faced in dynamic industrial settings, such as factories and warehouses, emphasising advanced robot management to maintain operational integrity amidst interactions with humans and machinery.

A central feature of InOrbit is its focus on managing autonomy exceptions, including localisation failures, navigation issues, and mechanical malfunctions. The platform's InOrbit Control component offers a customisable dashboard interface for

[4] https://www.open-rmf.org/
[5] https://www.inorbit.ai/

comprehensive fleet management and real-time data visualisation. This system allows for the creation of role-specific views, providing tailored insights for different operational roles, such as executive overviews, fleet health, and individual robot diagnostics. This functionality enhances operational visibility and streamlines control across the entire robot fleet.

InOrbit also supports operational scalability, from small deployments to managing thousands of robots, by mitigating autonomy exceptions and operational costs associated with scaling [33]. Its strategic use of cloud-based platforms aids in incident management, optimising robot utilisation, and implementing iterative improvements. Additionally, the time capsule feature (Fig. 1) provides historical data analysis, enabling users to "travel back in time" to review past operational states and identify patterns, root causes of issues, and optimisation opportunities. This retrospective insight is critical for continuous improvement, driving both robot performance and workflow refinement.

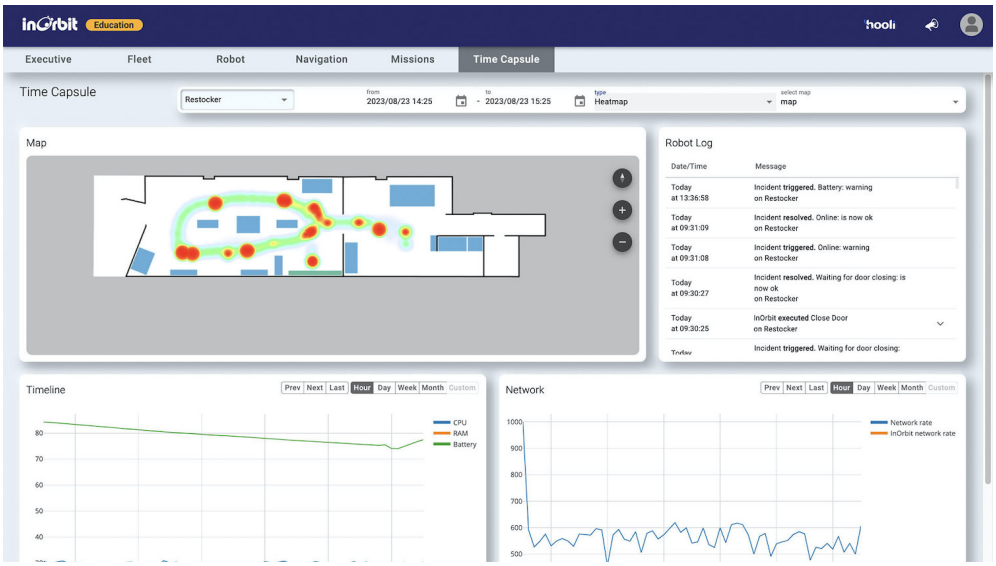


Fig. 1: InOrbit time capsule feature showcasing the robot log and timeline widgets where correlating entries can be observed such as trajectory carried out by the robot and network usage.

While InOrbit offers a robust framework for robot operations, its reliance on predefined scenarios and structured data may limit adaptability in unstructured and dynamic environments, such as field applications. These environments demand real-time decision-making based on incomplete information, potentially requiring more direct human intervention or advanced AI capabilities. This highlights the need for ongoing platform development to enhance versatility across a broader range of real-world applications.

2.2.5 Meili Robots

Similar to InOrbit, Meili Robots[6] offers a universal FMS designed to streamline the management of mobile robots across various industrial settings. A key strength of Meili Robots is its ability to integrate diverse robot fleets onto a single platform, regardless of their types or brands, eliminating interoperability issues and enhancing operational efficiency. This integration aligns with Meili Robots' mission to transform supply chain and logistics operations [34].

Key features include task allocation, map editing, traffic control, and user management, all engineered for seamless integration with enterprise resource planning (ERP) systems [35]. This approach facilitates easy on-boarding of new robots and allows for fleet scalability without additional infrastructure, enabling smooth human-robot collaboration. Interoperability remains a central focus, ensuring robots from different manufacturers operate cohesively, thus reducing operational risks like collisions and delays.

The FMS primarily supports robotic platforms using ROS and ROS2 technology, employing three advanced communication protocols for efficient data exchange (Fig. 2). It uses HTTP via RESTful API and Webhooks for flexible user access and automated workflows, WebSockets for bi-directional communication to enhance system responsiveness, and MQTT for lightweight, reliable messaging in resource-constrained environments.

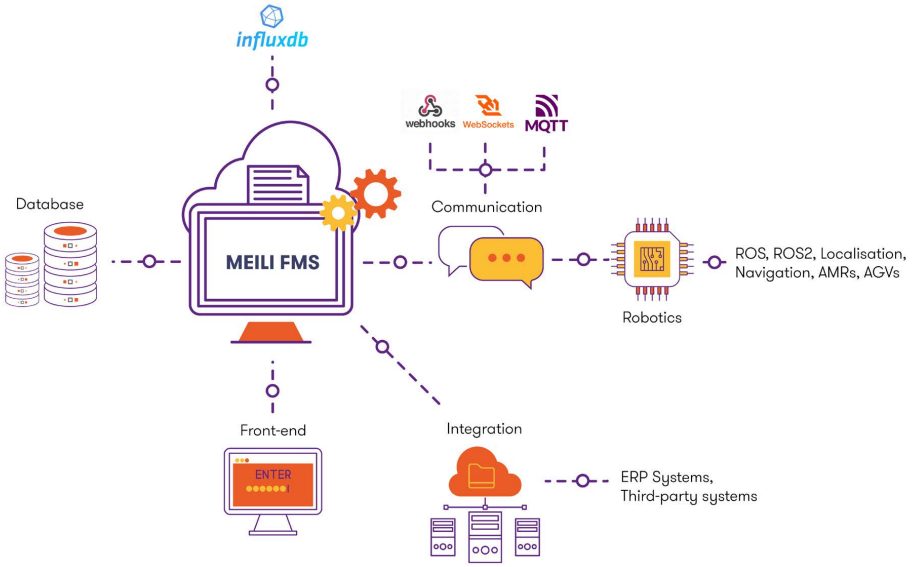


Fig. 2: The three main communication protocols integrated into Meili Robots FMS, with a strong compatibility with ROS / ROS2 robotic platforms.

[6] https://www.meilirobots.com/

Meili Robots offers robust fleet management and operational efficiency, making it a pioneering solution in automation. However, like InOrbit, its reliance on predefined operational environments may limit adaptability in highly dynamic or unstructured field applications, posing challenges in unforeseen conditions.

2.2.6 Formant

Formant[7] is a cloud-based FMS designed to enhance the operational efficiency of robot fleets across industries. Built on a foundation of modern, cloud-native technology, Formant places a strong emphasis on security, employing end-to-end encryption, zero-trust authentication, and scoped permissions to protect data integrity and privacy [36].

The platform offers a suite of features for managing heterogeneous fleets, allowing integration of robots from various manufacturers, thus providing a unified operational view. Its focus on remote command and control enables centralised fleet operations, streamlining data monitoring and scaling capabilities [37]. This central command system visualises field data, aggregated by location, customer, and fleet, facilitating swift responses to operational trends.

A standout feature of Formant is its workflow automation, particularly for teleoperation, fleet management, and localisation. These workflows, compatible with ROS, allow operators to set target positions and paths on the map with ease (Fig. 3). The re-localisation functionality further improves adaptability, providing tools to adjust robot positions in response to environmental changes. This enhances operational fluidity by allowing robots to re-orient based on a base reference frame.

Formant represents a significant advancement in the field of robot fleet management. Its approach to security, coupled with features for interoperability and centralised control, positions it as a promising solution for businesses aiming to leverage robotic technology to its fullest potential. Nevertheless, the journey towards optimising Formant for the complexities of unstructured environments highlights the ongoing evolution of FMS solutions in meeting the diverse needs of modern industries. The platform's design, while robust in standardised environments, might encounter challenges in adapting to highly dynamic or unforeseen conditions that characterise field operations. Preliminary efforts have been made to employ Formant within agriculture applications, though little insights have been shared to date [38].

2.2.7 Summary

The comparative analysis presented in Table 1 offers insights into how different FMS address the critical features discussed in Section 2.1. Notably, Open-RMF scores highly across most categories, particularly excelling in coordination and adaptability, reflecting its versatility in managing heterogeneous fleets under structured and previously known settings. ROOSTER and Formant also show strong performance, especially in task allocation and coordination, suggesting their effective handling of multi-agent operations. On the other hand, Meili Robots and InOrbit demonstrate a more balanced approach, with solid performance in key areas, but potential limitations

[7] https://formant.io/

Fig. 3: The re-localisation workflow in Formant allows operators to adjust a robot's pose using intuitive controls for movement and rotation, aligning it with map features for precise positioning, enhancing navigational accuracy.

in resilience and optimisation, particularly in dynamic environments. NVIDIA Mission Dispatch, while competent, appears to be more specialised, showing its strengths in task allocation rather than comprehensive fleet management. Overall, while each framework has its merits, the diversity in performance across features emphasises the need for adaptable and integrated solutions that can address the wide spectrum of requirements in real-world field applications, wherein most scenarios might be unknown and unstructured.

Table 1: Comparison of Fleet Management Systems based on the key features introduced before. More circles indicate a better handling of the respective feature.

Features	NVIDIA	ROOSTER	RMF	InOrbit*	Meili*	Formant*
Task Formulation	●●●	●●●●	●●●●●	●●●●	●●●●	●●●●
Task Allocation	●●●●	●●●●●	●●●●●	●●●●	●●●	●●●●
Coordination	●●●	●●●●	●●●●●	●●●●	●●●	●●●●
Optimisation	●●●	●●●	●●●●	●●●●	●●●	●●●
Adaptability	●●●	●●●●	●●●●●	●●●●	●●●●	●●●●
Resilience	●●●	●●●●	●●●●	●●●●	●●	●●●●

*Commercial frameworks.

3 FORMIGA

FORMIGA is a cutting-edge FMS designed to coordinate the operations of both robotic and human agents in dynamic and complex environments. It provides a versatile platform for managing multi-agent systems, optimising the deployment of resources, and ensuring seamless interaction between robots and human operators.

The development of FORMIGA involved a structured design process, beginning with the foundational ROOSTER framework. ROOSTER, an open-source ROS project introduced in Section 2.2.2, provides the essential functionalities for heterogeneous fleet management, such as task allocation, scheduling, and autonomous navigation. FORMIGA builds on this foundation, incorporating additional features tailored to the specific needs of challenging field applications involving both humans and robots.

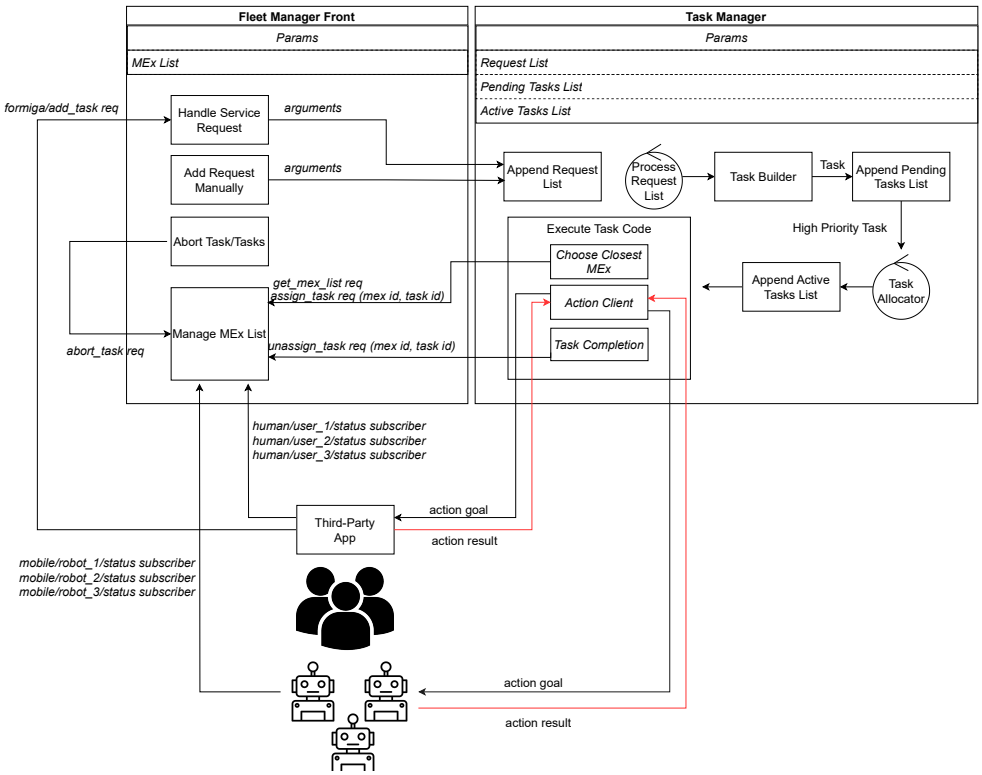


Fig. 4: FORMIGA General Overview

Figure 4 shows an overview of the FORMIGA ecosystem, highlighting key components that optimize fleet management. The architecture of FORMIGA integrates several key components addressed next:

- ROS: FORMIGA utilises ROS to facilitate seamless communication between agents (be it humans or robots), ensuring a cohesive operational framework.
- User Interface: FORMIGA's graphical interface enables operators to manage fleets of agents, monitor real-time system status, and configure agents accordingly (e.g. defining key ROS Actions).
- User-tailored task coding: FORMIGA allows to code tasks as collections of ROS Action Clients to the Action Servers enabled by agents.
- Task Management: FORMIGA builds on ROOSTER's task management with a robust system for defining, allocating, and executing tasks, enabling multi-agent collaboration and dynamic resource optimisation.
- LLM-generated task coding: FORMIGA utilises a large language model (LLM) to streamline and reduce the effort involved in task coding.

3.1 ROS Integration and Standardisation

The integration of ROS and the establishment of a robust standardisation process are pivotal to the effective functioning of the FORMIGA. These elements ensure seamless communication, interoperability, and efficient management of diverse robotic agents within the fleet. Additionally, ROS provides a flexible framework for writing robot software, offering a structured communication system between various components of a robotic system.

ROS adoption offers numerous advantages in FORMIGA, one of which is the use of namespaces. Namespaces in ROS are crucial for organising and managing data flow within a multi-agent system. In FORMIGA, namespaces are implemented using a standardised naming convention to ensure clarity and prevent conflicts. This structure generally follows the pattern `agent_type/agent_name/status`, allowing for straightforward identification and management of different robotic agents within the system. Examples of this naming convention include:

- `/ground_robot/robot_1/status`
- `/areal_robot/robot_2/status`
- `/human/user_1/status`

In the context of FORMIGA, however, ROS integration encompasses other critical aspects, including the key communication mechanisms topics, services and actions.

