ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCHOOL OF SCIENCE

Master degree in Computer Science

AUTONOMOUS CONTROL OF DRONE SWARMS USING AI AGENTS: DESIGN AND EXPERIMENTATION

Supervisor:
Prof. Marco Di Felice

Presented by:
Andrea Iannoli

Co-supervisors:
Prof. Angelo Trotta
Dr. Lorenzo Gigli

Session IV
Academic Year 2024/2025

Technology is neither good nor bad
nor is it neutral

Abstract

Large Language Models (LLMs) are being increasingly investigated as high-level reasoning components for cyber-physical systems, but applying them to real-time UAV swarm management is still difficult because of heterogeneous interfaces, weak grounding, and the need for persistent, closed-loop operation.

This thesis introduces a mission-independent, agent-augmented LLM framework for UAV swarm control in which users specify objectives in natural language and the system autonomously carries them out through grounded, real-time interactions. The architecture integrates an LLM-driven Agent Core with a Model Context Protocol (MCP) gateway and a Web-of-Drones layer built on W3C Web of Things (WoT) standards. By representing drones, sensors, and services as standardized WoT Things, the framework supports structured tool-based interaction, continuous state monitoring, and safe actuation without depending on code generation.

We assess the framework in an ArduPilot-based simulation across four swarm missions and six state-of-the-art LLMs. The results indicate that, although modern general-purpose LLMs exhibit strong reasoning capabilities, they still have difficulty delivering dependable execution, even on relatively simple swarm tasks, when explicit grounding and execution support are absent. Incorporating task-specific planning tools and runtime safeguards markedly increases robustness, and token usage by itself does not correlate with execution quality or reliability.

Overall, the findings underscore both the promise and the present constraints of LLM-driven swarm control, showing that agent-enhanced execution and standardized device abstractions are key to turning natural-language intent into reliable swarm behavior.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is the outcome of a research project that has led to an ongoing publication effort, reflected in the following paper [9] for the 27th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM). The following sections first present the background needed to understand the fundamental concepts underlying this work, and then introduce the research challenges addressed and the contributions of the thesis to the state of the art.

1.1 Context

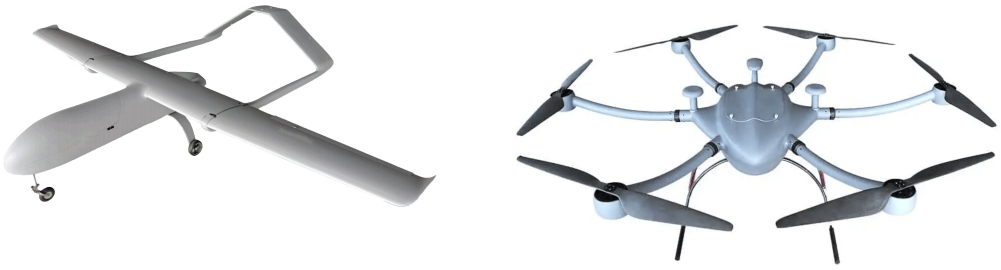
1.1.1 UAV

UAV stands for *Unmanned Aerial Vehicle* and is commonly used to indicate a drone, i.e., an aircraft that operates without a pilot on board.
Two main categories of UAVs can be distinguished:

1. *Fixed-wing UAVs:* These platforms rely on aerodynamic lift generated by wings and are generally not suitable for stationary operations. In particular, fixed-wing UAVs cannot hover and are therefore less appropriate for close-range inspection tasks. They are typically more energy-efficient during forward flight, since energy is primarily spent to

maintain motion in a given direction rather than to remain in place. As a result, fixed-wing UAVs can travel long distances and cover large areas. In many cases, they can sustain long endurance flights (up to several hours) because they may employ internal combustion engines and propellers rather than purely electric propulsion.

2. *Rotary-wing UAVs:* This category includes quadcopters, hexacopters, tricopters, and helicopters. Rotary-wing UAVs are commonly used for applications that require hovering and close observation of the environment, such as ground monitoring, surveillance, and inspection. Compared with fixed-wing platforms, rotary-wing UAVs typically have lower speed and limited payload capacity, but they can maintain a stationary position in the air and perform close-up inspection tasks.



(a) (b)

Figure 1.1: (a) fixed-wing UAV and (b) rotary-wing UAV.

Common UAV payloads can include: (i) *RGB cameras*, typically used for visual inspection, mapping, object detection, and photogrammetric reconstruction; (ii) *multispectral sensors/cameras*, used to derive vegetation indices, monitor crop vigor, and detect stress patterns not visible in standard imagery; (iii) *hyperspectral sensors*, which capture many narrow spectral

bands and are therefore suited to fine-grained material discrimination, early disease detection in agriculture, and advanced environmental characterization; (iv) *thermal infrared sensors/cameras*, used to detect heat anomalies, support search-and-rescue in low-visibility conditions, identify irrigation inefficiencies, and inspect infrastructure for overheating components; and (v) *LiDAR* payloads, used to obtain accurate 3D geometry, terrain elevation models, canopy structure estimates, and obstacle-aware navigation cues in complex environments.

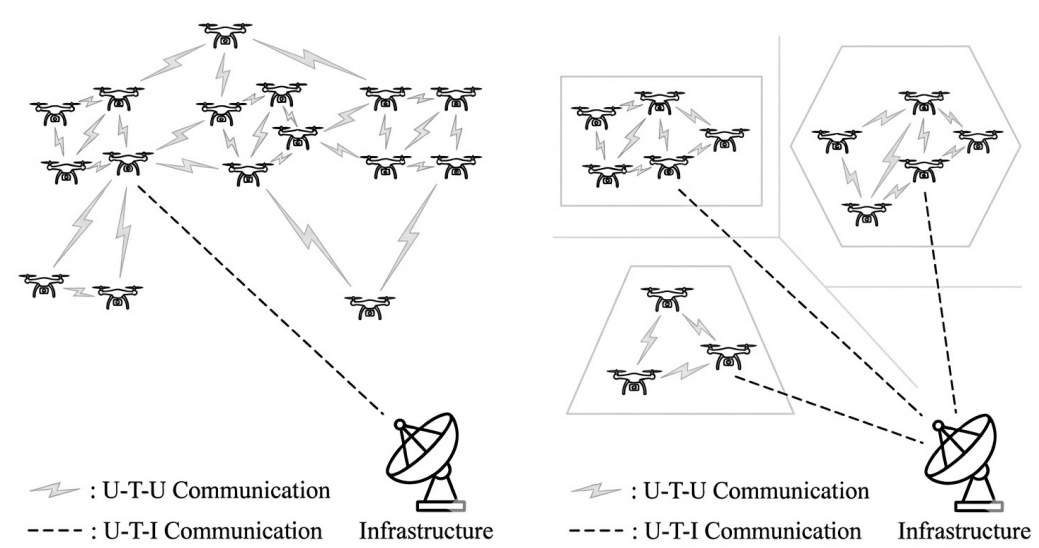
UAV type and payload selection is application-dependent, since mission value is largely determined by onboard sensing capabilities and operational constraints.

A UAV is composed of several fundamental subsystems that jointly enable flight and mission execution. Among these, the communication subsystem is particularly important, as it allows UAVs and their operators to coordinate actions and to exchange telemetry and mission data. Without reliable communication, unmanned flight would be severely constrained and the collection and transmission of aerial sensing and connectivity data would be impractical. Civilian UAV communication systems commonly operate in the 2.4 GHz and 5.8 GHz frequency bands.

1.1.2 Swarms

The concept of a UAV swarm refers to a group of UAVs that cooperatively performs a mission in a self-organized manner. Current trends in UAV development indicate increasing reliance on decentralized communication, where UAVs establish and maintain connectivity without a centralized controller. Common decentralized swarm architectures include Single-Group Swarm Ad hoc Networks (SGSAN), Multi-Group Swarm Ad hoc Networks (MGSAN), and Multi-Layer Swarm Ad hoc Networks (MLSAN).

Communication within UAV swarms is evolving toward broader integration with cloud-based and satellite-based infrastructures, enabling large-scale data exchange and coordination. Current research trends increasingly em-



(a) (b)

Figure 1.2: (a) Single-group UAV swarm and (b) Multi-group UAV swarm.

phasize the security of UAV swarm communications, given the critical role of connectivity in reliable swarm operation.

UAV swarms are now being adopted in a steadily expanding set of domains, including precision agriculture [21], environmental monitoring [17] search-and-rescue operations [1], and infrastructure inspection [12].

In precision agriculture, UAV swarms are used to support the full crop-management cycle rather than a single isolated task. Multi-drone teams can rapidly survey large fields to generate high-resolution orthomosaics, vegetation-index maps, and thermal maps that reveal early stress signals, irrigation anomalies, pest outbreaks, and nutrient deficiencies. Beyond monitoring, swarms can be configured for targeted intervention: selected drones may execute variable-rate spraying over localized zones identified as critical, while companion drones continue mapping and progress verification. This coordinated workflow enables farmers to reduce blanket chemical usage, shorten response time to emerging issues, and adapt treatment plans as field conditions change during the same mission window.

In environmental monitoring, swarms are valuable because they com-

bine spatial coverage, temporal continuity, and sensing diversity. Multiple UAVs can concurrently sample different regions of forests, rivers, coastlines, or urban ecosystems, collecting visual, multispectral, thermal, or atmospheric data to track vegetation health, water quality proxies, pollution plumes, erosion patterns, and habitat changes. In disaster-prone areas, they can also provide recurrent situational snapshots that support early warning and post-event impact assessment. Compared with single-drone operations, swarm deployments improve revisit frequency and robustness to individual-unit failures, making them suitable for long-duration monitoring campaigns where data timeliness and redundancy are critical.

For search-and-rescue operations, UAV swarms are used to accelerate victim detection and improve operational safety under time-critical constraints. Teams of drones can partition a search area, apply coordinated sweep patterns, and use complementary sensing modalities (e.g., RGB, thermal, and low-light imaging) to identify people, hazards, or access routes in difficult terrain. Some units can maintain persistent overwatch and communication relay functions while others perform close-range inspection, allowing responders to retain a global operational picture while refining local decisions. This division of labor is particularly useful in scenarios such as wildfire fronts, collapsed infrastructure, flood zones, or mountainous environments where ground access is limited and rapid triage can directly affect survival outcomes.

In infrastructure inspection, UAV swarms enable scalable and repeatable assessment of large distributed assets, including bridges, power lines, rail corridors, pipelines, and industrial plants. A coordinated fleet can inspect multiple segments in parallel, reducing downtime and minimizing the need for prolonged manual access to hazardous locations. Swarm-based inspections can combine coarse global scans with focused close-up passes over suspect areas, supporting crack detection, corrosion tracking, geometric deformation analysis, and anomaly localization. Repeated missions over time also allow temporal comparison, so maintenance teams can move from reactive interventions to condition-based planning informed by consistent, high-frequency

data acquisition.