3.1.1 Topics: MEx Status

ROS utilises a publish-subscribe model for inter-process communication, where nodes (individual processes) publish messages to topics, and other nodes subscribe to these topics to receive messages. This model ensures a decoupled and scalable communication system within the fleet.

At the core of FORMIGA's decision-making is the Mobile Executor (MEx) - a term originally used in ROOSTER to describe any mobile vehicle capable of executing tasks. While the term is retained in FORMIGA, it has been expanded to include both robots and human agents.

FORMIGA subscribes to the status topic of each MEx, published by robots and humans (e.g. through third-party apps), allowing it to continuously monitor their, availability, battery levels, pose, etc. FORMIGA then enhances MEx statuses with additional information available at the fleet level (e.g. task being executed by agent).

For instance, Fig.5 illustrates the typical status message `MexStatus.msg`. The completion performed by FORMIGA is highlighted in yellow. While `MexStatus.msg` is a custom ROS message defined to encapsulate the specific data structures required by FORMIGA, it makes use of existing ROS message types to maintain compatibility and leverage community-tested formats. This approach ensures that the system adheres to established standards and benefits from the robustness of widely-used message types.

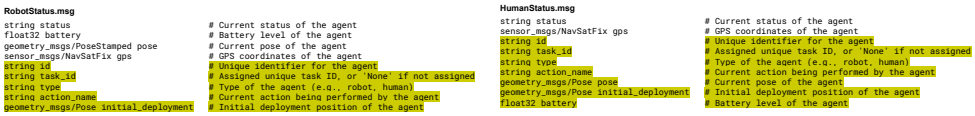


Fig. 5: MexStatus.msg structure for a Mobile Executor (MEx) in FORMIGA

As observed in Fig. 5, each field in the message provides crucial information needed for the operation and monitoring of both robotic and human agents, with multiple common parameters shared between both. This includes the the pose of the agent in the Cartesian local coordinate system (`geometry_msgs/Pose`), as well as its GNSS coordinates (`sensor_msgs/NavSatFix`).

3.1.2 Services: Task Request

ROS services are synchronous, providing a request-response communication pattern. In FORMIGA, a series of services are made available for direct interaction and immediate feedback between components, namely:

- **`/formiga/add_task`:** Requests a task with specified parameters, adding it to the task queue for processing and execution.
- **`/task_manager/abort_task`:** Cancels an ongoing or pending task by publishing a goal to the cancel topic for all associated actions.
- **`/fleet_manager_front/get_mex_list`:** Retrieves the status of all agents (MExs) to determine their availability, distance, and task progress.
- **`fleet_manager_front/assign_task`):** Allocates a task to a specific MEx, updating its status from `STANDBY` to `ASSIGNED`.
- **`/fleet_manager_front/unassign_task`:** Reverts the task allocation, updating the MEx status back to `STANDBY`.

This feature is particularly relevant as it increases user accessibility. For instance, while new tasks can be set manually using the user interface, third-party apps can also do so by making use of the `formiga/add_task` service. This facilitates the inclusion of

human knowledge to guide the autonomy of robot teams in order to realise behaviours that are in line with the desired domain objectives, safe and acceptable by involved human users. Furthermore, it does so by embracing ROS standards, namely by making sure that these services are used for short request-response interactions. For instance, `formiga/add_task` service is specified with the following parameters:

- **Priority:** `LOW`, `MEDIUM`, or `HIGH`;
- **Keyword:** Task name;
- **Arguments of the task:** Dependent on the actions composing the task (see next section).

3.1.3 Actions: Task Decomposition

Unlike services, ROS actions are often used for long-running behaviours, providing feedback, preemption, and result mechanisms. FORMIGA employs actions for complex operations, such as requesting a robot or human to move to a given location, with action servers managing these tasks within the robots or humans, and action clients within FORMIGA requesting them.

ROS actions are a fundamental part of the FORMIGA framework, enabling asynchronous, goal-oriented task execution. The use of ROS actions allows FORMIGA to handle complex tasks that require feedback and the ability to preempt ongoing operations. Key features include:

- **Goal Management**: FORMIGA can send goals to action servers (enabled by robots and humans), which then execute the actions and provide feedback on their progress. This is particularly useful for tasks that involve multiple actions and/or require continuous monitoring.
- **Preemption**: Actions in ROS can be preempted if a higher-priority task needs to be executed. FORMIGA utilises this capability to dynamically adjust to changing operational conditions and priorities, as well as human requests.
- **Feedback and Result Handling**: During the execution of an action, feedback is continuously sent to the action client, allowing for real-time monitoring and adjustments. Once the action is complete, the result is sent back to the client (i.e. FORMIGA), providing information on the outcome.

While section 3.2.2 further describes the concept of tasks within FORMIGA, actions are a fundamental part of it, wherein the `action_client` function is called with the arguments `robot_type`, `robot_name`, `action_name`, `goal_arg`, `task`, and `last_task=False`). The task is managed as a global variable containing the task ID and status, which are handled during the execution of the task code. The `last_task` flag is set to `False` by default and should only be set to `True` for the last action of the task.

After the goal is sent to the action server, the robot's status changes from `ASSIGNED` to `EXECUTING ACTION`, and the fleet tab allows to visualise the ongoing action name. The callback of the action result checks if the action is `SUCCEEDED` or `ABORTED`. If the action succeeds and the `last_task` flag is `False`, the code continues execution, indicating there are more actions to follow. If the flag is `True`, it is recognised as

the last action and the Task Completion callback runs. In this callback, the robot is unassigned, its status changes from `EXECUTING ACTION` to `STANDBY`, and the task is removed from the Active Tasks List. If any action results in `FAILURE` or `ABORTION`, the task status changes from `ACTIVE` to `ABORTED`, and Task Completion is called again.

3.2 User Interface

The FORMIGA interface is designed to be intuitive and user-friendly, ensuring efficient management and integration of robots and humans. The system offers a seamless experience for setting up and managing human-robot teams. Fig. 6 depicts the user flow of FORMIGA, which is further explained in the upcoming sections.

3.2.1 System setup and monitoring

FORMIGA simplifies the process of integrating agents into the system, allowing users to easily scan for and add available agents. By using a search bar and typing `status`, all agents present in the network are listed. Users can then select the desired robots and humans, adding them to the system for subsequent fleet management. As previously stated, FORMIGA recognises that agents can include humans and, therefore, additionally devices are required to make use of third party apps that can be bridged with ROS (e.g. smartphones). The type of each agent, such as ground robot, aerial robot, human, etc., is automatically detected upon selection based on the definition of namespace addressed in Section 3.1. Previously stored fleets can also be loaded from a SQLite database, allowing for quick integration of previously configured robots.

After the selection of agents to be included for fleet management, users can scan for all available action servers enabled by these. These actions, previously addressed in Section 3.1.3, can then be added to FORMIGA, which automatically builds the actions in case these are not recognised by the system.

With this in place, users can efficiently monitor the system in real-time. The integrated MEx statuses (Section 3.1.1), requested services (Section 3.1.2), and their ongoing actions (Section 3.1.3), provide fundamental feedback about task IDs, battery levels, and overall mission progress. A Leaflet-based visualisation, combining a OpenStreetMap (OSM) map view with a requested task pipeline, allows to assess unique task IDs, priority levels (`LOW`, `MEDIUM`, and `HIGH`), keywords (e.g., task names), status (`PENDING`, `ACTIVE`, `ASSIGNED`, `ABORTED`, `SUCCEEDED`), and assigned agents.

With these features, FORMIGA ensures a smooth and efficient process for managing and integrating robots and agents into the fleet, enhancing overall system performance and ease of use. Furthermore, it facilitates the mediation of robot's behaviours by the humans, now only enabling supervision, but also (occasional) intervention during operation.

3.2.2 Task Coding

As previously addressed in Section 3.1.3, Tasks are decomposed by multiple actions, be those performed by robots, humans, or both. More generally, tasks are defined as pythonic functions consisting of multiple ROS action clients, each sending goals

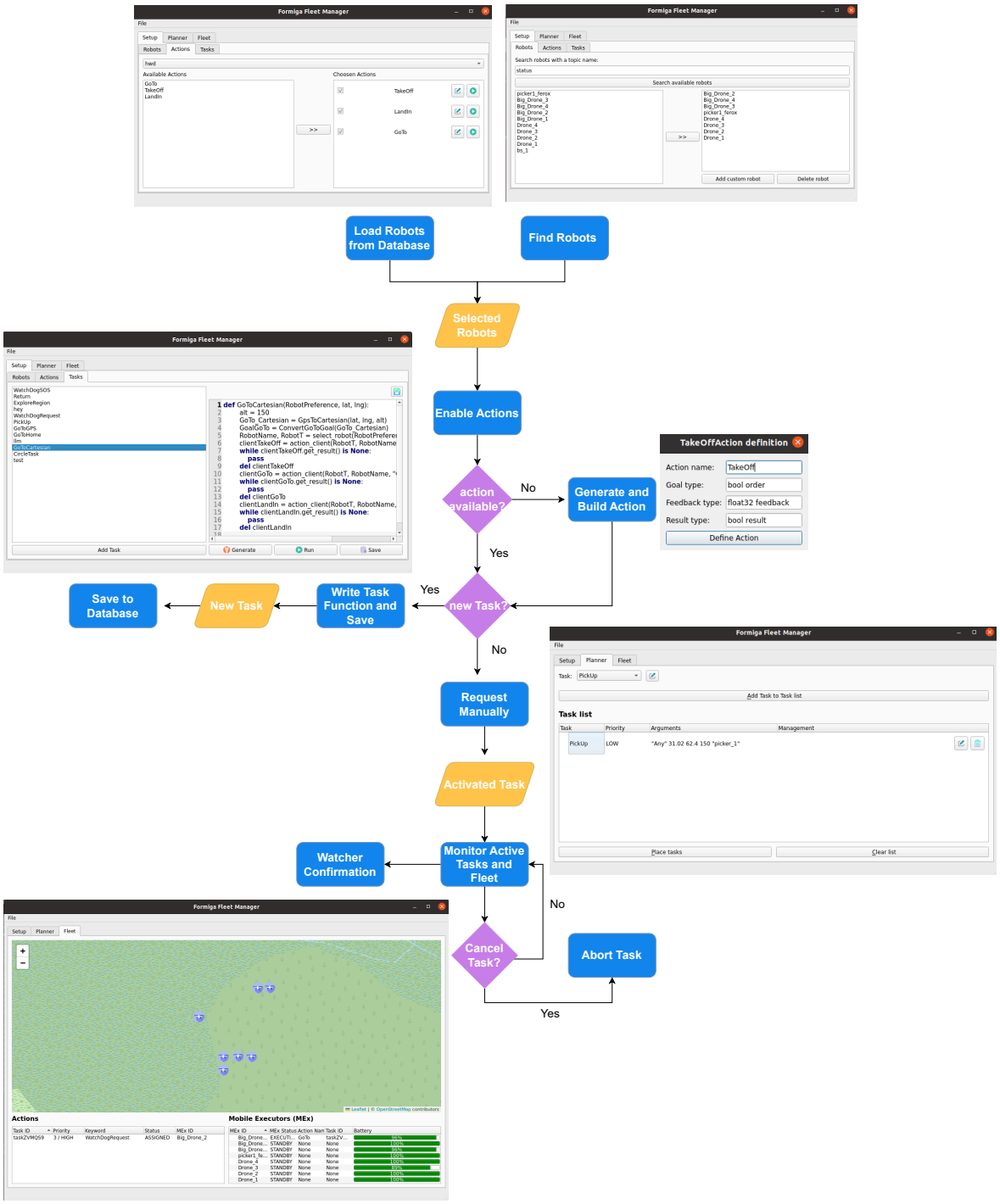


Fig. 6: User flow of the FORMIGA interface, illustrating the streamlined process for managing and integrating human-robot teams efficiently.

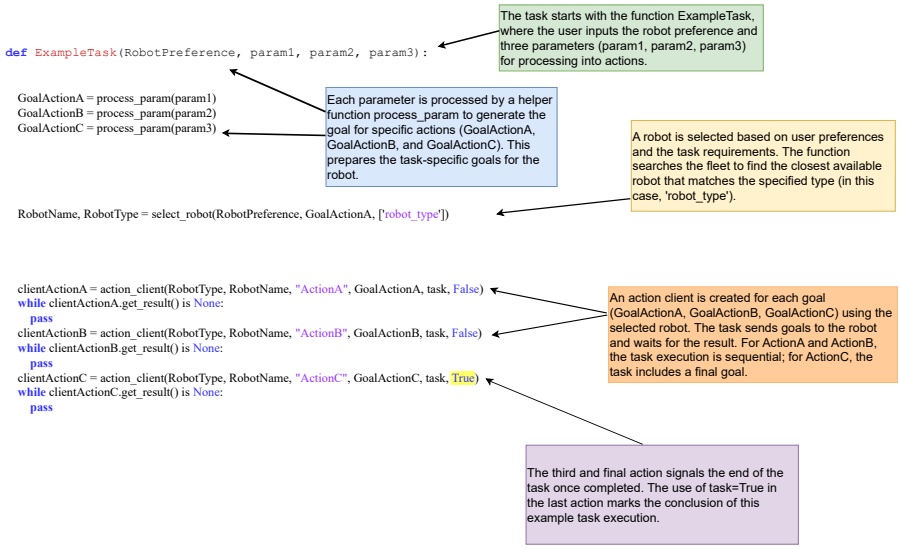


Fig. 7: User defined code of ExampleTask, selecting and commanding a robot to execute sequential actions.

to agents to accomplish specific objectives. During task execution (i.e., the execution of these pythonic functions), FORMIGA continuously updates the statuses of the MExs involved, ensuring efficient, coordinated fleet operations. Therefore, task coding in the FORMIGA ecosystem involves creating custom Python functions that define specific actions and behaviours for robots. These tasks run in the background using threads, ensuring the interface remains responsive and allowing multiple tasks to execute concurrently.

A typical task function starts by defining necessary parameters and converting inputs into actionable goals for the robots. The function then selects the appropriate robot based on the user's preference or allows FORMIGA to choose the best available option. Finally, it sends commands to the robot and waits for the results, handling any failures and completing the task by releasing resources.

An abstract example of a task function designed to accomplish a task with three actions as shown in Fig 7. The function uses FORMIGA's predefined functions (`select_robot`, and `action_client`) to manage task execution. The `select_robot` function determines the appropriate robot based on user preferences or allows FORMIGA to select the best available option. The `action_client` function sends goals to action servers, ensuring the specified actions are carried out correctly.

The `action_client` function sends goals to the action server using the following format:

```
1
2 action_topic = "/" + RobotType + "/" + RobotName + "/action/" + "/
      ActionA/"
```

Actions must be predefined in FORMIGA, and the GoalAction must be correctly formatted. The task function waits for each action to complete before proceeding, ensuring sequential execution and error handling.

The user-defined task coding approach provides flexibility and control for operators to tailor specific actions and behaviors. However, with increasing complexity and real-time demands in fleet operations, manually coding each task can become inefficient. This challenge is addressed in section 3.4, where we explore how integrating Large Language Models (LLMs) automates the task generation process. This not only simplifies task creation but also enhances adaptability, enabling faster, more efficient responses to evolving mission requirements and environmental conditions within the FORMIGA ecosystem.

3.3 Task Management

FORMIGA builds on ROOSTER's robust task management system, enabling efficient definition, allocation, and execution of tasks to facilitate multi-agent collaboration and dynamic resource optimisation. The process begins when the Task Manager node receives a task request and appends it to the global order list. A callback function, Task Builder, processes this list every second, constructing tasks from the provided parameters and adding them to the Pending Tasks List.

Once tasks are ready, the Task Allocator retrieves the highest-priority task and moves it to the Active Tasks List for execution. During task execution,the user-defined or LLM-generated task code is initiated, which includes a call to the `select_robot` function. This function assesses user preferences and the type of actions required, subsequently requesting an updated MEx list from the Fleet Manager. It then invokes the `choose_closest_mex` function, which filters available MEx units with a status of STANDBY, matching the task's requirements. This function calculates the distance to the target location for each available agent and selects the closest MEx based on this distance, ensuring that the optimal agent is chosen to minimise response time and enhance operational efficiency.

After selecting the robot, an `assign_task` service request is sent to the Fleet Manager Front module. The selected robot's status transitions from `STANDBY` to `ASSIGNED`, marking its readiness for the task.

Once all actions within the task are executed, as detailed in Section 3.1.3, the system carefully manages task completion and status updates. If the task is successfully completed, the robot's status returns to `STANDBY`, and the task is removed from the Active Tasks List. In the case of failure or abortion, the task status is updated to `ABORTED`, triggering the task completion processes to ensure reliable management of ongoing operations.

As tasks may involve multiple actions and require collaboration among several agents, the `select_robot` function may be invoked multiple times within a single task. This dynamic allocation allows for the assignment of additional robots as needed throughout the task execution, adapting to changing conditions and ensuring a coordinated response. For example, in the `ExampleTask` function (in Fig. 7), different action clients can be created for various actions, allowing for seamless integration and execution of commands for the selected robot(s). This flexibility further

enhances FORMIGA's capacity for efficient resource management and collaborative task execution.

3.4 LLM-Generated Task Coding

In recent years, the rise of Large Language Models (LLMs) has demonstrated remarkable potential for generating executable code, enabling advancements in general planning. This aligns perfectly with the dynamic nature of human-robot applications, where the generation of new, context-specific tasks is often necessary. LLMs, trained on vast programming corpora, have been shown to perform well in generating pythonic structures, as evidenced by state-of-the-art studies, like ProgPrompt [10]. In the FORMIGA ecosystem, we leverage Large Language Models (LLMs) to automate and streamline the development of complex robotic behaviours, reducing the need for extensive manual coding. As presented in the previous sectiopn, FORMIGA leverages the structured nature of Python to model tasks as a sequence of actions. This concept is directly compatible with LLMs' ability to understand and generate pythonic functions. This integration not only streamlines the task-coding process, but also augments FORMIGA's adaptability, allowing it to handle unforeseen operational challenges without extensive manual intervention. Consequently, this integration supports the execution of collaborative actions in real-world scenarios, enhancing the effectiveness of human-robot interaction within the ecosystem.

Fig. 8 depicts a general overview on how LLM-generated task coding is integrated within FORMIGA. In the following subsections, we delve into the specifics of the LLM framework adopted in FORMIGA, the methodology behind constructing programming language prompts for task generation, and insights into the actual task coding process.

3.4.1 LLM Framework

Integrating an LLM directly within FORMIGA provides the opportunity to harness powerful language models for task coding without relying on external infrastructure. However, a local deployment of an LLM, particularly models like GPT-4 or LLaMA 3, demands high computational power, often necessitating specialised hardware, such as NVIDIA A100 GPUs or custom AI accelerators. The hardware and energy costs associated with maintaining such infrastructure can be prohibitive, especially when targeting on-field human-robot operations, where mobile and compact computing solutions are preferred. Given these constraints, we opted to explore existing cloud-based LLM frameworks that offer high-performance computing without the need for costly local hardware setups.

Some potential alternatives to local LLM integration include:

- **OpenAI API:** Provides access to powerful models like GPT-4 via a cloud-based subscription service, making it possible to generate and fine-tune code with minimal local computational requirements.
- **Hugging Face Inference API:** Offers various state-of-the-art models, including the LLaMA series, in a flexible environment that allows developers to use models on-demand, with a focus on open-source solutions.

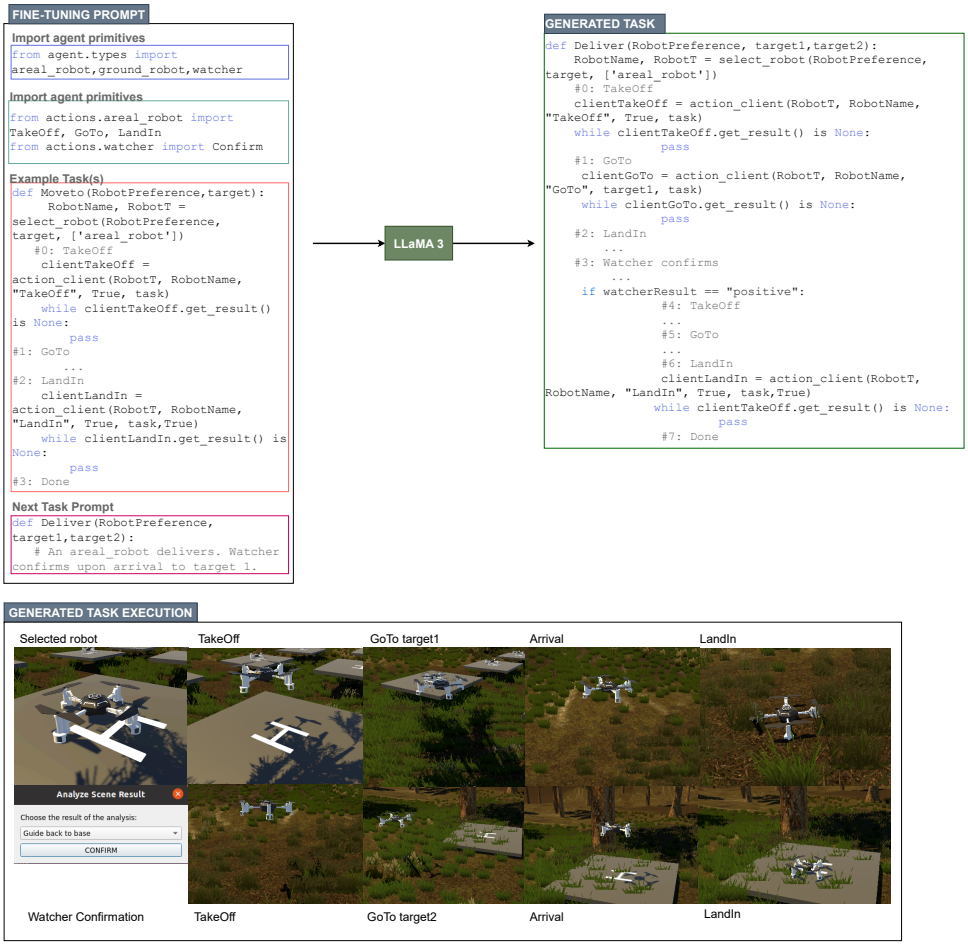


Fig. 8: LLM-Generated Task Execution with LLama3.

- **Azure OpenAI Service:** Integrates Microsoft Azure's infrastructure with OpenAI's language models, enabling scalable and secure access to LLMs within enterprise environments. This platform also includes specific optimisations for high-performance computation and cloud-based deployment.
- **Groq Platform:** A specialised cloud platform designed to work seamlessly with large-scale language models, providing low-latency access to AI capabilities and supporting the robust computational needs required for real-time task generation in robotics.

After evaluating these options, Groq[8] was chosen as the computing platform for FORMIGA due to its focus on handling high-throughput, low-latency AI computations that align with the operational demands of the fleet management system. In

[8] https://groq.com/

conjunction with the LLaMA 3 model, specifically the `llama3-70b-8192` configuration, Groq's infrastructure excels in generating sophisticated Python code as required by FORMIGA. The `llama3-70b-8192` model, with its 70 billion parameters, offers an exceptional balance between complexity and efficiency, ensuring that FORMIGA can generate reliable and context-aware task codes dynamically. Groq's platform not only meets the high computational requirements for real-time code generation but also offers the flexibility needed for integrating LLM-generated code directly into the ROS-enabled ecosystem, making it the optimal choice for our framework.

3.4.2 Programming Language Prompts

To optimise the LLaMA 3 model for our specific use case, we engaged in a process of fine-tuning approach by drawing inspiration from methodologies, such as ProgPrompt [10]. This fine-tuning was aimed at adapting the model to the domain-specific requirements of the FORMIGA ecosystem, advocating for a more structured approach to fine-tuning, where the model is trained not only on a wide variety of tasks, but also on programmatic specifications of available agents within an environment and related actions, enabling it to generate Python code that is both functionally correct and contextually appropriate for human-robot collaborative tasks.

The fine-tuning process was designed to help the model internalize the specific coding conventions and logical structures used within FORMIGA. We began by prompting the model with information on available agent types and their capabilities in the FORMIGA ecosystem, ensuring it understands which types of robots or humans can execute specific actions. Next, we provided action primitives—fundamental operations such as move, patrol, or load package—presented as Python functions to train the LLM in recognizing valid commands within the system. By integrating examples of the FORMIGA API, such as how actions are translated into commands for agents via ROS, we ensured seamless compatibility with FORMIGA's architecture. These examples also illustrated how tasks are constructed from basic actions, highlighting the relationship between function definitions and action sequences. For example, a task might instruct a drone to take off, patrol an area, and land, with the LLM generating a corresponding function that, when executed, sends these commands to the relevant agents.

By training on these examples, the model learned how to initialise parameters, interact with ROS action servers, and handle errors in a way that aligns with the operational needs of the FORMIGA ecosystem.

3.4.3 Task Coding Generation

Once fine-tuned, the model can be queried to generate new task functions that were not part of the initial training set. This capability is particularly valuable when a novel or unanticipated task needs to be coded quickly. For instance, the model can interpret a high-level goal provided by the user, map it to the appropriate agent actions, and generate Python code that defines the new task. The model automatically identifies necessary parameters, selects the suitable agent based on the context or user input, and establishes interaction with the ROS action server for execution. Fig. 8 exemplifies the kind of task structure the model can generate, highlighting how it translates

user intent into executable code compatible with the FORMIGA framework. In such cases, the LLM ensures that the generated code integrates seamlessly into the system, allowing concurrent execution with other tasks.

During the generation process, certain peculiarities were observed, such as the model occasionally producing invalid commands or referencing non-existent functions within the FORMIGA framework. If the generated code contains undefined functions or incorrect commands, Python raises exceptions during execution, preventing the task from starting. This behaviour ensures that any erroneous code is identified before execution, allowing for necessary corrections.

4 The FEROX Case Study

FEROX[9] is a project that aims to support workers collecting wild berries and mushrooms in the remote and challenging terrains of Nordic countries through the use of robotic technologies. The project places a strong emphasis on Human-Robot Collaboration (HRC) by deploying unmanned aerial vehicles (UAVs) to monitor and assist groups of workers during their field operations. This approach enhances worker safety in remote environments where access to help or assistance may be limited. The anticipated outcomes of the project include increased worker trust in collaborating with robots, higher quantities of harvested berries, improved berry quality for consumers, more efficient picking times, enhanced worker safety in remote locations, and reduced levels of worker exhaustion.

Fig. 9 illustrates the conceptual overview of the architecture aimed in the FEROX Project. The project tackles multiple dimensions falling outside the scope of this paper, including AI methods for berry yield identification and mapping [39], AI methods for human activity recognition [40], and human factors, standards and ethics [41].

4.1 The FEROX Simulator

The development of a simulator is an integral part of the FEROX project, providing a virtual environment that replicates real-world berry picking scenarios. The simulator ecosystem, illustrated in Fig. 10, encompasses a range of tools, including ROS and Unity, which work together to create a comprehensive simulation platform [42]. By leveraging these technologies, the simulator enables the testing, evaluation, and optimization of various aspects of the FEROX system in a controlled and replicable manner. This section outlines the architecture and key components of the FEROX simulator, highlighting the role of each tool and its integration within the simulation framework.

Unity[10] and ROS form the core of the FEROX Simulator, combining Unity's powerful physics and rendering capabilities with ROS's robust real-world robotics framework. Unity creates a realistic, immersive environment, while ROS ensures smooth integration with robotic systems, enabling seamless transitions from simulation to deployment.

[9] https://ferox.fbk.eu/
[10] https://unity.com/

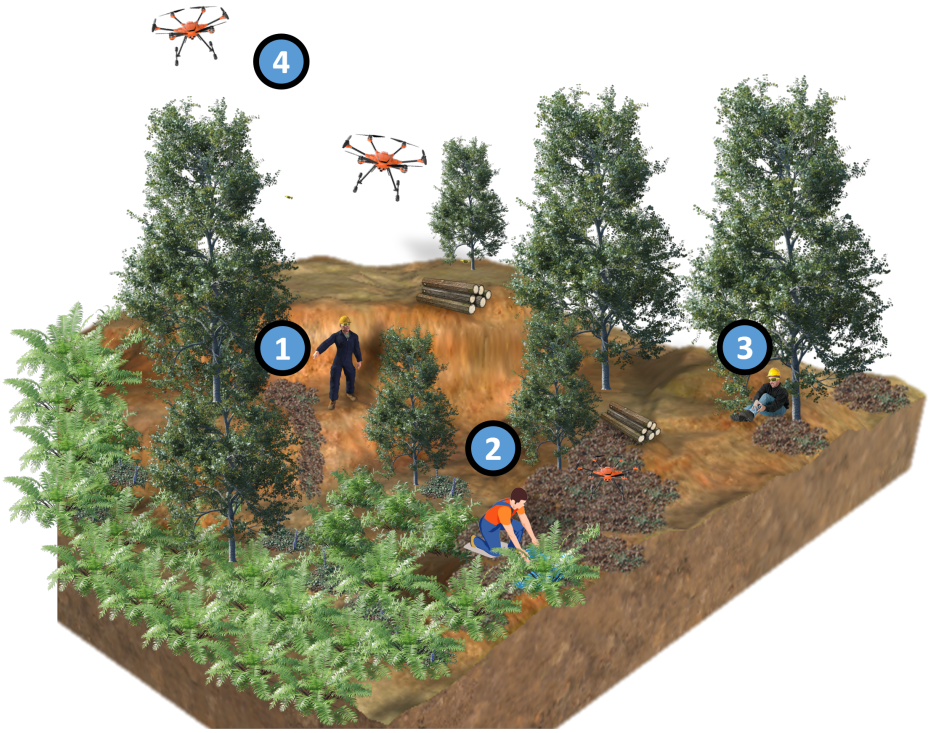


Fig. 9: An overview of the FEROX use case scenario. 1) The fleet manager acts as the watcher, coordinating humans and drones through FORMIGA and confirming triggers raised by robots; 2) Human pickers collect berries using the PickerApp for guidance and to request drone services; 3) Pickers can request tasks via the PickerApp (e.g., WatchdogRequest for guidance back to base) or have tasks automatically triggered (WatchdogSOS); 4) UAVs include LWDs for reconnaissance and HWDs for assisting humans during picking.