Despite this broad uptake, orchestrating multiple drones, often heterogeneous in terms of hardware, sensing payloads, and onboard capabilities, is still a difficult problem, especially when missions unfold in environments that are dynamic, uncertain, and only partially observable. Effective coordination typically requires reasoning about many interacting entities, aligning the capabilities of each platform with the operational requirements of the mission, and continuously adapting decisions as conditions evolve in real time [29]. In many existing deployments, swarm management is achieved through tightly coupled, application-specific solutions: these solutions usually demand significant engineering effort, and they often remain difficult to reuse or extend when the target platforms change or when the mission profile must be modified.

1.1.3 LLM

Language is a fundamental tool for human expression and communication. We begin acquiring it early in life and rely on it in many different ways throughout our lifetime. Machines, however, do not naturally understand or produce human language without the support of sophisticated Artificial Intelligence (AI). For this reason, enabling machines to read, write, and communicate in a human-like manner has long been a central scientific goal. In recent years, progress in deep learning, the availability of large-scale computational resources, and access to massive training datasets have jointly enabled the rise of Large Language Models (LLMs).

LLMs have emerged as high-level reasoning systems that can interpret natural-language instructions, decompose complex objectives into smaller steps, and coordinate multi-stage workflows. As a result, their adoption in cyber-physical and robotic systems has attracted increasing attention, since they may provide a bridge between high-level human intent and low-level actuation [30, 28]. At the same time, their effectiveness and practical usability in these settings are still not fully understood, especially when systems must

interact with the physical world under real-time constraints.

Tokenization

A key preliminary step in LLM processing is *tokenization*, a preprocessing procedure that converts raw text into a sequence of discrete units called *tokens*. Depending on the model and the tokenization scheme, tokens can correspond to characters, subwords, symbols, or whole words. This representation is essential because the model operates on numerical token identifiers rather than directly on raw text, enabling efficient training and inference over large text datasets.

What Is a Transformer?

Most modern LLMs are based on the *Transformer* architecture, a neural network design introduced to model sequences effectively while supporting parallel computation. Rather than processing tokens strictly one at a time, Transformers represent an input sequence as a set of token embeddings and update these representations through repeated layers that learn relationships among tokens.

At a high level, a Transformer consists of three main components. First, tokens are mapped to continuous vectors through an *embedding* layer. Since the model must also account for token order, these embeddings are combined with *positional* information (often via positional encodings or learned positional embeddings). Second, the core of the architecture is a stack of Transformer layers. Each layer typically includes an *attention* sub-layer, which allows each token to incorporate information from other tokens in the sequence, followed by a *feed-forward* sub-layer, which applies non-linear transformations independently to each token representation. Third, additional architectural elements, such as residual connections and normalization, are commonly used within each layer to stabilize training and improve gradient flow.

Depending on the task and model family, Transformers can be organized

as encoder-only, decoder-only, or encoder-decoder architectures. Many LLMs used for text generation adopt a *decoder-only* structure, where predictions are produced autoregressively: the model generates the next token based on previously generated tokens and the current context.

Attention Mechanism

Attention mechanisms are a central element of LLMs because they substantially influence both model expressiveness and performance. In general, attention improves the representation of an input sequence by learning dependencies among tokens and weighting their contributions when forming contextual embeddings. Several attention variants are commonly discussed. *Self-attention* refers to the setting where queries, keys, and values are all derived from the same sequence representation within a Transformer block. *Full attention* is often used to describe the standard (dense) form of self-attention, where each token can attend to every other token in the sequence. Finally, *cross-attention* occurs when the queries come from one sequence representation (e.g., a decoder state), while keys and values come from another representation (e.g., an encoder output), enabling information flow across different modules.

Limitations

One of the main practical limitations of current LLMs is context management. In particular, model performance is constrained by a finite context window[1]. As this window fills during long interactions, the model must allocate attention across an increasingly large amount of information, and important mission constraints may become less salient. Recent studies also report a phenomenon often referred to as *context rot*[2]. In practice, this can

[1] The *context window* is the maximum amount of text (measured in tokens) that a model can process at once when generating the next output. Tokens are subword units, so this limit applies to both the prompt and the generated conversation history.

[2] *Context rot* informally describes the progressive degradation of response quality in long conversations: even when the input remains within the nominal context-window

manifest as forgotten requirements, repeated or redundant planning steps, weaker long-horizon consistency, and occasional contradictions with earlier decisions.

This limitation is particularly critical for long-running robotic missions, where state, constraints, safety conditions, and intermediate plans accumulate over time. If not explicitly mitigated through memory management and structured execution control, context degradation can compromise both task effectiveness and operational safety.

In this thesis, we therefore focus on two research questions: (*i*) how can LLM-based techniques be leveraged to support real-time UAV swarm management when mission objectives are provided in natural language? and (*ii*) to what degree are current LLM-based agents mature enough for mission-critical drone applications?

1.1.4 Agents

An *AI agent* can be described as an autonomous software entity that perceives a task context, reasons about possible actions, executes selected actions through available interfaces, observes the resulting feedback, and iteratively updates its decisions until a goal is reached or execution is terminated. In contrast to a passive question-answering system, an agent is expected to be *goal-directed* and *interactive*: it does not only generate text, but uses text-based reasoning to drive concrete operations in an external environment.

Within this perspective, an LLM is the reasoning engine of the agent, but not the full agent by itself. A standalone LLM call is typically stateless (or weakly stateful) and output-oriented, whereas agent behavior requires an explicit control loop that maintains mission context over time, invokes tools through function-calling interfaces, validates intermediate outcomes, and decides whether to continue, re-plan, or stop. This distinction is especially important in robotics and cyber-physical settings, where generated outputs

limit, the model may lose coherence, hallucinate, overlook earlier constraints, or drift from previously established goals.

must be translated into safe, auditable, and reversible actions.

An *agent framework* is the software layer that operationalizes this loop. In practical terms, it provides the components needed to turn model outputs into reliable behavior, including: context and prompt management; short-term and long-term memory mechanisms; tool registration and invocation; world-state tracking; guardrails and policy checks; and execution monitoring with retry/fallback logic. In more advanced configurations, the framework can also coordinate multiple specialized agents (e.g., planner, executor, supervisor) that cooperate under shared constraints.

For UAV swarm management, these capabilities are crucial. Missions are long-running, the environment is dynamic, and decisions must be continuously grounded on telemetry, sensing data, and safety constraints. Therefore, in this thesis, the agent is treated as a closed-loop execution system rather than a one-shot text generator: the LLM provides high-level reasoning, while the surrounding framework ensures structured interaction with external tools and consistent mission control over time.

1.2 Research Challenges

Prior work has extensively explored LLM-based approaches for natural-language interfaces that allow users to issue commands to drones (e.g., [33, 11]). However, once these techniques are moved from command interfaces to real-time robotic control, several issues become apparent. A first difficulty is the heterogeneity of drones, sensors, and communication protocols: the resulting fragmentation produces multiple, incompatible control surfaces, which reduces interoperability and makes it difficult for an LLM to access state information in a unified way without relying on ad hoc prompt scaffolding [31]. A second difficulty concerns duration and interactivity. Long-running missions require persistent tracking of state, as well as repeated perception-action cycles; standalone prompting is not naturally suited to this setting and can degenerate into iterative loops that repeatedly attempt to generate code or

re-issue commands [32]. In such cases, mistakes in produced behaviors can directly translate into mission failure and, more importantly, may create safety risks (for instance, collisions). Finally, evaluation is complicated by the non-deterministic nature of LLM outputs: even with identical inputs, the model may select different actions, leading to different swarm behaviors. For this reason, recent work has emphasized evaluation metrics such as convergence properties and mission success rates measured across repeated trials [13, 22].

1.3 Contributions

Without claiming to address all of the challenges outlined above, this thesis studies AI agents for real-time, mission-agnostic UAV swarm management in a setting where there is no human in the loop during execution. Concretely, we consider a scenario in which a user specifies a mission objective using natural language (e.g., covering a target area), and the AI system is responsible for autonomously coordinating UAV actions from take-off through mission completion.

The main contribution of this work is an agentic, LLM-based architecture designed for long-running drone missions, with explicit support for real-time state retrieval, contextual reasoning, adaptive decision-making, and continuous execution. Compared with many existing LLM-based swarm systems, which often depend on a static code-generation step performed at mission initialization, our platform is characterized by two central innovations. First, it relies on LLM reasoning coupled with function-calling mechanisms to coordinate UAV actions, and it does so without introducing any code-generation phase. Second, rather than treating the mission as a one-shot plan, it maintains continuous control and adapts to real-time feedback from both the UAVs and the environment through a *closed-loop* reason-execute-monitor cycle. Interaction between the LLM agent and the physical world is realized via the Model Context Protocol (MCP) and the W3C Web of Things[3]

[3]https://www.w3.org/WoT/

(WoT) standard [25]. In particular, by modeling each drone as a WoT Thing, the framework enables the agent to discover, query, and actuate swarm resources using a standardized interface, thereby separating mission logic from platform-specific APIs and communication protocols. In addition, we extend the architecture with an execution-controller agent that observes runtime conditions and issues guardrail prompts [15] in order to steer reasoning and promote safe mission execution.

To assess the proposed approach, we conduct an extensive experimental evaluation in a hybrid real-simulated environment by interfacing our platform with the ArduPilot framework, which provides realistic flight control and physics simulation of various types of copters. The evaluation spans three classes of swarm missions: area coverage (with and without planning tools), formation control, and smart irrigation with ground-device communication. Across these missions, we compare six state-of-the-art LLMs that support function calling[4] and that represent different model scales and deployment characteristics: GPT v5.2, DeepSeek v3.2, GLM v4.7, Grok v4.1 Fast, Claude Haiku v4.5, and Qwen 3 8B.

Our results offer multiple observations about the performance and suitability of LLMs for agent-based swarm management. For example, in the coverage scenario, providing external knowledge, such as positioning algorithms drawn from the literature, tends to improve outcomes, although the magnitude and consistency of this improvement vary across models. We also observe that smaller LLMs optimized for low-latency inference generally achieve lower success rate than larger models. Finally, we find no clear relationship between mission success rates and token consumption, which we treat as a proxy for operational cost.

In summary, the contributions of this thesis are as follows:

[4]Function calling is a capability that lets an LLM produce structured tool-invocation requests (e.g., function name plus JSON-like arguments) instead of only free text, so external software can execute actions, return machine-readable results, and feed those results back into the model's reasoning loop.

- A mission-agnostic, agent-enhanced LLM architecture for UAV swarm management;
- A WoT-based interoperability framework enabling real-time bidirectional communication between the AI agent and physical systems (e.g., UAVs);
- A comprehensive empirical evaluation comparing six state-of-the-art LLMs across three swarm-management tasks, analyzing success rate, safety behavior, and token efficiency.

The thesis is structured as follows. Chapter 2 discusses the SOTA (State Of The Art) and related work, from both the drones and LLMs perspectives. Chapter 3 details the system architecture and operational flow. Chapter 4 describes the implementation highlithing all the technologies used in the framework. Chapter 5 presents the experimental setup and reports the results discussing evidences and making plausible assumptions. In the end, the final chapter 6 outlines future work.

Chapter 2

State Of The Art

In this chapter we survey the most relevant research directions at the intersection of UAV swarms, sensing and communication, and security/coordination mechanisms that are pertinent to LLM-driven swarm control. We deliberately omit basic UAV and LLM concepts that have already been introduced in Chapter 1.

2.1 UAV

2.1.1 UAV Swarm Coordination and Trajectory Planning

Recent surveys on UAV-formation trajectory planning classify existing approaches into four broad families: graph-based planners, sampling-based motion planners, optimization-based trajectory generation, and learning-based methods [34]. Graph-based planners (e.g., grid-based search, visibility graphs) and sampling-based methods (e.g., RRT and variants) are widely used for collision-free path finding in cluttered environments, but need additional coordination layers to enforce formation constraints. Optimization-based methods formulate swarm motion as a constrained optimal control problem, often solved via model predictive control (MPC) or convex ap-

proximations, explicitly capturing inter-UAV distances, dynamic limits, and mission objectives in a single framework.

At the control level, three main paradigms are typically distinguished for formation keeping: leader-follower, behavior-based, and virtual-structure control [34]. Leader-follower schemes designate one or more leaders whose trajectories are planned first; followers track relative offsets, which simplifies global coordination but makes formations sensitive to leader failures or link disruptions. Behavior-based approaches, in contrast, rely on distributed rules (e.g., cohesion, separation, alignment) that yield emergent formations and can improve robustness at the cost of reduced guarantees on exact shape maintenance. Virtual-structure methods treat the entire formation as a single rigid body, mapping desired structure motion to individual UAV trajectories and providing strong shape-preservation guarantees under accurate state estimation and control.

Across these families, a recurring limitation is that planning and control are often designed with idealized or static communication assumptions. Intermittent connectivity, dynamic topology changes, and heterogeneous link qualities, central issues in real-world swarms, are rarely part of the trajectory optimization objective. The literature therefore highlights the need for joint design frameworks in which formation control, communication topology, and task allocation are co-optimized rather than treated independently [34, 20].

2.1.2 UAV Sensing and Communication-Aware Operation

Surveys on UAV sensing emphasize the role of drones as mobile sensing platforms that can carry heterogeneous payloads (e.g., optical, multispectral, hyperspectral, LiDAR, radar, and RF sensors) to collect high-quality spatial and temporal data. Compared with static sensing infrastructures, UAVs provide adaptable viewpoints and flexible coverage, but sensing quality becomes tightly coupled with trajectory design, onboard energy constraints, and wireless link conditions [18, 19].

A major trend is the joint design of sensing and communication, often framed as UAV-enabled Integrated Sensing and Communication (ISAC) [18]. In this setting, the same platform and RF resources are used to support communication services and sensing tasks, so trajectory control, beam management, and resource allocation must be co-designed rather than optimized independently. Existing surveys report promising gains [18, 19], but also highlight open issues in online adaptation, robustness to mobility and link variability, and scalability to multi-UAV deployments.

Communication-Assisted Sensing and Edge Offloading

Another research direction studies how communication can actively assist sensing through collaborative offloading and information sharing. Because UAVs are typically constrained in compute and energy, offloading raw or pre-processed sensing data to edge nodes (ground stations, edge servers, or more capable UAVs) can reduce latency for demanding tasks such as dense 3D reconstruction, recognition, and multi-sensor fusion. This motivates joint optimization of communication resources and task-partitioning decisions under delay and energy constraints.[19]

In multi-UAV scenarios, distributed information sharing is central for robust situational awareness, but it also introduces significant overhead. The literature therefore explores efficient aggregation and fusion strategies, including communication-aware scheduling and map-assisted planning to handle dynamic links and obstacles [19]. Despite progress, scalable low-latency coordination under realistic mobility and channel variability remains an open challenge.

Security and Trust Mechanisms in UAV Networks

Security-oriented works [7] highlight blockchain as one possible mechanism to improve integrity, auditability, and decentralized access control in UAV networks, particularly in multi-operator swarm settings. Typical architectures keep lightweight UAV clients at the edge of the blockchain system, while

more capable ground or edge nodes handle ledger maintenance and consensus, reflecting UAV resource constraints and intermittent connectivity.

However, blockchain integration introduces trade-offs in latency, throughput, privacy, and communication overhead. Recent works therefore favor hybrid designs (e.g., off-chain data with on-chain commitments) and permissioned governance models tailored to mission-critical requirements. Systematic evaluations under realistic swarm mobility, adversarial conditions, and strict real-time constraints are still limited.

2.1.3 Discussion and Research Gaps

The surveyed literature shows that UAV swarm coordination, communication-aware sensing, and security mechanisms are active and rapidly evolving areas, but they are still often investigated in isolation. Formation and trajectory planning studies seldom integrate realistic communication and security constraints end-to-end; sensing and offloading studies typically assume fixed control abstractions; and security-focused approaches do not address high-level, language-based mission specification and adaptive swarm reasoning.[34, 19, 7]

This thesis positions itself at the intersection of these lines by adopting agent-enhanced LLMs as high-level reasoning components for UAV swarms, combined with standardized device abstractions and tool-driven execution. The state of the art provides the algorithmic and architectural context for this design, while also exposing a key gap: the lack of reusable end-to-end frameworks that connect natural-language intent, observable execution, and adaptive swarm coordination in a single operational stack.

2.2 Large Language Models and Autonomous Agents

The recent evolution of LLMs from static text predictors to autonomous agents has produced a rich ecosystem of architectures, interoperability pro-

tocols, and evaluation frameworks that extend far beyond traditional prompt-response usage.[1] In this section we focus on those developments that are most relevant to mission-critical cyber-physical control, in particular UAV swarms, and to the design choices adopted in this thesis, namely tool-augmented reasoning, standardized device abstractions, and protocol-level interoperability.

2.2.1 From Code Generation to Tool-Augmented Control

A first wave of LLM-based control systems relied heavily on *code generation*: the model is prompted with high-level specifications and asked to synthesize executable programs or scripts that implement the desired behavior. A recent survey on LLMs for code generation shows that this paradigm has matured across multiple software-engineering tasks [8], including code completion, synthesis from natural language, automated debugging, test generation, and program repair, supported by increasingly sophisticated data pipelines, de-duplication strategies, and evaluation benchmarks [10].

From a robotics and cyber-physical perspective, code generation offers an attractive separation between high-level intent and low-level execution: once synthesized, the program can in principle be verified, versioned, and deployed like any other piece of software. However, the same survey also highlights persistent challenges around hallucinated APIs, brittle dependencies on training distributions, and difficulties in maintaining correctness as environments or requirements evolve [10]. These limitations become particularly acute in long-running missions, where behavior must adapt to real-time feedback, and where re-generating and re-deploying code at every contingency is impractical or unsafe.

Recent work on autonomous LLM agents therefore shifts emphasis from one-shot program synthesis to *tool-augmented, closed-loop control* [26, 35]. Instead of emitting a static program, the model operates in a recurrent loop

[1] For an in-depth survey of autonomous LLM agents, see [5].

in which it plans, calls tools via structured interfaces, observes their outputs, and updates its decisions accordingly. In this design, tools encapsulate concrete capabilities, such as querying world state, invoking actuators, or running specialized planners, while the LLM focuses on high-level reasoning, decomposition, and policy selection.

This thesis adopts the latter approach: LLMs are not used to generate UAV control code, but to orchestrate a set of standardized tools (WoT Things accessed through MCP), enabling continuous, feedback-driven control of a drone swarm. This aligns with emerging best practices in the agent literature, which recommend constraining LLM outputs to typed tool invocations and delegating safety and correctness-critical logic to explicitly implemented components [5].

2.2.2 Architectures for Autonomous LLM Agents

Comprehensive surveys on autonomous LLM agents identify a recurring set of architectural building blocks that extend beyond raw language modeling: (i) a *reasoning module*, typically an LLM with system prompts that define its role and operating conventions; (ii) a *planning layer* that decomposes objectives into sub-tasks or multi-step strategies; (iii) a *tool interface* that exposes external functions, APIs, or simulators; (iv) *memory mechanisms* to store and retrieve intermediate results; and (v) an *execution environment* where actions are carried out and observed [5]. Within this template, individual frameworks differ in how much structure they impose: some remain largely reactive, prompting the model at each step, while others introduce explicit planners, self-evaluation loops, and multi-agent coordination.

A key trend is the explicit modeling of the agent as a *closed-loop control system* [26] that repeatedly interacts with its environment. Surveys report a shift from simple "ReAct"-style loops (interleaving reasoning and tool calls) toward richer pipelines that incorporate strategic planning, self-critique, and hierarchical decomposition, particularly in domains such as complex software engineering and long-horizon decision-making. In parallel, research has be-

gun to explore benchmark suites that stress temporal coherence, robustness to tool errors, and safety properties, rather than only static accuracy on textual tasks.

Beyond conceptual surveys, recent work has also started to introduce concrete experimental frameworks for studying autonomous LLM agents at scale. Agent Laboratory, for example, provides an end-to-end "research co-pilot" environment in which an LLM-based assistant autonomously performs literature search, code implementation, experiment execution, data collection, analysis, and report writing, and is then evaluated by human users along multiple dimensions such as usability, result quality, and research usefulness [24]. This style of framework emphasizes *agent collaboration* (sometimes referred to as a concilium of agents), where multiple specialized agents or roles jointly contribute to decision-making and evaluation rather than relying on a single monolithic model [24, 5]. In the context of UAV swarms, a similar pattern could be applied by equipping each drone with its own local agent and enabling collaborative deliberation across the swarm, so that mission-level decisions emerge from distributed negotiation and shared situational awareness instead of a purely centralized controller; exploring such concilium-style agent collaboration over WoT/MCP abstractions represents a promising direction for extending the architecture proposed in this thesis.

Compared with these general-purpose agent frameworks, the architecture proposed in this thesis instantiates the same design principles in a cyber-physical setting: the Agent Core plays the role of the execution environment and controller, the LLM provides high-level reasoning, and WoT-based tools expose UAVs, sensors, and services through uniform affordances. In particular, guardrail prompts and helper tools mirror the "self-evaluation" and "execution-monitoring" components highlighted in agent surveys, but are specialized for safety-critical swarm control and real-time state verification.

2.2.3 Interoperability Protocols for Agentic LLMs

As LLM agents increasingly operate across heterogeneous tools, services, and platforms, *interoperability* becomes a central concern. A recent survey systematically analyzes four major classes of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent (A2A) protocol, and Agent Network Protocol (ANP), each addressing a different layer of the agent stack [3].