The ROS-TCP Connector[11] is essential in linking human and robot instances through ROS topics and services, ensuring precise synchronization between operator commands and robotic actions. It enables bidirectional communication between Unity and ROS, allowing robots to access ROS services for tasks like navigation and exploration, which can be triggered by any connected client.

4.1.1 Multi-Instance Architecture

As depicted in Fig. 10, the FEROX simulator accommodates interactions involving both single and multiple humans and robots, supporting various short- and long-term HRC scenarios. The FEROX Simulator comprises two distinct types of instances:

[11] https://github.com/Unity-Technologies/ROS-TCP-Connector

Fig. 10: General overview of the FEROX simulator

- **Robot Instance (Backend)**: The backend, running on an Ubuntu system with ROS, manages the robot agents in the simulation. The Unity instance treats the robotic units as NavMesh Agents, allowing them to navigate the environment autonomously and manage dynamic aspects of the simulation, such as spawning berries and updating their visibility based on UAV exploration data. The ROS-TCP Connector enables bidirectional communication, allowing robots to perform tasks like navigation and exploration through ROS services. This setup streamlines robot control, treating them like game characters with integrated path planning.
- **Human Instance (Frontend)**: The Human Instance provides each player with their own Unity instance, allowing them to control a 3D character (picker) in the virtual environment. These frontend instances communicate with the ROS network via the ROS-TCP Connector, using concepts from LAN and online multiplayer games to ensure real-time interaction and synchronization between player movements and the backend's robotic agents. Each human instance operates independently, creating a third-person multiplayer simulation. Players can request robotic services, such as UAV mapping or berry collection assistance, and interact with the environment by gathering berries and loading them onto robots [43].

The FEROX Simulator stands out by allowing users to control avatars similar to video game characters. A built-in scoring system awards points based on berry collection, fostering a "coopetitive" environment where users can collaborate or compete, exploring both cooperative and competitive dynamics in HRC scenarios. This feature, along with its focus on human interaction, distinguishes it from other simulators, which mainly focus on robotic perspectives, making it essential for HRC development.

4.1.2 Agents

In the FEROX project, the agents involved in the Human-Robot Collaboration (HRC) system are both humans and robots, with a focus on Unmanned Aerial Vehicles (UAVs). The human agents include berry pickers and a watcher, while the UAVs are categorized into two types: Light Weight Drones (LWD) and High Weight Drones

(HWD) (Fig. 11). Each type of UAV has specific roles and predefined tasks that support the overall berry collection process.

- **Human agents**
  - **Picker:** The picker is a human agent responsible for the manual collection of berries. Their role is to collaborate with the LWDs, which explore and map areas to identify berry locations. Once the exploration data is available, the picker harvests the berries. The picker also interacts with the HWDs to facilitate the transportation of harvested berries by loading them onto the drones, which then deliver the berries to a designated base station.
  - **Watcher:** The watcher persona oversees and validates the execution of key tasks involving both human and robotic agents. This agent plays a critical role in verifying the completion of specific steps, such as ensuring that a drone successfully meets a picker for a berry transfer during the `PickUp` task. The watcher confirms task milestones, ensuring smooth workflow and coordination between the picker and UAVs.
- **Robotic agents**
  - **Light Weight Drones (LWD):** LWDs are tasked with exploring and mapping predefined regions to assist pickers in locating berry-rich areas. Their primary role is reconnaissance, gathering spatial data that informs the picker's berry collection strategy. The LWDs operate autonomously but in direct support of the picker's objectives.
  - **High Weight Drones (HWD):** HWDs are responsible for logistical tasks such as the transportation of berries, emergency support, and assisting pickers with navigation or geofencing compliance. These drones can transport larger loads of harvested berries and provide on-demand assistance for tasks initiated by the picker or triggered by geofence breaches.

Both LWDs and HWDs operate using ROS action servers, which handle core functions such as `TakeOff`, `GoTo`, and `LandIn`. These fundamental actions allow the UAVs to autonomously manage their flight, navigation, and landing in response to specific commands. LWDs are additionally equipped with an `Explore` action server, enabling them to autonomously survey designated areas. The integration of these action servers allows the UAVs to perform their roles with a high degree of autonomy and precision.

Each UAV relies on a client-server architecture to execute these actions, where clients send specific commands or goals to the UAVs. The action servers then process these goals and provide real-time feedback, ensuring that the UAVs dynamically adjust their operations based on the current environment and conditions. This continuous feedback loop enhances the responsiveness and adaptability of the UAVs, allowing them to perform complex manoeuvres efficiently while remaining in sync with other agents in the system.

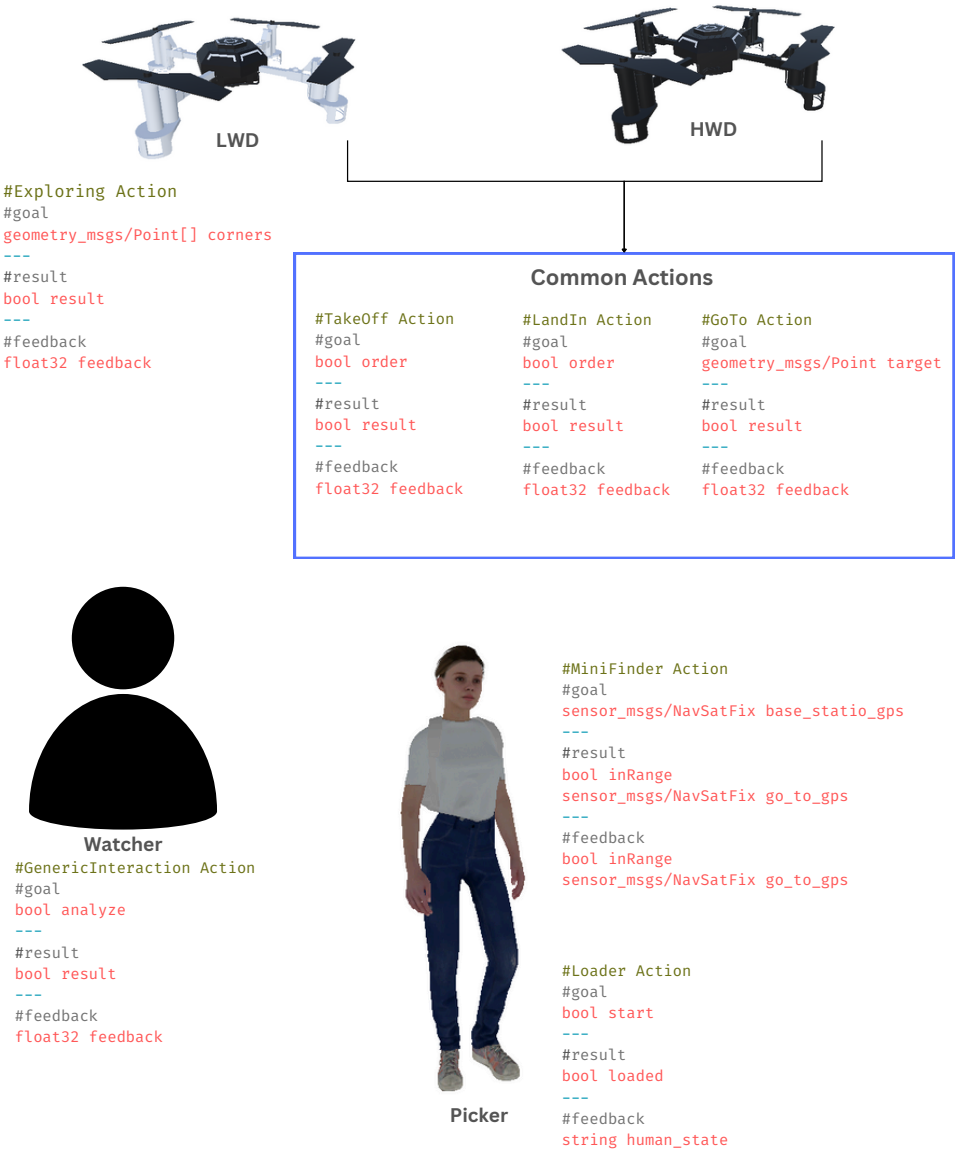


Fig. 11: General overview of FEROX agents.

4.2 Human-Mediated Operations

In the context of the FEROX system, human-mediated operations play a crucial role in enhancing the efficiency and effectiveness of drone-assisted forestry tasks. This section highlights the collaborative efforts between human operators and autonomous drones during two key phases: Reconnaissance and Picking. Each phase encompasses specific tasks that are essential for the successful execution of missions, ensuring that operators can leverage drone technology to gather data and optimise berry collection processes. The details of these tasks, along with their associated code, are outlined in Table 2 and provided in Appendix A.

Table 2: Task Descriptions for Human-Mediated Operations

Tasks	Description
MoveTask	A HWD or LWD goes back to the base station and lands there.
ExploreRegion	A LWD autonomously explores a designated region, collecting data to generate detailed semantic maps for forestry inventory.
PickUp	A HWD is dispatched to collect harvested berries from a human picker after receiving a request, subsequently transporting the payload back to the base station.
WatchDogSOS	Upon receiving an SOS alert (e.g., when a picker enters a geo-fenced area or faces an emergency), a HWD is sent to assess the situation, confirming the picker's safety before returning to the base station.
WatchDogRequest	A HWD assists a disoriented or lost picker by autonomously guiding them back to the base station, following a request for help.

4.2.1 Reconnaissance

The reconnaissance phase focuses on gathering essential forestry data through aerial scanning, preceding the berry-picking phase. LWDs execute the `ExploreRegion` task, mapping large areas to generate semantic maps for forestry inventory and berry yield predictions. Using UAV-aware segmentation principles, the Operator App segments the region into manageable areas and sends a ROS service request to FORMIGA for each segment, specifying the GNSS coordinates of the area's corners. (Fig. 12). FORMIGA processes the request, adds the task to the pending list, and, upon activation, selects the closest available LWD to execute the task. This approach ensures optimal coverage while considering drone constraints like battery life.

4.2.2 Picking

The Picking phase, following reconnaissance, is critical to berry collection, consisting of three tasks: `PickUp`, `WatchDogSOS`, and `WatchDogRequest`, which enhance drone-human collaboration for efficiency and safety (Fig. 13). The `PickUp` task involves a drone assisting a picker by collecting berries and returning to its port, triggered via a

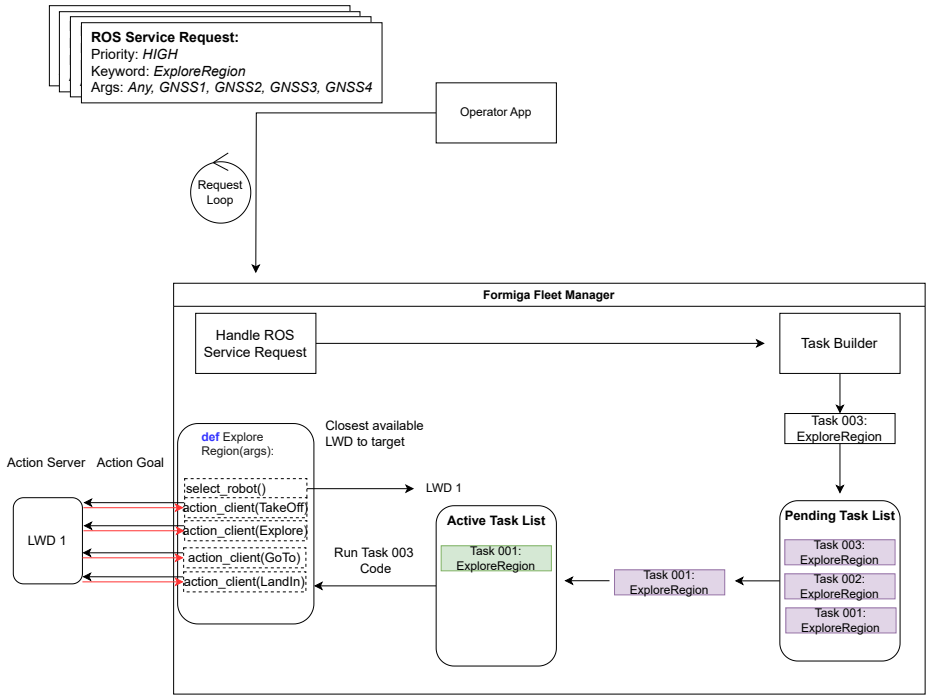


Fig. 12: An overview of FORMIGA `Exploration` task running flow.

third-party app, PickerApp - an intuitive tool for viewing berry maps and requesting drone support. `WatchDogSOS` is activated when a picker crosses a geofence or sends an emergency alert, dispatching a drone to ensure their safety. `WatchDogRequest` helps guide disoriented pickers back to base using real-time updates from a GNSS MiniFinder, ensuring safe navigation across large fields. Additionally, the `MoveTask` can be utilised by LWD and HWD to return to the base station when required.

5 Experiments

In this section, we outline the experimental setup and evaluation methods used to assess both the effectiveness of the FORMIGA Fleet Management System and the dynamic task generation capabilities of a fine-tuned LLM within the FEROX simulator.