Model Context Protocol (MCP)

MCP is positioned as a standardized, HTTP-based protocol for exposing tools and data sources to LLMs in a model-agnostic way, with explicit schemas for tool definitions, inputs, and outputs [3]. Leading model providers have begun to adopt MCP-like interfaces for function calling, enabling agents to discover and invoke tools without hard-coding provider-specific APIs. The survey emphasizes MCP's role as the *tool-access layer*: it structures how an LLM reasons about capabilities (tools), how external software advertises them, and how results are fed back into the model's context.

Agent Communication Protocol (ACP)

ACP, by contrast, focuses on *message-level communication* between agents rather than between an agent and tools [3]. It specifies how agents represent roles, dialogue state, and message types (e.g., proposals, commitments, justifications), enabling multi-agent workflows such as negotiation, delegation, and consensus. Whereas MCP standardizes tool access, ACP aims to standardize the semantics of inter-agent interaction across heterogeneous implementations.

Agent-to-Agent (A2A) and Agent Network Protocol (ANP)

A2A protocols provide lighter-weight, often application-specific mechanisms for pairwise agent communication, while ANP targets *network-level* con-

cerns such as routing, discovery, and scalability in large agent networks [3]. The survey argues that, taken together, MCP, ACP, A2A, and ANP form a layered interoperability stack: MCP at the tool level, ACP/A2A at the communication level, and ANP at the network-management level. Current implementations remain fragmented, but there is a clear trajectory toward reusable, protocol-driven agent ecosystems.

The architecture developed in this thesis can be seen as an instantiation of this interoperability vision with a particular focus on the MCP layer. By exposing WoT-based drone and sensor abstractions through an MCP gateway, the system aligns with the surveyed best practices for tool-level standardization, while higher-level agent-agent protocols (ACP, A2A, ANP) remain an open avenue for extending swarm-level coordination across multiple agents or services.

2.2.4 Active Tool Discovery

Most early MCP-based systems assume a relatively small, manually curated set of tools, all described in the agent's system prompt or configuration. This design simplifies prompt engineering but does not scale well to large tool ecosystems, nor does it support agents that must autonomously explore new environments or adapt to previously unseen capabilities. The MCP-Zero framework directly addresses this limitation by introducing *active tool discovery* for autonomous LLM agents [4].

Instead of hard-coding all available tools in the initial prompt, MCP-Zero maintains a separate catalog of MCP servers and tool descriptions, and lets the agent dynamically search, retrieve, and select tools at runtime using retrieval-augmented mechanisms [4]. The framework compares different strategies for tool selection: (i) encoding tool metadata directly in the system prompt; (ii) using retrieval-augmented generation (RAG) over long contexts; and (iii) the proposed active-discovery method, where the agent issues explicit "tool queries" and receives ranked candidates. Experiments show that active discovery can reduce prompt length, improve selection accuracy when

the tool set is large, and better support multi-turn scenarios where different phases of a task require different subsets of tools [4].

While MCP-Zero is evaluated primarily on software-oriented benchmarks, the underlying ideas are directly relevant to cyber-physical systems such as UAV swarms. In realistic deployments, the set of available platforms, sensors, and services may change over time, and an agent that can discover and bind to new WoT Things or planning services at runtime would be more robust and extensible than one that relies on a fixed, pre-registered tool set. The current thesis adopts a static MCP tool surface tailored to a specific swarm scenario, but its WoT and MCP-based design is compatible with future integration of MCP-Zero-style discovery.

2.2.5 Discussion and Research Gaps

The surveyed literature sketches a clear trajectory: from static code generation toward tool-augmented, closed-loop LLM agents; from ad hoc integrations toward protocolized interoperability via MCP and related standards; and from small, hand-crafted tool sets toward scalable, discovery-enabled ecosystems [10, 3, 4, 5]. Most empirical evaluations, however, are still conducted in digital domains (software engineering, web automation, information retrieval), or in robotic scenarios where the environment is abstracted by simulators with relatively simple temporal and safety constraints.

By contrast, the problem addressed in this thesis, real-time, autonomous management of UAV swarms with heterogeneous platforms and sensing resources, poses additional challenges. First, cyber-physical missions demand strict safety guarantees (e.g., collision avoidance, proper landing and disarming), which are only partially addressed in current agent evaluations and usually not grounded in realistic flight dynamics [5]. Second, interoperability in robotic settings must reconcile high-level protocols such as MCP with device-level standards like the W3C Web of Things, ensuring that abstractions remain expressive enough for control while hiding low-level communication details. Third, as highlighted by existing LLM-UAV studies, code-

generation-based approaches struggle to incorporate fine-grained, real-time feedback during execution, leading to brittle or static mission logic.

The architecture proposed here occupies this gap by: (i) combining an agentic LLM with a WoT and MCP-based execution environment, so that all UAV and sensor interactions pass through standardized, machine-readable affordances; (ii) emphasizing long-running, closed-loop reasoning with explicit guardrails and helper tools for state verification; and (iii) evaluating multiple LLMs on realistic swarm missions with quantitative metrics such as success rate, collision count, execution time, and token efficiency. In this sense, the work can be viewed as a domain-specific extension of the general agent frameworks surveyed above, bringing protocol-driven, tool-augmented LLM agents into the concrete setting of UAV swarm control and exposing limitations and opportunities that are not visible in purely digital benchmarks.

2.3 LLM-UAV

LLMs, after influencing several networking domains, are widely anticipated to catalyze a paradigm shift in mission-critical communications and in the management of robotic swarms [12]. The importance of this research direction is underscored by the sharp increase in publications over the past two years, even though the underlying technologies are still relatively new. An up-to-date survey on LLM-UAV integration is provided in [28], which summarizes both opportunities and open challenges across navigation, perception, and planning.

Across the literature, three broad lines of work on LLMs for UAV swarm management can be distinguished: bandwidth optimization (e.g., [14]), positioning optimization, and task code generation. Within positioning-oriented approaches, LLMs are commonly employed to interpret natural-language requests and infer positioning constraints, while the actual low-level control is delegated to established techniques. For instance, [33] investigates LLM-based Drone Delivery as a Service (DDaS) platforms in which delivery re-

quirements are extracted from free-text user requests. In [11], LLMs support the specification of swarm choreographies by translating high-level natural-language objectives into inputs for a motion-planning framework that then computes collision-free trajectories. Along similar lines, FlockGPT [16] showcases LLM-assisted generation of geometric formations for drone swarms. In [6], the authors introduce SwarmChain, a collaborative inference approach that distributes LLM computation across resource-constrained UAVs through model-splitting mechanisms.

Among these directions, the use of LLMs to generate executable robotic code is the most directly related to our contribution. In [22], the authors compare classical swarm methods, specifically Boids and Ant Colony Optimization, against LLM-generated implementations. Their findings indicate that traditional Boids consistently outperform LLM-based variants, whereas LLM-generated ACO can surpass its classical counterpart, emphasizing the role of prompt design. Building on this perspective, [31] proposes a structured prompting framework that explicitly exposes available APIs, domain constraints, and illustrative examples in order to improve the reliability of reasoning in LLM-driven swarm control. A closed-loop, code-generation workflow is then proposed in [32], where mission code produced by an LLM is repeatedly validated in simulation and refined using feedback from a secondary AI agent. Even so, the resulting code remains essentially static and does not natively incorporate real-time feedback during execution. This shortcoming is addressed to some extent in [27], which uses prompt representations resembling programs to describe available actions and relevant environmental objects, enabling more structured interaction. However, this strategy does not exploit interoperability standards such as the W3C Web of Things (WoT), whose value for automated, LLM-based mashup generation has been recently illustrated in [23]. Finally, [13] proposes a unified LLM-driven framework that performs multiple stages of multi-robot task planning directly from natural-language instructions. Experiments on complex tasks report consistent performance across different LLMs, while ablation results

underline the benefits of modular planning stages.

Chapter 3

System Architecture

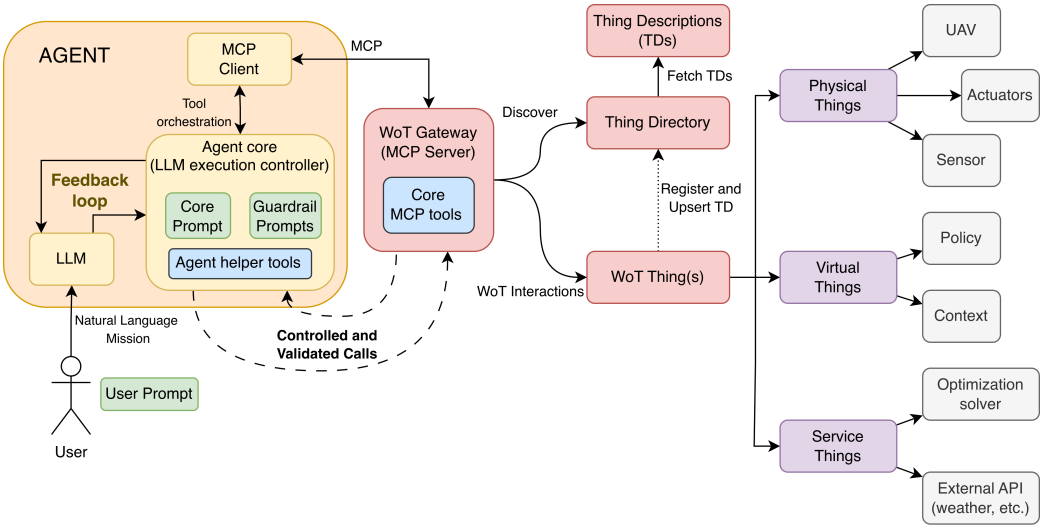


Figure 3.1: High-level overview of the proposed agent-enhanced, WoT-directed architecture. The Agent encapsulates an LLM and an Agent Core with persistent prompts and guardrails, interacting with a WoT ecosystem through controlled MCP-mediated calls.

In this thesis, we present a novel mission-agnostic software framework that enables autonomous, LLM-based management of heterogeneous UAV swarms operating in dynamic and uncertain environments. The main goal of the framework is to let users express high-level mission objectives, such as area

monitoring, formation control, or distributed data collection, through natural language. Starting from these high-level objectives, the system autonomously coordinates sensing, actuation, and mobility across multiple aerial devices in order to execute the mission in a grounded and operationally consistent way.

Figure 3.1 illustrates the proposed architecture at a high level. Compared with state-of-the-art approaches discussed in Section 2.3, our solution introduces three key novelties (N):

- *N1: Interoperability support*, namely the capability to manage heterogeneous drone swarms through a shared, hardware-independent interaction model;
- *N2: Real-time system-state retrieval*, including both swarm drones and external data sources;
- *N3: Autonomous closed-loop reasoning*, where system-state feedback is reintegrated into subsequent reasoning steps, enabling iterative tool calling and adaptation during mission execution without human intervention.