5.1 Assessment of the FORMIGA Fleet Management System

The purpose of this experiment was to evaluate the efficiency and usability of the FORMIGA FMS by comparing two modes of operation: fully autonomous task execution, managed by FORMIGA, and semi-autonomous execution, where a human

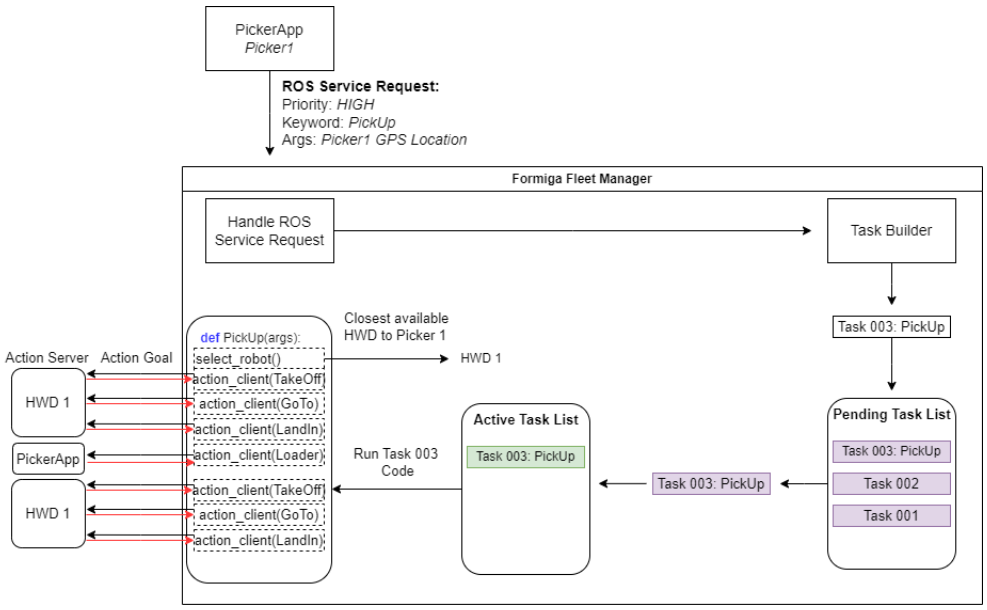


Fig. 13: An overview of FORMIGA PickUp task running flow.

operator manually triggered individual actions. As outlined earlier, FORMIGA automates the execution of tasks by selecting the most appropriate drones and executing the required actions according to predefined task codes. However, in semi-autonomous mode, the operator must manually trigger actions, which allows us to assess the cognitive load associated with completing a task without full automation.

For this experiment, we selected the `PickUp` task, executed on a HWD drone, and conducted 10 trials in both autonomous and semi-autonomous modes. The `PickUp` task is relatively simple, yet representative of repetitive field operations. We measured the time elapsed between each pair of consecutive actions (e.g., the time between the completion of the `TakeOff` and the `GoTo` actions) to assess the efficiency of task flow management in both modes. Notably, human interaction in this task occurs during the loading confirmation, where the FMS waits for the human (via a third-party app, as the PickerApp) to signal that the berries have been loaded onto the drone. The FMS autonomously handles the following steps (task code presented in Appendix A):

1. Selects the most suitable HWD drone based on its proximity to the target.
2. `TakeOff:` Initiates take-off.
3. `GoTo:` Navigates the drone to the target location, where the human (as determined by the PickerApp) is located.
4. `LandIn:` Lands at the specified target.
5. `Watcher:` Waits for the human to confirm that the loading process is complete.
6. `TakeOff:` Initiates take-off again.
7. `GoTo:` Returns the drone to its original port.
8. `LandIn:` Executes a final landing at the port.

In the semi-autonomous scenario, the human operator had to manually control each action, including selecting the drone, inputting target coordinates, and sending commands sequentially. This required waiting for the completion of one action before triggering the next. Additionally, the operator had to manually retrieve and enter the Cartesian coordinates for the `GoTo` command, introducing further cognitive load and operational delays. Once the drone landed at the target, the operator waited for the loading process and manually directed the drone back to its base.

To visualise the time sequence of a task triggered semi-autonomously versus autonomously, a variation of the classical box plot has been adopted. This variation, illustrated in Fig. 14, combines the temporal structure of a Gantt chart with the statistical features of a box plot, making it particularly valuable for assessing the execution time of actions within a task sequence. It effectively captures time dependencies and variability in action duration, offering a clear view of task structure and performance.

The comparison between the two modes revealed significant efficiency gains with the autonomous FMS. The system demonstrated a 37.21% reduction in overall execution time compared to semi-autonomous operation. This improvement is primarily due to FORMIGA's seamless transitions between actions, eliminating the idle time inherent in manual task management. The results underscore the FMS's capacity to optimise task flow and reduce human involvement, which is particularly valuable for repetitive, labour-intensive tasks, like berry picking. In realistic settings, these benefits compound over time, as the FMS autonomously manages parallel tasks across multiple agents (drones and humans), minimising delays and improving overall productivity.

In addition to time savings, the autonomous FMS also demonstrated improved consistency in task execution. The variability in task completion times, as shown by the interquartile range in our results, was significantly lower in the autonomous trials compared to the semi-autonomous ones. This reduction in variability highlights the system's ability to execute tasks with greater predictability, which is essential in operations requiring precise coordination between multiple agents. In a field scenario, such consistency can translate to better resource management and a smoother operational flow, ultimately enhancing overall system reliability.

By reducing the cognitive load on human operators and streamlining task execution, FORMIGA's autonomous task flow management not only improves time efficiency, but also creates a flexible system that allows occasional human intervention, aligning with the paradigm of human-mediated robot autonomy. This combination of human oversight with autonomous execution is critical to ensuring safety, acceptability, and scalability in complex ecological environments, such as those addressed in this study.

5.2 Evaluation of LLM-Generated Task Coding

This subsection details the evaluation process for testing the task generation capabilities of the fine-tuned LLM. The experiment involved two main stages: i) follow the methodology presented in Section 3.4, fine-tuning the model with a set of five predefined tasks, $T_{pre} = T1, T2, T3, T4, T5$ (see Appendix A); and ii) generating and evaluating five new tasks, $T_{LLM} = T6, T7, T8, T9, T10$, (with the ground truth provided in Appendix B). It is important to note that these five new tasks were not part

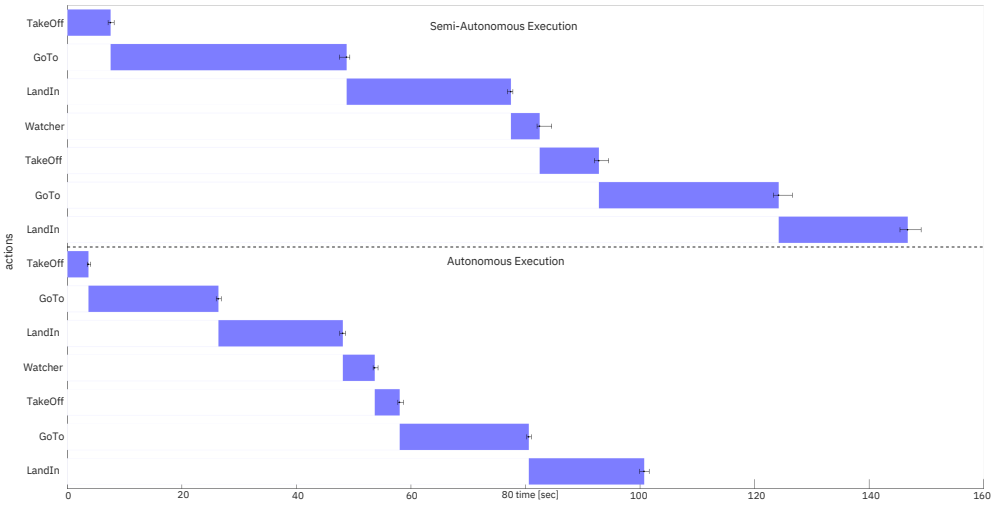


Fig. 14: Time Comparison of Autonomous and Semi-Autonomous Executions

of the original training data, being considerably more complex than these. Each of the generated tasks was produced 30 times, resulting in a total of 150 task variations. These tasks were then executed and evaluated within the FEROX simulator.

The five new tasks generated by the LLM for this experiment are as follows (Appendix B):

- `Circle` ($T6$): A LWD takes off and flies in a circular pattern with a 3-meter radius around the base station. This task tests the drone's ability to maintain consistent motion and path following around a central point.
- `Triangle` ($T7$): In this task, three LWDs take off and form a triangular formation. One drone is positioned directly above a given GNSS coordinate, representing the right-angle corner of the triangle. Once in formation, all three drones land together. This task evaluates coordination and precision in multi-drone formations.
- `ZigZag` ($T8$): A single LWD takes off and reaches its target by moving in a zigzag pattern. After completing the zigzag motion, the drone lands at the target location. The task focuses on path diversity and movement complexity.
- `Patrol and Escort` ($T9$): Two LWDs are involved in this task. The first drone takes off and lands at a given GNSS coordinate, which could be useful for scenarios where the drone escorts a human. Meanwhile, the second drone patrols by moving in a circular path within a defined radius around the first drone and lands asynchronously. This task is designed to test simultaneous and complementary behaviours in multi-drone missions.
- `Delivery` ($T10$): Two LWD drones take off, fly to a specified GNSS target, and land 5 meters apart. After confirmation from a watcher (for loading or unloading a package), the first drone returns to the base port and lands, while the second proceeds to a new destination and lands. Both drones operate asynchronously in this task, mimicking real-world delivery scenarios.

These tasks introduce a range of complexities, including multi-drone coordination, asynchronous operation, and varying movement patterns.

The evaluation metrics used are outlined below:

1. **Success Rate**: The Success Rate measures whether the generated code successfully performs the intended task within the simulator. A task is considered successful if the generated code achieves the desired outcome as specified. This metric helps determine the effectiveness of the LLM-generated code in executing the tasks as intended.
$$\text{Success Rate} = \left(\frac{\text{Number of Successful Tasks}}{\text{Total Number of Tasks}} \right) \times 100 \tag{1}$$
2. **CPU Time:** CPU Time assesses the computational efficiency of the code generation process. It is calculated as the time difference between the initiation of the code generation request and the display of the generated code. This metric provides insights into the responsiveness and performance of the LLM in generating code.
3. **CodeBLEU Metrics:** To evaluate the quality of the generated code, we utilised CodeBLEU [44], which extends traditional BLEU scoring to include syntactic and semantic aspects. The following CodeBLEU metrics were employed:
   (a) **Weighted N-Gram Match:**
   The Weighted N-Gram Match adapts the traditional BLEU metric by assigning different weights to n-grams based on their importance, particularly for programming languages. Unlike natural languages, which have a large vocabulary and flexible word order, programming languages are designed with specific keywords (such as "int" or "public"). These keywords play a critical role in the functionality of code. Traditional BLEU scores treat all n-grams equally, but this approach may overlook the significance of these keywords in code synthesis. To address this, Weighted N-Gram Match assigns higher weights to important keywords, improving the accuracy of code evaluation. In this metric, the weight for keywords is five times higher than for other tokens. The weighted precision is calculated by comparing the number of matching n-grams between the candidate and reference code, weighted by the importance of the n-grams:
   The weighted n-gram precision is computed as follows:
$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{i=1}^{l} \mu_i^n \cdot \text{Count}_{\text{clip}}(C(i, i+n))}{\sum_{C' \in \text{Candidates}} \sum_{i=1}^{l} \mu_i^n \cdot \text{Count}(C'(i, i+n))} \tag{2}$$
   where n represents the length of the n-gram, $C(i, i+n)$ denotes the n-gram from position i to position $i+n$, and $\text{Count}_{\text{clip}}(C(i, i+n))$ is the maximum number of n-grams co-occurring in a candidate code and a set of reference codes. The term μ_i^n represents the weights assigned to different keywords or n-grams. In this paper, the weight μ_i^n for keywords is set to be five times the weight assigned to other tokens.
   (b) **Syntactic AST Match**
   The Syntactic Abstract Syntax Tree (AST) Match assesses the structural accuracy of the generated code by comparing its AST with the reference code's AST. The AST represents the hierarchical structure of the code, with nodes denoting

syntactic constructs (e.g., loops, conditionals) and leaves representing variable and function names.

To evaluate this match, sub-trees are extracted from both the candidate and reference ASTs using a parser like tree-sitter. For our comparison, the leaf nodes are omitted, focusing instead on the structural elements.

The match score is calculated as:

$$\text{Match}_{\text{AST}} = \frac{\text{Count}_{\text{clip}}(T_{\text{cand}})}{\text{Count}(T_{\text{ref}})}$$

where Count(T_ref) is the number of reference sub-trees. Count_clip(T_cand) is the number of candidate sub-trees that match the reference.

(c) **Semantic Data-flow Match**

Semantic Data-flow Match evaluates the correctness of code based on variable dependencies. It uses data-flow graphs, where nodes represent variables and edges show the flow of data between them. This approach is more robust than syntactic metrics, as it captures how data is transferred and used in the code.

To compute the Semantic Data-flow Match score, the process involves several steps. First, data-flow graphs are constructed for both the candidate and reference code, where nodes represent variables and directed edges illustrate the flow of data between these variables. Next, data-flow items are normalised by standardising variable names, renaming them sequentially to ensure consistency in comparison. Finally, the match score is calculated using the formula:

$$\text{Match}_{\text{df}} = \frac{\text{Count}_{\text{clip}}(DF_{\text{cand}})}{\text{Count}(DF_{\text{ref}})}$$

where Count(DF_ref) is the total number of data-flow items in the reference, and Count_clip(DF_cand) is the number of candidate data-flow items that match the reference. This metric ensures that the generated code is evaluated for its semantic accuracy by comparing the data dependencies and logical flow.

Considering the aforementioned metrics, let T denote the set of tasks used in the experiments, where $T = \{T_1, \ldots, T_{10}\}$. Here, T_1 to T_5 represent the tasks used for pre-training the LLM, and T_6 to T_{10} correspond to the tasks generated by the LLM. Accuracy (as success rate) is presented in Table 3. CPU time and CodeBLEU metrics are illustrated in Figure 15 , with each box plot displaying the median as the central mark, and the 25^{th} and 75^{th} percentiles indicated by the bottom and top edges of the box, respectively.