Novelties *N1* and *N2* are achieved by adopting the W3C Web of Things (WoT) standard [25] within a Model Context Protocol (MCP) environment [2]. In particular, the WoT paradigm, standardized by the W3C, provides a uniform abstraction layer in which heterogeneous devices expose capabilities as Web resources described through machine-readable Thing Descriptions (TDs). Within our architecture, system capabilities, including drones, sensors, abstract mission state, and supporting services, are all uniformly represented as WoT Things. This design establishes a common interaction model for swarm-level resources and supports runtime discovery, late binding, and uniform control through standardized interaction patterns.

As a direct consequence, heterogeneous drone platforms and auxiliary services can be integrated without changing the core swarm-management logic (*N1*). In addition, the system state exposed through TDs is made available in real time to the LLM via MCP, an open protocol that enables AI

applications to connect to external data providers. This allows the swarm to be managed during execution through structured AI-agent invocations (function-calling), without any code-generation phase (*N2*).

To enable *N3*, we design an *Agent Core* module that grounds LLM reasoning in physical execution by constraining decision making to available WoT affordances. Concretely, the Agent Core mediates execution through structured tool requests and responses, thereby reducing the space of admissible interactions and mitigating hallucinated commands, while progressively feeding execution feedback into subsequent reasoning steps (*N3*).

In the following, we describe the three main subsystems of the architecture shown in Figure 3.1. Finally, in Section 3.4, we detail the end-to-end operational flow.

3.1 W3C WoT Ecosystem

At a conceptual level, the W3C WoT standard provides a common application-layer model that sits above heterogeneous IoT protocols. Rather than replacing existing communication stacks, WoT offers a protocol-agnostic interaction abstraction that allows clients to interact with diverse devices through a uniform model. This abstraction is centered on *Thing Descriptions* (TDs), i.e., machine-readable metadata documents that describe the capabilities of a Thing, the data exchanged, security information, and the protocol bindings needed to access each interaction endpoint. In this way, applications can discover and consume resources consistently even when underlying technologies differ.

Within this model, *Properties* represent the exposed state of a Thing. A Property is typically readable, may optionally be writable, and can also be observable so that consumers receive state updates when changes occur. In practice, Properties are used to access sensor readings, configuration parameters, and state variables that need to be inspected or updated in a controlled way.

Actions represent executable functions offered by a Thing. Unlike Properties, which expose state, Actions capture operations that are explicitly invoked by a consumer and may trigger immediate or long-running processes, possibly affecting one or more internal states. This interaction type is particularly suitable for command-like operations such as actuation, coordinated task execution, or invocation of internal procedures that are not directly modeled as simple state assignments.

Events represent asynchronous notifications produced by a Thing and pushed to consumers when relevant state transitions or runtime conditions occur. Events are therefore used when the interaction must be notification-driven rather than polling-driven, for example for alarms, mission milestones, or periodic streams that must be propagated as they are generated. Together, Properties, Actions, and Events form the core WoT interaction model adopted in our architecture.

The right-hand side of Fig. 3.1 represents the WoT ecosystem in our architecture, which provides interfaces toward physical entities (e.g., drones and tools). It includes three core components: Thing Descriptions, the Thing Directory, and WoT Things.

Thing Descriptions provide a machine-readable interface specification including drone-level operations such as arming, takeoff, navigation, or mode switching, that together with sensor-level operations such as measurement sampling and reporting, can be represented as TD actions. The *Thing Directory* maintains a registry of available TDs and supports discovery operations, which enables late binding and runtime adaptability.

WoT Things are the execution entities that expose capabilities through TDs. Each Thing encapsulates the logic needed to map WoT-level affordance interactions into device-specific or service-specific operations, while remaining fully decoupled from LLM-driven reasoning. In our framework, WoT Things are organized into three categories: (i) *Physical Things*, such as drones and sensors; (ii) *Virtual Things*, representing abstract or derived entities such as mission state or contextual information; and (iii) *Service*

Things, representing computational or informational services such as planners, optimizers, or external APIs. Swarm-level behavior then emerges from coordinated orchestration of multiple WoT Things by the Agent Core. By composing primitive interactions across multiple entities, the framework realizes swarm-level missions while remaining decoupled from platform-specific APIs and communication protocols. At the same time, every resource is accessed through the same WoT interaction model.

3.2 MCP-based WoT Gateway

The MCP-based WoT Gateway acts as the exclusive bridge between the agent-enhanced LLM and the WoT ecosystem. From the Agent's perspective, it is the only interface available for discovering and interacting with external resources, thus enforcing a strictly *WoT-directed* execution model.

The MCP-based WoT Gateway provides two core functionalities, exposed as *core MCP tools*. First, it supports *resource discovery* by querying the Thing Directory to identify WoT Things that satisfy task requirements. Second, it supports *interaction* by mediating access to WoT Things through affordances defined in their TDs.

In our architecture, the WoT Gateway is implemented as the MCP server, while the Agent operates as the MCP client. Together, they improve system reliability and keep interactions well structured through request-payload validation and sanitization.

Execution outcomes, state updates, and event notifications are propagated back to the Agent through MCP. This mediation preserves protocol transparency and maintains a clear separation between high-level reasoning and low-level execution.

3.3 The Agent

The leftmost part of the architecture in Figure 3.1, denoted as the *Agent*, is the central execution entity responsible for interpreting user intents and orchestrating interactions with the external system. The Agent consists of three components: an LLM, an Agent Core, and an MCP client.

3.3.1 LLM

The *LLM* provides high-level reasoning capabilities for interpreting user-defined natural-language missions, such as area coverage, formation control, or coordinated data collection. Instead of generating low-level drone-control code, the LLM operates at a semantic level and reasons over abstract concepts, including capabilities, constraints, and execution intent. Using the tokens derived either from the user's natural-language request or from a recent iteration (which may consist of a tool invocation or intermediate reasoning), the model processes them within its context window and returns an output that can be structured reasoning steps and tool-invocation requests, which are grounded by the Agent Core before affecting the physical system.

3.3.2 Agent Core

The *Agent Core* acts as the execution controller and contextual runtime around the LLM. It hosts the LLM-driven execution loop and maintains a persistent execution context through a *core prompt*. This prompt does not encode hard-coded control logic or planning algorithms; rather, it defines semantic boundaries and interaction principles that govern LLM behavior, including its role as a drone-swarm coordinator and the use of W3C WoT as the interaction paradigm. Beyond the core prompt, the Agent Core manages a set of *guardrail prompts* [15] that can be activated conditionally at runtime. These guardrails are triggered by situations such as incomplete task execution, invalid interaction sequences, or unmet termination conditions. When triggered, they are injected into the LLM context to constrain or redi-

rect reasoning, thereby reinforcing execution correctness without hard-coded control policies.

As a result, the interaction between the LLM and the Agent Core forms a closed *feedback loop*, as shown in Figure 3.1. Observations produced by tool invocations and execution feedback are progressively incorporated into subsequent reasoning steps, enabling adaptive behavior across long-running missions. This resulting loop is "closed" and does not include a human in the loop, because the Agent issues direct requests to the MCP layer and receives direct feedback from the drones. The user provides only the initial natural-language request and does not supervise the full execution process; the Agent continues until the goal is reached, the mission is deemed infeasible, or a timeout condition occurs.

3.3.3 MCP Client

The *MCP client* provides the communication interface between the Agent and the external execution environment. Beyond simple transport, it acts as the single mediation point through which the Agent invokes gateway-backed MCP tools and receives structured responses, ensuring consistent request/response handling across all mission phases. In addition, the Agent Core integrates a set of *agent helper tools* that extend the basic LLM-driven execution loop with auxiliary computational and control capabilities. These tools address limitations related to reasoning efficiency, numerical processing, and execution semantics that may arise with specific LLM configurations, including edge-oriented models (examples are discussed in Section V). In practical terms, the helper layer can expose higher-level operations such as bulk position dispatch and state-verification routines (e.g., wait-until-armed, airborne, arrived, or safely landed/disarmed), which encapsulate repetitive polling and completion checks. This design reduces long-horizon bookkeeping burden and unnecessary tool-call volume while preserving mission semantics. Importantly, helper tools do not bypass core interaction constraints: they invoke the same validated MCP tool surface through the MCP client, thereby

maintaining consistent WoT-grounded execution and traceable state feedback.

3.4 End-to-End Operational Flow

From an operational perspective, swarm mission execution is organized as an iterative reasoning-execution loop managed by the Agent Core (see Figure 3.2). At initialization, the Agent Core loads the *core prompt*, which defines the swarm-management context, the WoT-directed interaction model, the available MCP tools, and the execution constraints that govern LLM behavior. One representative constraint is maintaining minimum safety distances between drones during mission execution to prevent collision events. The core prompt typically follows a structured template composed of short paragraphs and rule lists that specify phase-level procedures, safety constraints, and essential operational context. For larger models, the core prompt intentionally avoids mission-specific step-by-step instructions, thus preserving the mission-agnostic nature of the framework (see Appendix Section A.1). For smaller models, such as Qwen 3 8B, additional workflow guidance is provided to improve execution reliability, at the cost of partially reducing this agnosticity (see Appendix Section A.2).

A mission starts when a user provides a high-level objective in natural language. For example, a user may specify a mission in which all available drones must reach a target area and collect data from ground-based IoT sensors. Starting from the *user prompt*, the LLM produces an initial reasoning state that can include inferred objectives, constraints, and intended interactions with the environment. During execution, the LLM issues tool-invocation requests (i.e., WoT-Thing affordance invocations), which are mediated by the Agent Core and executed through the MCP-based WoT Gateway.

Runtime guardrails are continuously evaluated to constrain or halt interactions in response to invalid requests or unsafe conditions. Moreover, execution feedback, including telemetry updates, sensor readings, and event

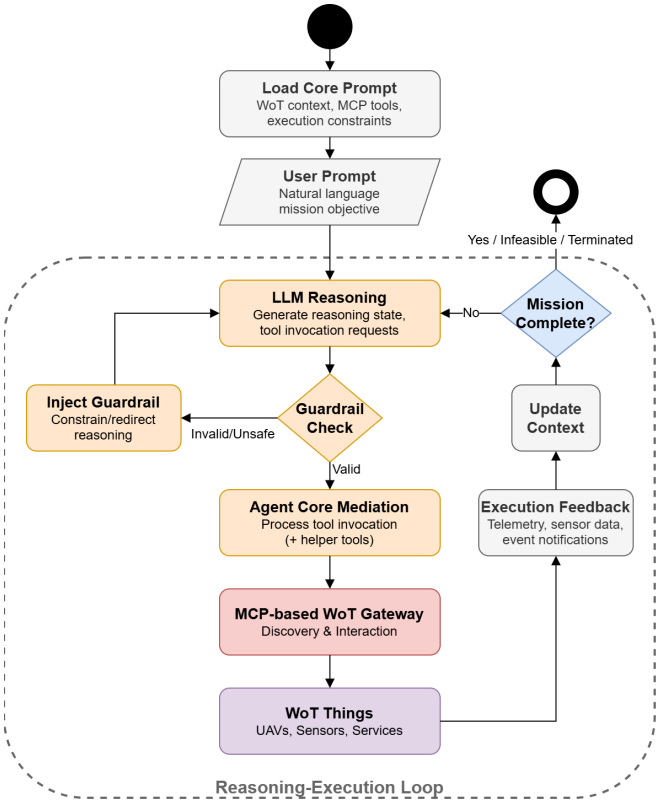


Figure 3.2: Operational flow of the agent-enhanced swarm mission execution. The iterative reasoning-execution loop integrates LLM-based planning, runtime guardrails, and MCP-mediated WoT interactions, with execution feedback progressively incorporated into subsequent reasoning steps.

notifications, is appended to the interaction context and incorporated into subsequent reasoning steps. Through repeated composition of primitive per-entity interactions, coordinated behavior across multiple drones can progressively emerge, enabling the realization of swarm-level missions.

During the execution loop, feedback from execution is fed into each new model iteration and contributes to the context, which is updated at every iteration.

The execution loop continues until mission objectives are achieved, detected as infeasible under current conditions, or explicitly terminated. This operational model enables flexible and largely automated drone-swarm man-

agement while preserving grounded interaction, safety constraints, and a clear separation among semantic reasoning, execution orchestration, and device-level control.

Chapter 4

Implementation

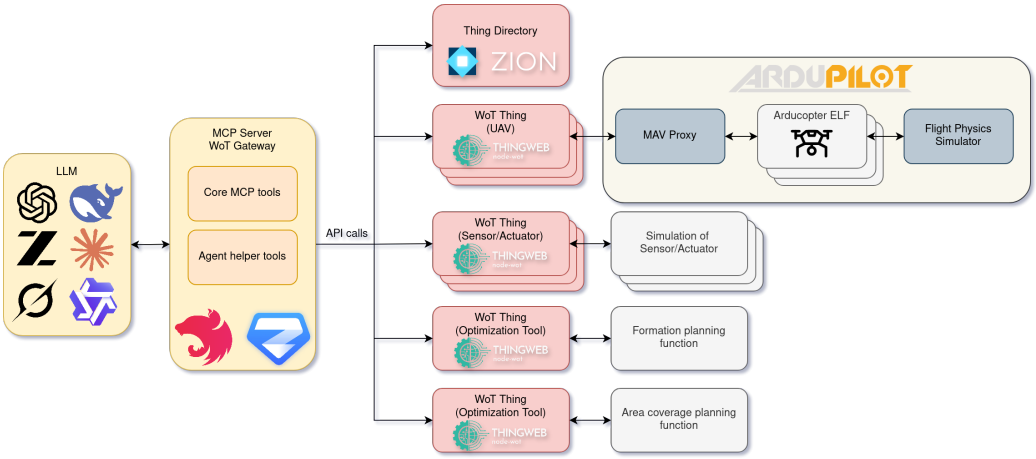


Figure 4.1: Implementation overview of the proposed system, highlighting the Web-of-Things layer, the MCP-based gateway, and the LLM agent with their main interactions.

This chapter details the implementation of the architecture introduced in Chapter 3. Following Figure 3.1, the system is realized as three layers: (*i*) a *Web-of-Things (WoT) layer* exposing UAVs and auxiliary devices as W3C-compliant Things; (*ii*) an *MCP-based WoT Gateway* providing typed access to WoT discovery and interaction; and (*iii*) an *LLM-based Agent* executing the iterative reasoning–action loop detailed in Section 3.4.

In our implementation each layer consist in one or more service's components. For example the WoT layer will include multiple services such as the Thing Description Directory and the WoT servients. The system is composed of five main services:

1. A multi-vehicle digital twin based on ArduPilot SITL;

2. WoT servients that expose each drone and sensor as a Thing;

3. A Thing Description Directory (Zion) used as the single source of truth for discovery;

4. An MCP server that exposes discovery, actuation, and planning tools to LLMs;

5. An LLM agent and evaluation harness that orchestrate missions and collect telemetry and logs.

All services are deployed via Docker Compose. Therefore each layer is realized as a set of containerized services with well-defined interfaces to ensure modularity, reproducibility, and ease of deployment. Each service and component use different technologies, in the Table 4.1 is it possible to have a general look about the service's component, the technology that it use and its responsability.

4.1 Web-of-Things Layer

The WoT layer exposes all physical and virtual resources involved in mission execution as Web Things described through TDs. Each Web Thing defines its properties, actions, and events, enabling uniform access and runtime discovery.

Component	Tech	Responsibility
Drone digital twin	ArduPilot SITL + MAVProxy	Simulate a fleet of quadrotors, emit MAVLink telemetry, and accept MAVLink commands.
Drone servients	Node.js + Node-WoT	Expose each drone as a WoT Thing (actions/properties) and periodically upsert its TD to Zion.
Sensor simulator	Node.js + Node-WoT	Expose stationary sensors (humidity, temperature, ...) as WoT Things with position/range metadata and synthetic readings.
Thing directory	Zion + PostgreSQL	Store and serve TDs for discovery and late binding.
MCP gateway	NestJS + Zod schemas	Provide typed MCP tools for discovery, actuation, and planning; proxy calls to WoT Things.
LLM agent	Python + LangChain	Tool-calling loop that plans and executes missions, verifies state, and logs tool I/O.

Table 4.1: Main implementation components and their roles.

4.1.1 WoT UAV Things

MAVLink Protocol

MAVLink (Micro Air Vehicle Link) is the protocol used in our implementation to exchange control commands and telemetry between the simulated vehicles and upper software layers. As described in the ArduPilot developer documentation, MAVLink is a compact binary protocol with transport-

independent framing, and messages are intentionally small (up to 263 bytes in MAVLink v1 and up to 280 bytes in MAVLink v2)[1]. This lightweight packet structure is particularly useful when multiple vehicles must stream state and receive commands at high frequency, because it limits communication overhead while preserving a standardized message model.

In our stack, MAVProxy acts as the bridge between WoT-facing services and MAVLink endpoints[2]. MAVProxy forwards command traffic to the correct vehicle instance and relays telemetry back to the rest of the system; it can also forward streams to multiple consumers when needed. This bridging role keeps MAVLink handling isolated from high-level agent logic. In addition, because MAVLink delivery is not guaranteed at protocol level, mission execution relies on explicit state verification after command issuance (e.g., by reading position, mode, and arming status) rather than assuming that a single command acknowledgement implies completion.

ArduPilot

ArduPilot is used as the autopilot backend for the UAV digital twin. In this work, vehicles are executed in Software-In-The-Loop (SITL), so flight-control behavior and telemetry can be reproduced in a fully software-based environment while preserving realistic autopilot semantics. This enables repeatable experiments under controlled conditions and allows the same mission pipeline to be validated without requiring physical drones during development.

From an integration perspective, ArduPilot provides the low-level execution semantics consumed through MAVLink and exposed upward through WoT abstractions. Operations such as arming, takeoff, guided navigation, return-to-launch, and landing are therefore represented as high-level tool calls while still mapping to concrete autopilot state transitions. This preserves separation of concerns: the agent reasons at task level, while ArduPilot enforces flight-level execution behavior.

[1] https://ardupilot.org/dev/docs/mavlink-basics.html
[2] https://ardupilot.org/mavproxy/index.html

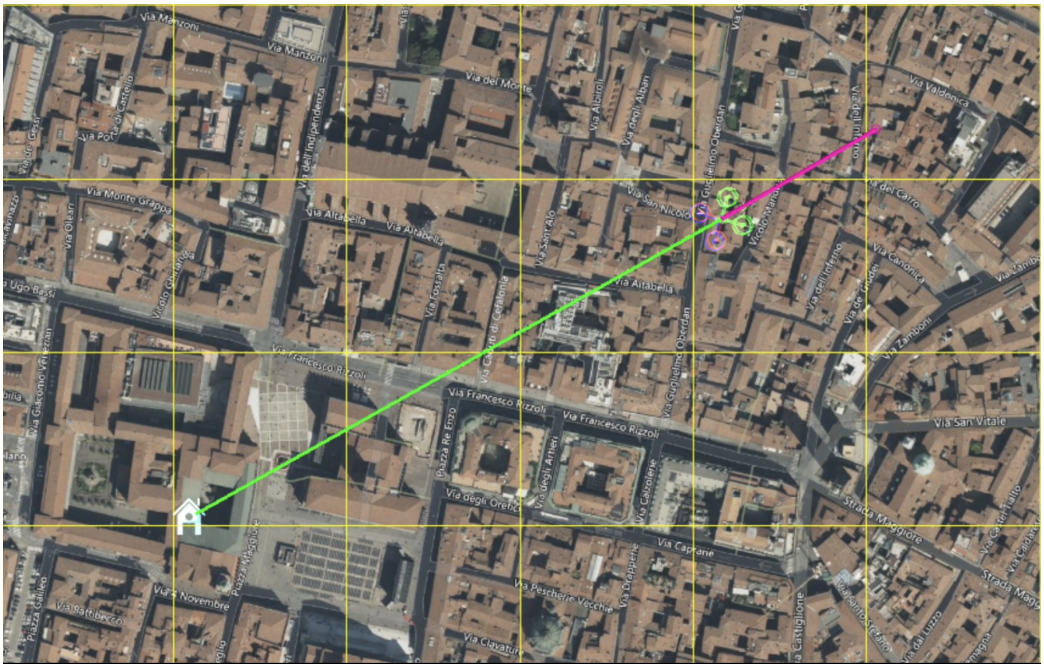


Figure 4.2: ArduPilot UAV map view

Overview

Each UAV is exposed as an individual WoT Thing by a dedicated Servient. The Servient provides a standardized WoT mobility/telemetry interface and forwards the actual commands through MAVProxy, which bridges the requests to the ArduPilot simulator via MAVLink. MAVProxy multiplexes MAVLink traffic and forwards it to a single `mavlink2rest`[3] endpoint, providing an HTTP interface for reading vehicle state and issuing commands. This architecture keeps the simulation fast to reset and allows multiple WoT servients to share the same MAVLink transport while addressing different SYSIDs. The MAVLink protocol is adopted due to its wide availability, vendor neutrality, and well-defined semantics, enabling interoperability across heterogeneous UAV platforms without exposing low-level flight-control details to the Agent. The Thing ID follows a URN convention: `urn:dev:ops:drone:{sysid}`.

[3]https://github.com/mavlink/mavlink2rest

The TD exposes:

- **Actions** for actuation (e.g., `arm`, `disarm`, `takeoff`, `goto`, `rtl`, `land`, `setMode`, `setSpeed`, `setYaw`) and for sensor interaction (`collectSensorData`).
- **Properties** for telemetry and metadata (e.g., `position`, `mode`, `armed`, `battery`, `groundspeed`, `specs`).