Table 3

	T6	*T7*	*T8*	*T9*	*T10*	*T10+*
Acurracy	**93.33%**	**76.67%**	**73.33%**	**53.33%**	**53.33%**	**73.33%**

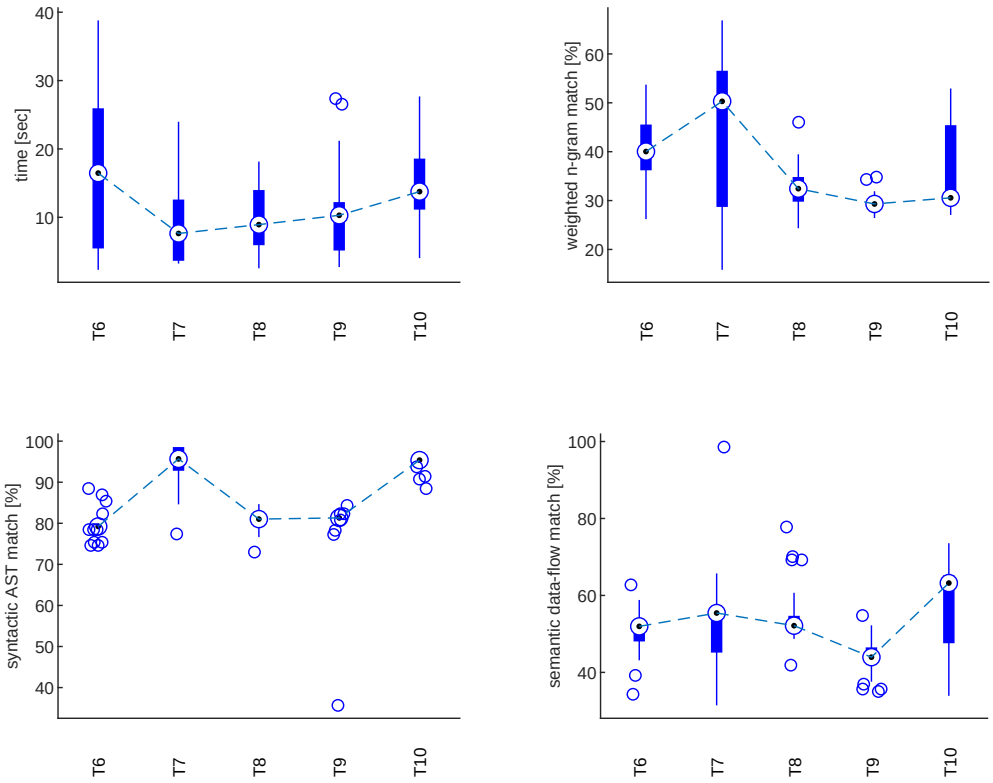


Fig. 15: Results of CodeBLEU metrics and CPU Time

The analysis of CPU time, accuracy, and CodeBLEU metrics reveals significant insights into the performance of LLM-generated code across varying task complexities. CPU Time shows that simpler tasks, like $T7$, have lower median CPU times (7.64 seconds), indicating efficient processing. In contrast, T6, with the highest median CPU time (16.47 seconds), achieves the highest accuracy (93.33%). This suggests that, while complex tasks may require more time, they can yield more accurate results if the generation process is thorough. However, $T10$, despite having a high median CPU time (13.76 seconds), shows lower accuracy (53.33%), indicating that increased processing time does not necessarily guarantee better task performance.

Weighted N-Gram Match scores reflect the model's capability to capture key programming constructs. $T7$ achieves the highest median score (0.503), suggesting strong performance for simpler tasks. In contrast, $T9$ and $T10$ show lower scores (0.293 and 0.305, respectively), correlating with their decreased accuracy. This decline highlights the model's difficulty in maintaining syntactic relevance for more complex tasks.

Syntactic AST Match scores are high across tasks, particularly for $T7$ (0.957) and $T10$ (0.954), indicating good alignment with syntactic structures. Despite this, accuracy drops for more complex tasks, showing that syntactic correctness alone is insufficient for functional success.

Semantic Data-Flow Match scores show that $T10$, with the highest median score (0.632), performs well in maintaining semantic consistency. However, the lower scores for T9 and T8, along with $T10$'s low accuracy, underscore that high semantic accuracy does not always translate into successful task execution.

Overall, while the LLM generates code that is often syntactically and semantically accurate, achieving high task performance—especially for complex tasks—requires more than just correctness in these dimensions. It underscores the need for the model to address functional requirements effectively. Therefore, aligned with this need for improved functionality, we have incorporated the ground truth code for the first four LLM-generated tasks, $T6, T7, T8, T9$, into the fine-tuning dataset. Following this, we tasked the LLM with generating $T10$ once again, now referred to as $T10+$, to clearly differentiate the model's performance before and after this additional fine-tuning. This process was repeated 30 times to determine whether including additional ground truth data from earlier tasks would improve the model's ability to generate the new task more effectively.

The rationale for this additional fine-tuning was to examine whether familiarising the LLM with similar tasks from the same domain could enhance its generalisation capabilities, resulting in more accurate and consistent code generation for subsequent tasks. Comparing the model's performance before ($T10$) and after ($T10+$) fine-tuning enables us to assess the impact of targeted retraining on refining the model's grasp of complex task structures and its ability to generate new, untrained tasks with greater precision.

The results of fine-tuning the LLM with additional ground truth task code demonstrated significant improvements in task performance (Fig. 16). The accuracy of $T10+$ increased from 53.33% to 73.33%, highlighting the effectiveness of fine-tuning in enhancing the model's ability to generate correct solutions for complex tasks. Concurrently, CPU time was significantly reduced from a median of 13.76 seconds for $T10$ to 4.33 seconds for $T10+$, indicating improved computational efficiency. The Weighted N-Gram Match score also improved, with the median rising from 0.306 to 0.408, suggesting better alignment with key programming constructs. Although the Syntactic AST Match score showed a modest increase and the Semantic Data-Flow Match score even slightly decreased, the overall enhancements in accuracy and efficiency underscore the value of incorporating relevant ground-truth data in the fine-tuning process. These findings confirm that targeted fine-tuning can effectively improve both the accuracy and efficiency of code generation for complex tasks.

6 Discussion and Initial Field Trials

This study introduces FORMIGA, a fleet management system tailored to managing and coordinating tasks within a heterogeneous team of humans and robots, specifically designed for challenging field operations. The methodology employed throughout the design and evaluation of FORMIGA relied heavily a ROS-based architecture, allowing for seamless communication between agents (humans and robots) and standardisation of task execution. One notable aspect of the system is its use of a LLM to automate the generation of robotic behaviours, thereby reducing the need for extensive manual

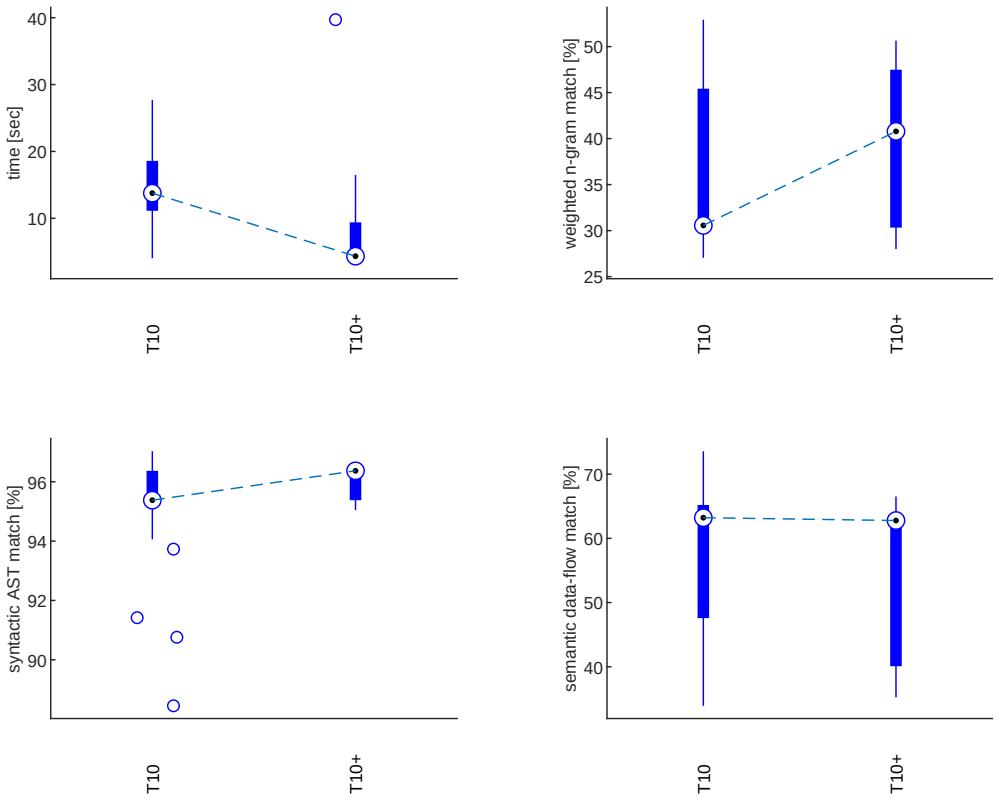


Fig. 16: Comparison of CodeBLEU metrics and CPU Time for the final task

coding. The LLM's integration into the FORMIGA ecosystem enhances its adaptability, making it highly suitable for environments that demand the generation of new tasks *in loco* to comply with unforeseen operational challenges. Although still under development, the system demonstrated significant potential, particularly in the management of complex tasks, such as multi-region exploration, human guidance, and emergency response.

The experimental results obtained using the FEROX simulator provided key insights into the functionality and scalability of FORMIGA. Simulations allowed us to assess the system's reliability in managing a fleet of drones and humans, executing collaborative tasks, such as `PickUp`, `WatchdogRequest` and `WatchdogSOS` (see Appendix A). One of the primary observations from these experiments was the system's flexibility in handling asynchronous task execution, integrating a wide range of simple actions, such as take-off, landing, and navigating to a desired pose. This flexibility was critical in ensuring that the system outperformed semi-autonomous operation, even in the hands of an experienced operator. Moreover, the results highlight the varied performance of LLM-generated code across tasks of different complexities. While simpler tasks achieved high accuracy with shorter CPU times, more complex tasks exhibited

lower accuracy despite longer processing times, indicating that increased computational time does not necessarily lead to better outcomes. However, fine-tuning the model improved accuracy by approximately 20% and reduced CPU time by about a third, demonstrating the effectiveness of incorporating additional ground-truth data to enhance both accuracy and efficiency for complex tasks.

Building upon the promising results obtained through simulations, preliminary real-world validation of the FORMIGA system was undertaken in September 2024, during field trials in Rovaniemi, Finland (Fig. 17). While the primary focus of this paper is the evaluation of FORMIGA within simulated environments, these field trials provided valuable initial insights into the practical challenges and potential of FORMIGA in real-world operations. Conducted as part of the FEROX project, the trials aimed to explore human-drone interaction technologies in berry picking operations, with FORMIGA playing a central role in managing and coordinating tasks in dynamic, unstructured environments.

Fig. 17: Preliminary real-world experiments executed in Rovaniemi (Finland). Left) Team managing the drones autonomously operating in the forest, acting as the watcher persona; Centre) Two HWD have been deployed to execute a series of tasks, including `PickUp`, `WatchdogRequest` and `WatchdogSOS` (Section 4.2); and Right) Smartphone app (PickerApp) used to support berry picking operations, including by requesting specific tasks to FORMIGA, as well as providing feedback to the pickers' ROS actions.

The field trials marked the first opportunity to test the `PickUp`, `WatchdogRequest`, and `WatchdogSOS` tasks in live environments, transitioning these tasks from simulation to reality. Although the scale and scope of these trials were limited, they demonstrated FORMIGA's capability to manage complex, real-time operations involving both drones and human workers. One of the main challenges encountered during the trials was managing communication over high-latency cellular networks. This necessitated external modifications, including the integration of a peer-to-peer virtual private network and a transition to a multi-master architecture to maintain efficient inter-robot communication. These changes fall outside the scope of FORMIGA's core system, and henceforth this paper, but were essential to enable its deployment in real-world conditions.

Despite these external adjustments, FORMIGA required minimal changes during the sim-to-real transition. New action clients were created to enable GNSS-guided navigation for the drones, and some safety protocols within the ROS actions were temporarily relaxed to accommodate the limitations of wireless communication in the

field. These modifications allowed FORMIGA to effectively coordinate tasks, such as the `PickUp` task, where drones autonomously collected berry buckets and returned them to the base station, reducing the physical burden on human workers. Similarly, the `WatchdogRequest` task successfully guided pickers back to safety, and the `WatchdogSOS` task enabled a rapid response to planned emergency situations. These use cases highlight the potential of FORMIGA in improving the safety and efficiency of labour-intensive operations, particularly in remote and hazardous environments.

It is important to emphasise that, while these initial field trials provided promising results, they were conducted on a small scale with limited participants and a reduced number of trials. As such, these trials should be viewed as a preliminary case study rather than a fully scaled validation of FORMIGA. Future work will include more extensive trials, involving larger groups of participants and additional robots, to rigorously test the system's scalability, reliability, and robustness in real-world scenarios. Nevertheless, this preliminary real-world assessment offered valuable feedback on FORMIGA's capacity to manage human-robot collaboration, and the lessons learned will guide the system's continued refinement and future deployment.

7 Conclusion

This work presented FORMIGA, a ROS-based fleet management system designed to coordinate humans and robots in unstructured environments, demonstrating its potential in enhancing task execution through autonomous decision-making. By integrating a LLM for automated task generation, FORMIGA reduces the need for manual coding and simplifies the coordination of multi-agent operations. Experimental results from simulations showcased its effectiveness in managing multiple human-robot tasks within the context of the FEROX EU Project, achieving significant time savings and reduced variability in task execution.

While only preliminarily evaluated in small scale real-world experiments, FORMIGA represents a crucial step toward effective human-robot collaboration, with promising implications for applications in agriculture, forestry, and other industries reliant on manual labour in remote settings. Future efforts will focus on scaling up the trials to assess FORMIGA's performance with larger teams and more complex scenarios, as well as integrating more end-user feedback to further refine the system.

Supplementary information. The FORMIGA framework, developed as part of our research, will be publicly available for academic and research purposes in the following link upon acceptance:

https://gitlab.ingeniarius.pt/ingeniarius_public/formiga

The framework, along with its documentation and supplementary files, can be accessed through our dedicated repository. This includes guides on implementation, usage, and customisation to facilitate further research and application in various fields of robotics. The repository is intended to support transparency, collaboration, and advancement within the research community by providing a comprehensive resource for exploring and building upon the FORMIGA framework.

Statements and Declarations

This work has been partly funded by the FEROX (https://ferox.fbk.eu/) EU project. FEROX has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreements No. 101070440. Views and opinions expressed are however those of the authors only and the European Commission is not responsible for any use that may be made of the information it contains. 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, http://www.ieeta.pt/).

Competing Interests

The authors declare that they have no competing interests related to the work submitted for publication.

Compliance with Ethical Standards

This study did not involve human participants or animals, and as such, no formal ethical approval was required.