Actuation is implemented by mapping WoT actions to MAVLink commands. For example, arming uses `MAV_CMD_COMPONENT_ARM_DISARM`, mode changes use `MAV_CMD_DO_SET_MODE`, takeoff uses `MAV_CMD_NAV_TAKEOFF`, and waypoint motion uses `MISSION_ITEM_INT`. Action inputs and state transitions are validated at the Servient boundary to enforce basic execution constraints and prevent ill-defined or unsafe interactions. As an example, the `takeoff` requires a strictly positive `alt` field, and the servient sets the drone to `GUIDED` before issuing a takeoff command. Such validation ensures that higher-level components always operate on physically meaningful abstractions, independently of the reasoning capabilities of the LLM.

Listing 4.1: Excerpt from the UAV Thing Description for the `goto` action.

```
...
"goto": {
    description: "GUIDED goto lat/lon/alt (m AGL)",
    input: {
        properties: {
            lat: { type: "number" },
            lon: { type: "number" },
            alt: { type: "number" },
        },
        required: ["lat", "lon", "alt"],
    },
},
...
```

In Listing 4.1, we show a portion of the TD for the *goto* action; a complete

UAV TD example is available at https://doi.org/10.5281/zenodo.18433213. Additional sensing and actuation resources are exposed as independent WoT Things and become automatically discoverable once their TDs are registered, without requiring changes to the Agent or the MCP-based WoT Gateway.

4.1.2 Thing Description Directory

Zion[4] is used as the Thing Description Directory and acts as the authoritative system catalog for all available Web Things. Concretely, drone and sensor servients register their TDs at startup and periodically upsert them to keep runtime metadata synchronized with the current deployment. This gives the platform a single discovery surface even when Things are distributed across multiple containers and network endpoints.

From an interface perspective, Zion exposes standard CRUDL operations over TDs, which we use for registration and lookup. Discovery requests can be expressed through TD metadata filters and JSONPath-based constraints, so clients can search for capabilities (e.g., Things exposing `goto` or `collectSensorData`) rather than relying on fixed Thing IDs. As a result, the Agent and MCP gateway perform late binding: they resolve the target Thing and its forms at runtime from the directory before executing actions.

This design is central for reproducibility and extensibility. Adding a new drone or sensor requires only starting its servient and publishing its TD to Zion; no hard-coded endpoint updates are needed in the Agent logic. Likewise, if a Thing implementation changes while preserving its TD contract, higher layers remain stable because integration depends on the directory-level WoT abstraction rather than direct service coupling.

4.1.3 WoT Sensor Simulator

The sensor simulator reads a declarative manifest and spawns a set of WoT Things (e.g., crop humidity sensors and a field temperature sensor). Each

[4]https://github.com/vaimee/zion

sensor Thing includes static metadata (position, range) and a synthetic reading stream generated by a bounded random walk with a sensor-type specific range.

The `collectSensorData` action allows a drone to request a measurement from a sensor Thing. The drone's servient includes the correspondent drone's GPS position in the request, and the sensor (and in the simulated scenario, the sensor simulator) validates that the requester is within a configured transmission range before returning a reading. This implements a realistic constraint where sensing is feasible only when the platform is physically close enough to the field device.

4.2 MCP Tool Gateway

The MCP-based WoT Gateway is implemented as a dedicated NestJS server and represents the only integration point between the Agent and the WoT layer. It exposes a set of *core MCP tools* for discovery and interaction, each defined with explicit input and output schemas and validated at runtime using Zod[5]. Zod is a TypeScript-first schema validation library that lets developers define the expected structure, types, and constraints of tool inputs and outputs in a single declarative schema. In this implementation, Zod provides both runtime checks (to reject malformed or incomplete tool arguments) and static type inference inside NestJS services. This combination reduces integration errors, makes tool contracts explicit, and helps contain LLM hallucinations by enforcing strict, machine-checkable I/O boundaries.

To support asynchronous and long-running operations, the gateway separates action invocation from execution completion. Action calls return acknowledgements or TD form references, while completion can be verified by the Agent through subsequent property reads or event notifications. The gateway also supports batched and multi-Thing invocations, reducing latency and token consumption in coordinated multi-UAV operations.

[5]https://zod.dev/

4.3 LLM Mission Agent

The LLM Mission Agent is developed in Python using the LangChain[6] framework and provides a concrete implementation of the Agent architecture described in Chapter 3. More precisely:

4.3.1 Prompt Realization

The Agent Core maintains three prompt artifacts with distinct lifecycles. A persistent *core prompt* is loaded at startup and remains fixed throughout a run; it encodes execution invariants identified as critical for reliable operation, including mandatory state verification, safe termination conditions, and exclusive use of MCP-mediated interactions. A *user prompt* is injected once per run to specify the mission objective. Both prompts are implemented as static templates and are not modified during execution.

4.3.2 Guardrails Derived from Execution Failures

Runtime guardrails are implemented as short, targeted prompt fragments that are injected dynamically by the Agent Core. Unlike the core prompt, guardrails were not designed a priori but derived empirically from repeated failure patterns observed during early deployments and simulation runs. Specifically, guardrails are triggered when the Agent Core detects conditions such as: (*i*) inferred task completion without corresponding state verification, (*ii*) stalled execution characterized by repeated non-progressing tool calls, or (*iii*) attempted termination while one or more drones remain airborne or armed. When activated, guardrails explicitly instruct the LLM to re-evaluate system state and complete missing execution steps. This mechanism improves robustness without embedding task-specific control logic in code.

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

4.3.3 Helper Tools as Execution Shortcuts

Agent helper tools are implemented as optional, higher-level wrappers around existing MCP tools. They encapsulate recurring execution patterns such as synchronization, completion verification, and coordinated multi-drone commands that were observed to cause excessive tool-call verbosity or bookkeeping errors, particularly in smaller or latency-optimized models. Importantly, the helper tools do not bypass the WoT-directed execution model: internally, they invoke the same validated MCP tools available to the LLM. Their availability is explicitly controlled in the evaluation to assess their impact on execution reliability and token efficiency.

Concrete examples of the prompts, guardrails, and helper tools used in the experiments are reported in Chapter 5.

4.4 Swarm Monitor

While the previous sections focused on WoT and MCP integration, this section describes the operator-facing Swarm Monitor. The monitor provides a single operational surface for supervising and controlling the WoT-enabled drone fleet and its environmental sensors. Its information architecture is organized into two connected views: *Fleet*, for real-time drone supervision and command dispatch, and *Sensors*, for sensor lifecycle management. Both views use live polling, so telemetry and readings are continuously updated without manual page reloads.

4.4.1 Operational Scope and Data Flow

From an architectural perspective, the frontend acts as a thin orchestration layer above two HTTP backends: the set of drone WoT servients (for property reads and action invocation) and the sensor simulator API (for listing, creating, and updating sensor resources).

The Fleet view periodically collects per drone state (armed flag, flight

mode, position, groundspeed, battery, and platform specifications), while the Sensors view retrieves the latest readings and metadata for all registered sensors. This separation maintains a clear boundary between mission-time actuation (drone controls) and infrastructure administration (sensor registry operations).

4.4.2 Fleet View: Monitoring and Command Interface

The Fleet view combines refresh control, geospatial situational awareness, compact sensor awareness, and per-drone command panels in a single workflow. At page level, the operator can set the telemetry polling interval in milliseconds (with a lower bound to avoid excessive request rates), trigger an immediate refresh of both drone telemetry and sensor data, and export a JSON file containing currently available drone positions plus optional coverage-relevant metadata (altitude and camera field of view).

The map (see Figure 4.3) overlays drone markers and sensor markers. Drone popups report label, mode, groundspeed, latitude, and longitude, while sensor popups report name, type, and the latest measured value when available. If no valid coordinates are available yet, the interface shows an explicit waiting state rather than stale map content.

Each drone card (see Figure 4.4) reports current status indicators (armed/idle state, mode, and last update timestamp), key telemetry (latitude, longitude, altitude, groundspeed), a battery summary (remaining percentage, voltage, current), and a direct link to the corresponding Thing endpoint.

Command controls are exposed through buttons and forms for `arm`, `setMode` to `GUIDED`, `disarm`, `rtl`, `land`, `takeoff` with operator-defined altitude, `setSpeed` with operator-defined target speed, and `goto` with operator-defined latitude, longitude, and altitude.

Action dispatch feedback is explicit and stateful (*pending*, *success*, or *error*) so operators can distinguish acknowledged commands from failures. If a servient is unreachable, the card reports the communication error and