References

[1] Gonzalez-de-Santos, P., Fernández, R., Sepúlveda, D., Navas, E., Emmi, L., Armada, M.: Field robots for intelligent farms—inhering features from industry. Agronomy **10**(11), 1638 (2020)

[2] Gombolay, M.C., Huang, C., Shah, J.: Coordination of human-robot teaming with human task preferences. In: 2015 AAAI Fall Symposium Series (2015)

[3] Hoffman, G., Breazeal, C.: Collaboration in human-robot teams. In: AIAA 1st Intelligent Systems Technical Conference, p. 6434 (2004)

[4] Sikand, K.S., Zartman, L., Rabiee, S., Biswas, J.: Robofleet: Open source communication and management for fleets of autonomous robots. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 406–412 (2021). IEEE

[5] Goldberg, K.: Robots and the return to collaborative intelligence. Nature Machine Intelligence **1**(1), 2–4 (2019)

[6] He, H., Gray, J., Cangelosi, A., Meng, Q., McGinnity, T.M., Mehnen, J.: The challenges and opportunities of human-centered ai for trustworthy robots and autonomous systems. IEEE Transactions on Cognitive and Developmental Systems **14**(4), 1398–1412 (2021)

[7] Michalec, O., O'Donovan, C., Sobhani, M.: What is robotics made of? the interdisciplinary politics of robotics research. Humanities and Social Sciences Communications **8**(1) (2021)

[8] Antunes, A., Jamone, L., Saponaro, G., Bernardino, A., Ventura, R.: From human instructions to robot actions: Formulation of goals, affordances and probabilistic planning. In: 2016 IEEE International Conference on Robotics and Automation (ICRA), pp. 5449–5454 (2016). IEEE

[9] Michalos, G., Spiliotopoulos, J., Makris, S., Chryssolouris, G.: A method for planning human robot shared tasks. CIRP journal of manufacturing science and technology **22**, 76–90 (2018)

[10] Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., Garg, A.: Progprompt: Generating situated robot task plans using large language models. In: 2023 IEEE International Conference on Robotics and Automation (ICRA), pp. 11523–11530 (2023). IEEE

[11] Hua, P., Liu, M., Macaluso, A., Wang, L., Lin, Y., Zhang, W., Xu, H., Wang, X.: Gensim2: Realistic robot task generation with llm. In: 8th Annual Conference on Robot Learning

[12] Ao, J., Wu, F., Wu, Y., Swikir, A., Haddadin, S.: Llm as bt-planner: Leveraging llms for behavior tree generation in robot task planning. arXiv preprint arXiv:2409.10444 (2024)

[13] Khamis, A., Hussein, A., Elmogy, A.: Multi-robot task allocation: A review of the state-of-the-art. Cooperative robots and sensor networks 2015, 31–51 (2015)

[14] Matarić, M.J., Sukhatme, G.S., Østergaard, E.H.: Multi-robot task allocation in uncertain environments. Autonomous Robots **14**, 255–263 (2003)

[15] Shiroma, P.M., Campos, M.F.: Comutar: A framework for multi-robot coordination and task allocation. In: 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 4817–4824 (2009). IEEE

[16] Zlot, R., Stentz, A.: Complex task allocation for multiple robots. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation, pp. 1515–1522 (2005). IEEE

[17] Zhang, K., Collins Jr, E.G., Shi, D.: Centralized and distributed task allocation in multi-robot teams via a stochastic clustering auction. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **7**(2), 1–22 (2012)

[18] Khamis, A.M., Elmogy, A.M., Karray, F.O.: Complex task allocation in mobile surveillance systems. Journal of Intelligent & Robotic Systems **64**, 33–55 (2011)

[19] Darrah, M., Niland, W., Stolarik, B.: Multiple uav dynamic task allocation using mixed integer linear programming in a sead mission. In: Infotech@ Aerospace, p. 7164 (2005)

[20] Yan, Z., Jouandeau, N., Cherif, A.A.: A survey and analysis of multi-robot coordination. International Journal of Advanced Robotic Systems **10**(12), 399 (2013)

[21] Vail, D., Veloso, M.: Multi-robot dynamic role assignment and coordination through shared potential fields. Multi-robot systems **2**, 87–98 (2003)

[22] Verma, J.K., Ranga, V.: Multi-robot coordination analysis, taxonomy, challenges and future scope. Journal of intelligent & robotic systems **102**, 1–36 (2021)

[23] Chakraa, H., Guérin, F., Leclercq, E., Lefebvre, D.: Optimization techniques for multi-robot task allocation problems: Review on the state-of-the-art. Robotics and Autonomous Systems, 104492 (2023)

[24] Shelkamy, M., Elias, C.M., Mahfouz, D.M., Shehata, O.M.: Comparative analysis of various optimization techniques for solving multi-robot task allocation problem. In: 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), pp. 538–543 (2020). IEEE

[25] Karlsson, M., Ygge, F., Andersson, A.: Market-based approaches to optimization. Computational Intelligence **23**(1), 92–109 (2007)

[26] Tolmidis, A.T., Petrou, L.: Multi-objective optimization for dynamic task allocation in a multi-robot system. Engineering Applications of Artificial Intelligence **26**(5-6), 1458–1468 (2013)

[27] Bolu, A., Korçak, Ö.: Adaptive task planning for multi-robot smart warehouse. IEEE Access **9**, 27346–27358 (2021)

[28] Couceiro, M.S., Machado, J.T., Rocha, R.P., Ferreira, N.M.: A fuzzified systematic adjustment of the robotic darwinian pso. Robotics and Autonomous Systems **60**(12), 1625–1639 (2012)

[29] Mina, T., Kannan, S.S., Jo, W., Min, B.-C.: Adaptive workload allocation for multi-human multi-robot teams for independent and homogeneous tasks. IEEE Access **8**, 152697–152712 (2020)

[30] Couceiro, M.S., Figueiredo, C.M., Rocha, R.P., Ferreira, N.M.: Darwinian swarm exploration under communication constraints: Initial deployment and fault-tolerance assessment. Robotics and Autonomous Systems **62**(4), 528–544 (2014)

[31] Prorok, A., Malencia, M., Carlone, L., Sukhatme, G.S., Sadler, B.M., Kumar,

V.: Beyond robustness: A taxonomy of approaches towards resilient multi-robot systems. arXiv preprint arXiv:2109.12343 (2021)

[32] Gil, S., Kumar, S., Mazumder, M., Katabi, D., Rus, D.: Guaranteeing spoof-resilient multi-robot networks. Autonomous Robots **41**, 1383–1400 (2017)

[33] InOrbit: Scaling robot fleets: The journey from 50 to 5000. White Paper (2021)

[34] Robots, M.: Industry insights: Robotics interoperability. White Paper (2021)

[35] Robots, M.: Industry insights: Interfleet software integration. White Paper (2022)

[36] Formant: Formant security. White Paper (2022)

[37] Formant: What is robotics as a service? a practical guide to raas. White Paper (2024)

[38] Formant: Agriculture: Scale fleet operations quickly and efficiently. White Paper (2024)

[39] Technologies of Vision (TeV) @ FBK: WildBe (Revision 010f5f0). Hugging Face (2024). https://doi.org/10.57967/hf/2550 . https://huggingface.co/datasets/FBK-TeV/WildBe

[40] 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: 2024 IEEE International Conference on Robotics and Automation (ICRA) (2024). IEEE

[41] Fletcher, S., Oostveen, A.M., Chippendale, P., Couceiro, M., Turtiainen, M., Ballester, L.S.: Developing unmanned aerial robotics to support wild berry harvesting in finland: Human factors, standards and ethics. In: Proc. in the International Conference Series on Robot Ethics and Standards (ICRES 2023), pp. 109–119 (2023). CLAWAR

[42] Yalçinkaya, A.C.M.S.S.F.S.P.V.A. Beril; Araújo: Unlocking the potential of human- robot synergy under advanced industrial applications: The ferox simulator. In: 2024 European Robotics Forum (ERF) (2024). ERF

[43] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., *et al.*: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3, p. 5 (2009). Kobe, Japan

[44] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)

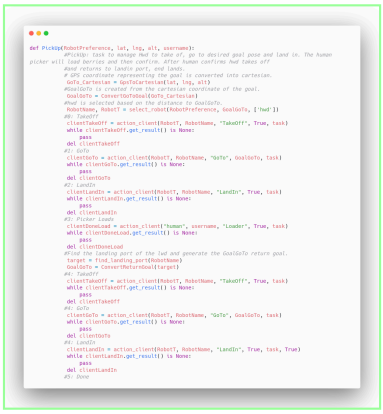
A Appendix A: User-defined tasks used for pre-training the LLM

```
def MoveTask(RobotPreference):
        #import necessary modules"
        #MoveTask: task manage to hwd/lwd to take off move around the base station following a line as 12 meter
away from the base station and then lands in.\n "
            import math
        #lwd/hwd is selected without a goal target.
         RobotName, RobotT = select_robot(RobotPreference, [], ['hwd', 'lwd'])
        #0:TakeOff
         take_off_client = action_client(RobotT, RobotName, "TakeOff", True, task)
         while take_off_client.get_result() is None:
            pass
         del take_off_client
        #1:GoTo around circle
         centre_gps = get_base_station()
         cartesian = GpsToCartesian(centre_gps.latitude, centre_gps.longitude, centre_gps.altitude)

         points_list = []
         points = []

         for i in range(4):

             x = cartesian.x + 3
             y = cartesian.y
             z = cartesian.z
             cart_coords = Cartesian_Coordinates()
             cart_coords.x = x
             cart_coords.y = y
             cart_coords.z = z
             Goal
         while land_client.get_result() is None:
            pass
         del land_client
        #3:Done
```



```
def PickUp(RobotPreference, lat, lng, alt, username):
            #PickUp: task to manage hwd to take off, go to desired goal pose and land in. The human
picker will then perform the loading operation. The human confirms hwd takes off
            # Hwd returns to landing port, and lands.
            # GPS coordinate representing the goal is converted into cartesian.
             GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
            #GoalGoTo is created from the cartesian coordinate of the goal.
             GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)
            #hwd is selected based on the distance to GoalGoTo.
             RobotName, RobotT = select_robot(RobotPreference, GoalGoTo, ['hwd'])
            #0: TakeOff
             clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
             while clientTakeOff.get_result() is None:
                pass
             del clientTakeOff
            #1: GoTo
             clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
             while clientGoTo.get_result() is None:
                pass
             del clientGoTo
            #2: LandIn
             clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task)
             while clientLandIn.get_result() is None:
                pass
             del clientLandIn
            #3: Picker Loads
             clientDoneLoad = action_client('human', username, "Loader", True, task)
             while clientDoneLoad.get_result() is None:
                pass
             del clientDoneLoad
            #Find the landing port of the lwd and generate the GoalGoTo return goal.
             target = find_landing_port(RobotName)
             GoalGoTo = ConvertReturnGoal(target)
            #4: TakeOff
             clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
             while clientTakeOff.get_result() is None:
                pass
             del clientTakeOff
            #5: GoTo
             clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
             while clientGoTo.get_result() is None:
                pass
             del clientGoTo
            #6: LandIn
             clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task, True)
             while clientLandIn.get_result() is None:
                pass
             del clientLandIn
            #7: Done
```

```
def ExploreRegion(RobotPreference, lat1, lng1, alt1, lat2, lng2, alt2, lat3, lng3, alt3, lat4, lng4,
alt4):
            #ExploreRegion: task to manage a lwd to take off, navigate to exploring area and explore, and
then return to its landing port and land in.
            # GPS coordinates representing the corners of the area are converted into cartesian.
             P1_Cartesian = GpsToCartesian(lat1, lng1, alt1)
             P2_Cartesian = GpsToCartesian(lat2, lng2, alt2)
             P3_Cartesian = GpsToCartesian(lat3, lng3, alt3)
             P4_Cartesian = GpsToCartesian(lat4, lng4, alt4)
            #Cartesian points are stored in the list.
             points_list = [P1_Cartesian, P2_Cartesian, P3_Cartesian, P4_Cartesian]
            #GoalExplore is created from the points_list.
             GoalExplore = ConvertPointsToJsonList(points_list)
            #Lwd is selected based on the distance to GoalExplore.
             RobotName, RobotT = select_robot(RobotPreference, GoalExplore, ['lwd'])
            #0: TakeOff
             clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
             while clientTakeOff.get_result() is None:
                pass
             del clientTakeOff
            #1: Explore
             clientExplore = action_client(RobotT, RobotName, "Explore", GoalExplore, task)
             while clientExplore.get_result() is None:
                pass
             del clientExplore
            #Find the landing port of the lwd and generate the GoalGoTo return goal.
             target = find_landing_port(RobotName)
             GoalGoTo = ConvertReturnGoal(target)
            #2: GoTo
             clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
             while clientGoTo.get_result() is None:
                pass
             del clientGoTo

            #3: LandIn
             clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task, True)
             while clientLandIn.get_result() is None:
                pass
             del clientLandIn
            #3: Done"
```

```
def WatchDogRequest(RobotPreference, alarmtype, lat, lng, username):
          #WatchDogRequest: task to manage hwd to take of, go to desired goal pose. The human watcher
will confirm. If human confirms as positive MiniFinder action client
          #is triggered. Hwd starts to guide human until human picker reaches the base station. After
hwd returns to landin port and lands in.
           alt = 150
          # GPS coordinate representing the goal is converted into cartesian.
           GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
          #GoalGoTo is created from the cartesian coordinate of the goal.
           GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)
          #hwd is selected based on the distance to GoalGoTo.
           RobotName, RobotT = select_robot(RobotPreference, GoalGoTo, ['hwd'])
          #0: TakeOff
           clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
           while clientTakeOff.get_result() is None:
               pass
           del clientTakeOff
          #1: GoTo
           clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
           while clientGoTo.get_result() is None:
               pass
           del clientGoTo
           base_gps = get_base_station()
          #2: Watcher's Confirmation
           clientWatch = generic_interaction_client()
           while clientWatch.get_result() is None:
               pass
           watcherResult = clientWatch.get_result().result
          #3: Guide human picker to base station
           if watcherResult == "positive":
                while True:
                    pickerStatus = action_client("human", username, "MiniFinder", base_gps, task)
                    while pickerStatus.get_result() is None:
                        pass
                    pickerStatusResult = pickerStatus.get_result()
                    del pickerStatus
                    if pickerStatusResult.inRange:
                        break
                    newGoTo_Cartesian = GpsToCartesian(pickerStatusResult.go_to_gps.latitude,
pickerStatusResult.go_to_gps.longitude, 146)
                    newGoalGoTo = ConvertGoToGoal(newGoTo_Cartesian)
                    clientGoTo = action_client(RobotT, RobotName, "GoTo", newGoalGoTo, task)
                    while clientGoTo.get_result() is None:
                        pass
                    del clientGoTo
                    time.sleep(15)
          #Find the landing port of the lwd and generate the GoalGoTo return goal.
               target = find_landing_port(RobotName)
               GoalGoTo = ConvertReturnGoal(target)
          #4: GoTo
               clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
               while clientGoTo.get_result() is None:
                   pass
               del clientGoTo
          #5: LandIn
               clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task, True)
               while clientLandIn.get_result() is None:
                   pass
               del clientLandIn
          #6: Done
```

```
def WatchDogSOS(RobotPreference, alarmtype, lat, lng):
          #WatchDogSOS: taks to manage hwd to go to desired goal. Human watcher confirms its arrival.
After confirmation hwd returns to landing port and lands in.
           alt = 150
          # GPS coordinate representing the goal is converted into cartesian.
           GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
          #GoalGoTo is created from the cartesian coordinate of the goal.
           GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)
          #hwd is selected based on the distance to GoalGoTo.
           RobotName, RobotT = select_robot(RobotPreference, GoalGoTo, ['hwd'])
          #0: TakeOff
           clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
           while clientTakeOff.get_result() is None:
               pass
          #1: GoTo
           clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
           while clientGoTo.get_result() is None:
               pass
           clientWatch = generic_interaction_client()
           while clientWatch.get_result() is None:
               pass
          #2: Watcher's Confirmation
           watcherResult = clientWatch.get_result().result
           if watcherResult == "positive":
               #Find the landing port of the lwd and generate the GoalGoTo return goal.
               target = find_landing_port(RobotName)
               GoalGoTo = ConvertReturnGoal(target)
              #3:GoTo
               clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
               while clientGoTo.get_result() is None:
                   pass
              #4:LandIn
               clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task, True)
               while clientLandIn.get_result() is None:
                   pass
          #5:Done
```

B Appendix B: User-defined tasks to benchmark the LLM-generated Tasks

```
def DeliverTask(RobotPreference, RobotPreference2, lat, lng, alt, lat2, lng2, alt2):
    GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
    GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)
    RobotName, RobotT = select_robot(RobotPreference, GoalGoTo, ['hwd'])

    clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
    while clientTakeOff.get_result() is None:
        pass
    del clientTakeOff
    RobotName2, RobotT2 = select_robot(RobotPreference2, GoalGoTo, ['hwd'])
    clientTakeOff = action_client(RobotT2, RobotName2, "TakeOff", True, task)
    while clientTakeOff.get_result() is None:
        pass
    del clientTakeOff
    def robot1_task():
        # Robot 1 takes off and goes to the given GPS point



        clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
        while clientGoTo.get_result() is None:
            pass
        del clientGoTo

        clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task)
        while clientLandIn.get_result() is None:
            pass
        del clientLandIn

    def robot2_task():
        # Robot 2 takes off and goes to 1 meter away from the given GPS point
        GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
        GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)


        # Adjust the target to land 1 meter away
        GoTo_Cartesian.x =  GoTo_Cartesian.x + 1
        GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)

        clientGoTo = action_client(RobotT2, RobotName2, "GoTo", GoalGoTo, task)
        while clientGoTo.get_result() is None:
            pass
        del clientGoTo

        clientLandIn = action_client(RobotT2, RobotName2, "LandIn", True, task)
        while clientLandIn.get_result() is None:
            pass
        del clientLandIn

    # Create and start threads for Robot 1 and Robot 2 tasks
    robot1_thread = threading.Thread(target=robot1_task)
    robot2_thread = threading.Thread(target=robot2_task)

    robot1_thread.start()
    robot2_thread.start()

    # Wait for both threads (Robot 1 and Robot 2) to finish
    robot1_thread.join()
    robot2_thread.join()

    # Watcher confirms once both robots have landed
    clientWatch = generic_interaction_client()
    while clientWatch.get_result() is None:
        pass
    watcherResult = clientWatch.get_result().result

    if watcherResult == "positive":
        def robot1_return_task():
            # Robot 1 goes back to its port and lands
            clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
            while clientTakeOff.get_result() is None:
                pass
            del clientTakeOff

            target = find_landing_port(RobotName)
            GoalGoTo = ConvertReturnGoal(target)

            clientGoTo = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
            while clientGoTo.get_result() is None:
                pass
            del clientGoTo

            clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task)
            while clientLandIn.get_result() is None:
                pass
            del clientLandIn

        def robot2_second_task():
            # Robot 2 goes to the second GPS point and back to port
            clientTakeOff = action_client(RobotT2, RobotName2, "TakeOff", True, task)
            while clientTakeOff.get_result() is None:
                pass
            del clientTakeOff

            GoTo_Cartesian = GpsToCartesian(lat2, lng2, alt2)
            GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)

            clientGoTo = action_client(RobotT2, RobotName2, "GoTo", GoalGoTo, task)
            while clientGoTo.get_result() is None:
                pass
            del clientGoTo

            clientLandIn = action_client(RobotT2, RobotName2, "LandIn", True, task)
            while clientLandIn.get_result() is None:
                pass
            del clientLandIn

            # After landing at the second point, Robot 2 returns to its port
            target = find_landing_port(RobotName2)
            GoalGoTo = ConvertReturnGoal(target)

            clientTakeOff = action_client(RobotT2, RobotName2, "TakeOff", True, task)
            while clientTakeOff.get_result() is None:
                pass
            del clientTakeOff

            clientGoTo = action_client(RobotT2, RobotName2, "GoTo", GoalGoTo, task)
            while clientGoTo.get_result() is None:
                pass
            del clientGoTo

            clientLandIn = action_client(RobotT2, RobotName2, "LandIn", True, task, True)
            while clientLandIn.get_result() is None:
                pass
            del clientLandIn

        # Create and start threads for Robot 1 returning and Robot 2 going to the second point
        robot1_return_thread = threading.Thread(target=robot1_return_task)
        robot2_second_thread = threading.Thread(target=robot2_second_task)

        robot1_return_thread.start()
        robot2_second_thread.start()

        # Wait for both threads to finish
        robot1_return_thread.join()
        robot2_second_thread.join()
```

```
def Triangletask(RobotPreference1, RobotPreference2, RobotPreference3, lat, lng, alt):
    # Convert GPS coordinates to cartesian coordinates
    center_cartesian = GpsToCartesian(lat, lng, alt)

    # Calculate the three corner points of the triangle, 10 meters away from each other
    p1_cartesian = Cartesian_Coordiantes()
    p1_cartesian.x = center_cartesian.x
    p1_cartesian.y = center_cartesian.y
    p1_cartesian.z = center_cartesian.z

    p2_cartesian = Cartesian_Coordiantes()
    p2_cartesian.x = center_cartesian.x + 5
    p2_cartesian.y = center_cartesian.y
    p2_cartesian.z = center_cartesian.z

    p3_cartesian = Cartesian_Coordiantes()
    p3_cartesian.x = center_cartesian.x
    p3_cartesian.y = center_cartesian.y + 5
    p3_cartesian.z = center_cartesian.z

    # Select three different UAV robots based on the distance to the goal points
    RobotName1, RobotT1 = select_robot(RobotPreference1, p1_cartesian, ['hwd'])
    RobotName2, RobotT2 = select_robot(RobotPreference2, p2_cartesian, ['hwd'])
    RobotName3, RobotT3 = select_robot(RobotPreference3, p3_cartesian, ['hwd'])

    # 0: TakeOff for each robot
    clientTakeOff1 = action_client(RobotT1, RobotName1, "TakeOff", True, task)
    clientTakeOff2 = action_client(RobotT2, RobotName2, "TakeOff", True, task)
    clientTakeOff3 = action_client(RobotT3, RobotName3, "TakeOff", True, task)

    while clientTakeOff1.get_result() is None:
        pass
    del clientTakeOff1

    while clientTakeOff2.get_result() is None:
        pass
    del clientTakeOff2

    while clientTakeOff3.get_result() is None:
        pass
    del clientTakeOff3

    # 1: GoTo for each robot
    GoalGoTo1 = ConvertGoToGoal(p1_cartesian)
    GoalGoTo2 = ConvertGoToGoal(p2_cartesian)
    GoalGoTo3 = ConvertGoToGoal(p3_cartesian)

    clientGoTo1 = action_client(RobotT1, RobotName1, "GoTo", GoalGoTo1, task)
    clientGoTo2 = action_client(RobotT2, RobotName2, "GoTo", GoalGoTo2, task)
    clientGoTo3 = action_client(RobotT3, RobotName3, "GoTo", GoalGoTo3, task)

    while clientGoTo1.get_result() is None:
        pass
    del clientGoTo1

    while clientGoTo2.get_result() is None:
        pass
    del clientGoTo2

    while clientGoTo3.get_result() is None:
        pass
    del clientGoTo3

    # 2: LandIn for each robot
    clientLandIn1 = action_client(RobotT1, RobotName1, "LandIn", True, task, True)
    clientLandIn2 = action_client(RobotT2, RobotName2, "LandIn", True, task, True)
    clientLandIn3 = action_client(RobotT3, RobotName3, "LandIn", True, task, True)

    while clientLandIn1.get_result() is None:
        pass
    del clientLandIn1

    while clientLandIn2.get_result() is None:
        pass
    del clientLandIn2

    while clientLandIn3.get_result() is None:
        pass
    del clientLandIn3
```

```
def CircleTask(RobotPreference):
#Fwd is selected without a goal target.
RobotName, RobotT = select_robot(RobotPreference, [], ['lwd'])

#0: TakeOff
take_off_client = action_client(RobotT, RobotName, "TakeOff", True, task)
while take_off_client.get_result() is None:
pass
del take_off_client

#1: GoTo base station
base_station_gps = get_base_station()
base_station_Cartesian = GpsToCartesian(base_station_gps.latitude, base_station_gps.longitude,
base_station_gps.altitude)
GoalGoTo = ConvertGoToGoal(base_station_Cartesian)
go_to_client = action_client(RobotT, RobotName, "GoTo", GoalGoTo, task)
while go_to_client.get_result() is None:
pass
del go_to_client

#2: Circle around base station
points_list = []
for i in range(36): # 36 points to form a circle
x = base_station_Cartesian.x + 5 * math.cos(i * math.pi / 18) # radius of 5 meters
y = base_station_Cartesian.y + 5 * math.sin(i * math.pi / 18)
z = base_station_Cartesian.z
cart_coords = Cartesian_Coordiantes()
cart_coords.x = x
cart_coords.y = y
cart_coords.z = z
GoalGoTo = ConvertGoToGoal(cart_coords)
points_list.append(GoalGoTo)

for point in points_list:
circle_client = action_client(RobotT, RobotName, "GoTo", point, task)
while circle_client.get_result() is None:
pass
del circle_client

#3: LandIn
land_client = action_client(RobotT, RobotName, "LandIn", True, task, True)
while land_client.get_result() is None:
pass
del land_client

#4: Done
```

```python
def PatrolandEscort(RobotPreference1, RobotPreference2, lat, lng, alt):
            # Convert GPS coordinate to Cartesian
            GoTo_Cartesian = GpsToCartesian(lat, lng, alt)
            GoalGoTo = ConvertGoToGoal(GoTo_Cartesian)
            # Select two robots based on their preferences
            RobotName1, RobotT1 = select_robot(RobotPreference1, GoalGoTo, ['lwd', 'hwd'])
            RobotName2, RobotT2 = select_robot(RobotPreference2, GoalGoTo, ['lwd', 'hwd'])
            # Define the task for the first drone
            def task1():
                # 0: TakeOff
                clientTakeOff = action_client(RobotT1, RobotName1, "TakeOff", True, task)
                while clientTakeOff.get_result() is None:
                    pass
                del clientTakeOff
                # 1: GoTo
                clientGoTo = action_client(RobotT1, RobotName1, "GoTo", GoalGoTo, task)
                while clientGoTo.get_result() is None:
                    pass
                del clientGoTo
                # 2: LandIn
                clientLandIn = action_client(RobotT1, RobotName1, "TakeOff", True, task)
                while clientLandIn.get_result() is None:
                    pass
                del clientLandIn
            # Define the task for the second drone
            def task2():
                # 0: TakeOff
                clientTakeOff = action_client(RobotT2, RobotName2, "TakeOff", True, task)
                while clientTakeOff.get_result() is None:
                    pass
                del clientTakeOff
                # 1: GoTo around circle
                centre_gps = GoTo_Cartesian
                points_list = []
                for i in range(36):
                    x = centre_gps.x + 10 * math.sin(i * math.pi / 18)
                    y = centre_gps.y + 10 * math.cos(i * math.pi / 18)
                    z = centre_gps.z
                    cart_coords = Cartesian_Coordinates()
                    cart_coords.x = x
                    cart_coords.y = y
                    cart_coords.z = z
                    GoalGoTo = ConvertGoToGoal(cart_coords)
                    points_list.append(GoalGoTo)
                for i in points_list:
                    circle_client = action_client(RobotT2, RobotName2, "GoTo", i, task)
                    while circle_client.get_result() is None:
                        pass
                    del circle_client
                # 2: LandIn
                clientLandIn = action_client(RobotT2, RobotName2, "TakeOff", True, task, True)
                while clientLandIn.get_result() is None:
                    pass
                del clientLandIn
            # Create and start two threads for the tasks
            t1 = threading.Thread(target=task1)
            t2 = threading.Thread(target=task2)
            t1.start()
            t2.start()
            # Wait for both threads to finish
            t1.join()
            t2.join()
```

```python
def ZigZagTask(RobotPreference, start_lat, start_lng, start_alt, final_lat, final_lng, final_alt):
    # Task for a robot to take off, move to a starting GPS point, then move to a final point in a
zigzag pattern, and land.

    # Convert the start and final GPS coordinates to cartesian.
    Start_Cartesian = GpsToCartesian(start_lat, start_lng, start_alt)
    Final_Cartesian = GpsToCartesian(final_lat, final_lng, final_alt)

    # Select the robot based on the preference.
    RobotName, RobotT = select_robot(RobotPreference, Start_Cartesian, ['lwd'])

    # 0: TakeOff
    clientTakeOff = action_client(RobotT, RobotName, "TakeOff", True, task)
    while clientTakeOff.get_result() is None:
        pass
    del clientTakeOff

    # 1: Move to the starting point
    GoalStart = ConvertGoToGoal(Start_Cartesian)
    clientGoToStart = action_client(RobotT, RobotName, "GoTo", GoalStart, task)
    while clientGoToStart.get_result() is None:
        pass
    del clientGoToStart

    # Calculate the zigzag pattern waypoints from the starting point to the final point
    zigzag_points = []
    steps = 10  # Define the number of zigzag steps
    for i in range(steps):
        # Linear interpolation between the start and final points
        x = Start_Cartesian.x + i * (Final_Cartesian.x - Start_Cartesian.x) / steps
        y = Start_Cartesian.y + i * (Final_Cartesian.y - Start_Cartesian.y) / steps + (-1) ** i * 1  #
Zigzag by 0.5 meters
        point_cart = Cartesian_Coordinates()
        point_cart.x = x
        point_cart.y = y
        point_cart.z = Start_Cartesian.z  # Keep altitude constant for simplicity
        zigzag_points.append(point_cart)

    # 2: Move from the starting point to the final point in a zigzag pattern
    for point in zigzag_points:
        GoalZigzag = ConvertGoToGoal(point)
        clientZigzag = action_client(RobotT, RobotName, "GoTo", GoalZigzag, task)
        while clientZigzag.get_result() is None:
            pass
        del clientZigzag

    # 3: Land at the final point
    GoalFinal = ConvertGoToGoal(Final_Cartesian)
    clientLandIn = action_client(RobotT, RobotName, "LandIn", True, task, True)
    while clientLandIn.get_result() is None:
        pass
    del clientLandIn
```