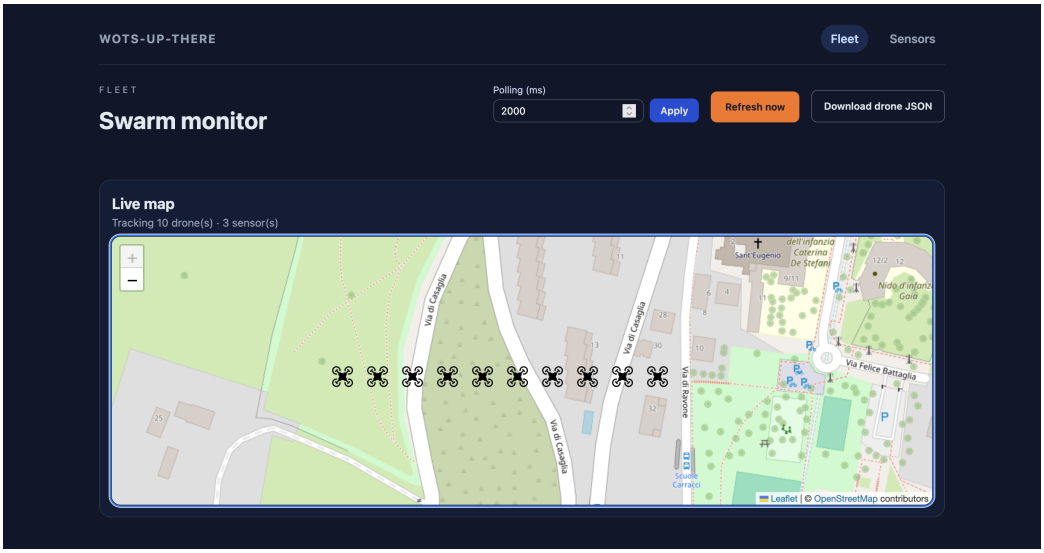


Figure 4.3: Swarm monitor frontend map

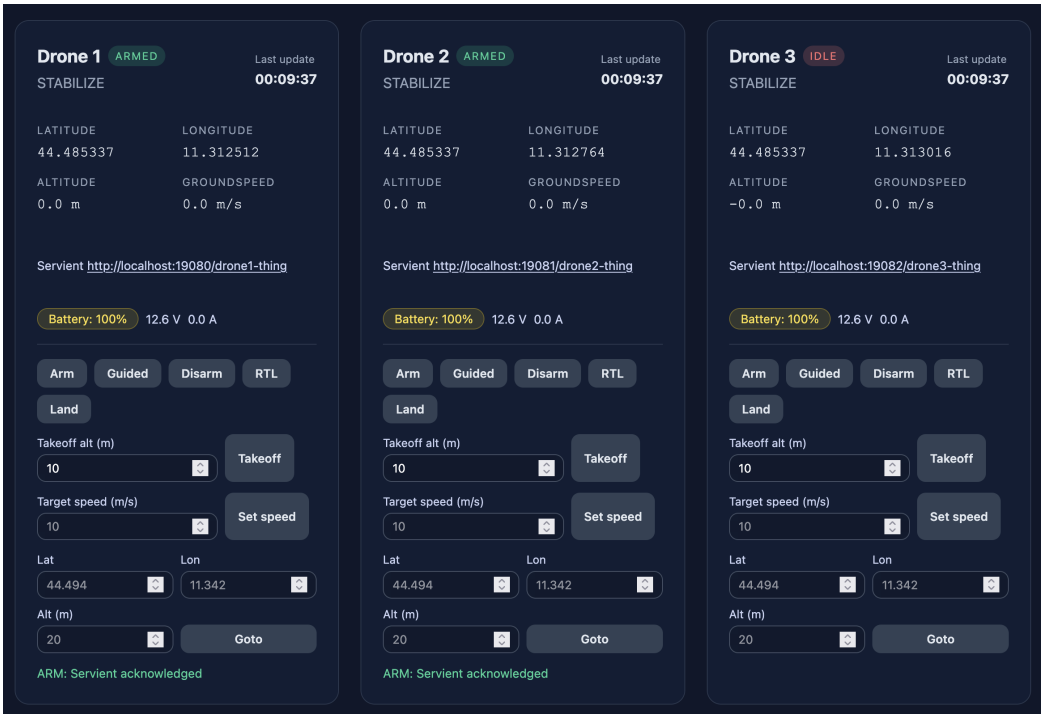


Figure 4.4: Swarm monitor frontend drones view

disables action controls to prevent unsafe blind dispatch.

4.4.3 Battery Management

The battery button opens a dedicated modal that allows operators to simulate battery telemetry values (remaining charge, voltage, current), enabling repeatable tests of low-energy or degraded-power scenarios. The same modal supports clearing the override and returning to live telemetry.

4.4.4 Sensors View: Dashboard and Resource Management

A compact "Ambient sensors" strip (see Figure 4.5) presents a short list of live sensor readings directly in the Fleet page and provides a direct navigation link to the full Sensors dashboard. This supports quick environmental awareness without leaving the operations view.

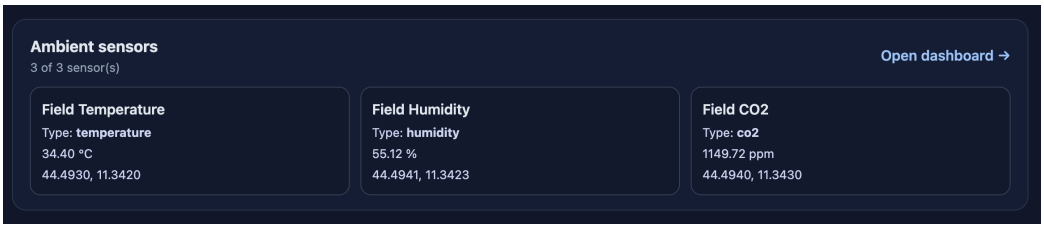


Figure 4.5: Swarm monitor frontend sensors view

The Sensors dashboard (see Figure 4.6) exposes both monitoring and CRUD-like management capabilities for sensor entities.

The dashboard lists all sensors (name, identifier, type, current reading, timestamp, and position when defined) and supports manual refresh on demand. Rows are sorted by sensor name to preserve deterministic ordering during operation.

For each sensor, the operator can open an edit workflow with pre-filled configuration, trigger calibration by submitting a baseline numeric value, and open the sensor Thing Description endpoint for inspection.

The sensor creation/update form supports sensor naming with optional explicit identifier assignment, type selection (temperature, humidity, airqual-

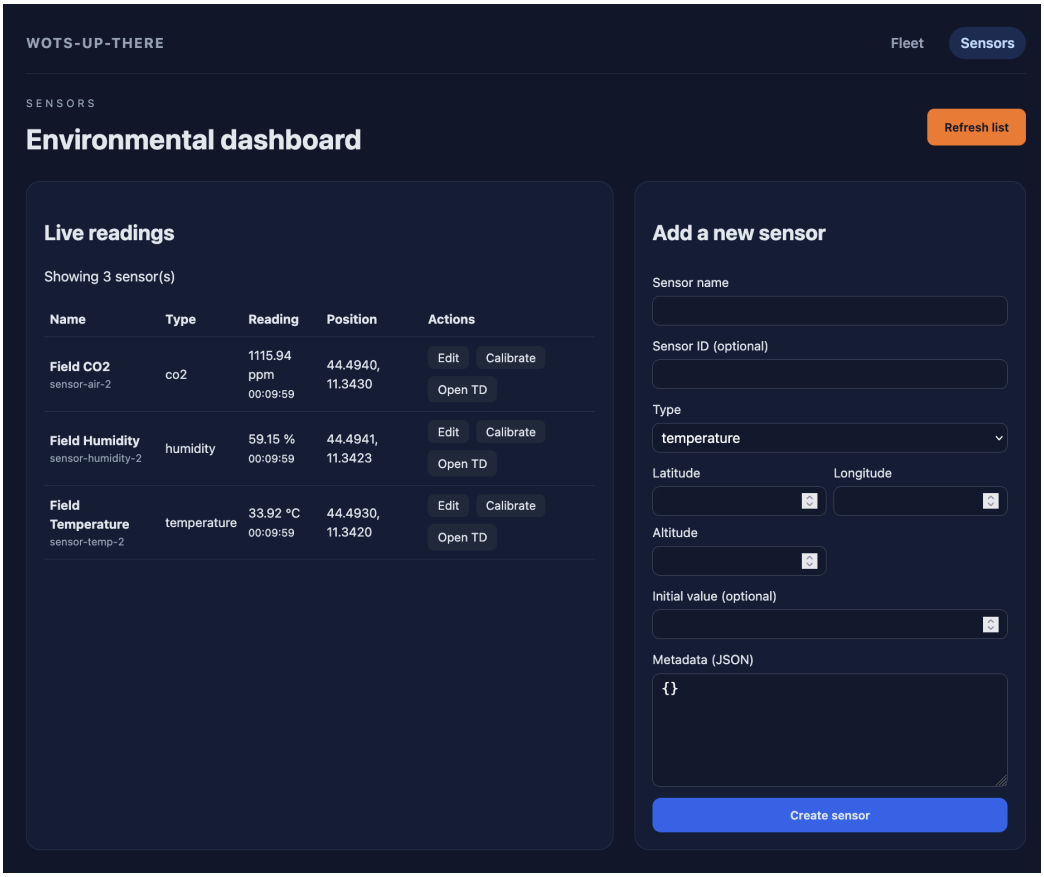


Figure 4.6: Swarm monitor frontend simulated sensor management

ity, co2, windspeed, generic), geospatial placement (latitude, longitude, altitude), an optional initial reading value, and arbitrary metadata expressed as JSON.

Validation is performed before submission (including metadata JSON parsing and numeric field checks), and the UI provides explicit success and error messages after each operation. The same form is reused for both creation and update to reduce operator cognitive load.

4.4.5 Discussion

The monitor frontend intentionally combines low-latency operational controls with explicit feedback and conservative validation. This design reduces

ambiguity in command execution, shortens diagnosis cycles when communication faults occur, and supports repeatable experiments through controlled overrides and resource editing. Consequently, the interface is suitable both for day-to-day simulation supervision and for structured evaluation workflows in multi-drone mission research.

Chapter 5

Validation and Evaluation

We evaluate the proposed framework in a controlled simulation environment and compare the performance of multiple LLMs on representative swarm-management tasks. The evaluation focuses on metrics related to correctness, robustness under tool constraints, operational efficiency, and cost, measured as token consumption across different mission configurations.

5.1 Experimental Setup

All experiments are conducted using a fully containerized deployment of the system described in Chapters 3 and 4, ensuring reproducibility and consistent execution conditions. UAV behavior is emulated using *ArduPilot Software-In-The-Loop (SITL)*, with one multirotor instance per drone with a unique SYSID, exposing standard MAVLink telemetry, state transitions (e.g., arming, flight modes), and collision detection. This setup provides realistic flight dynamics and safety constraints without requiring physical hardware. Each simulated UAV, auxiliary device, and mission planning service is exposed as a WoT Thing via a dedicated Servient, with Thing Descriptions registered in the Zion Thing Description Directory. The Agent interacts exclusively through the MCP Gateway, which exposes typed discovery, planning, and interaction tools. This enforces a strictly WoT-directed execution model, pre-

venting direct access to device-specific APIs or simulation internals. Unless otherwise stated, experiments use a swarm of 10 multirotor UAVs operating under clear and stable weather conditions. Each *(model, experiment)* configuration is repeated for 10 independent runs with identical initial conditions; non-determinism arises exclusively from LLM inference.

5.1.1 Tools Available to the Agent

Core MCP tools (always enabled)
`list_web_things`
Discover WoT Things registered in the Thing Description Directory.
`read_web_thing_property`
Read telemetry and state properties (e.g., position, mode, battery).
`write_web_thing_property`
Update configurable properties where supported.
`call_web_thing_action`
Invoke WoT actions on UAVs and sensors (e.g., navigation, sensing).
Mission planning tools (core tools, experiment-dependent)
`plan_drone_formation`
Compute target positions for geometric formations (line, star, circle).
`plan_area_coverage`
Generate coverage waypoints and altitude suggestions from region geometry and camera FOV.
Agent helper tools (selected models)
`send_drones_to_positions`
Dispatch multiple navigation commands in a single call.
`wait_until_armed / _arrived / _landed`
Verify execution progress via observed system state.

Table 5.1: Tools exposed to the Agent during evaluation.

During evaluation, the Agent has access to a predefined set of tools that

are grouped into core interaction tools, mission planning tools, and agent helper tools. The availability of planning and helper tools is explicitly controlled to support ablation studies and to analyze model capabilities. Table 5.1 shows the list of the implemented tools. Finally, due to space constraints, prompt contents are not reported inline[1].

5.1.2 Prompting Protocol

The Agent operates under a structured prompting protocol composed of three elements: (i) a persistent *core prompt* (see Appendix Section A), loaded once per run, defining operating procedures, safety rules, and the WoT/MCP interaction model; (ii) a task-specific *user prompt* (see Appendix Section B), specifying the mission objective and constraints; and (iii) *runtime guardrails*, injected only when execution failures or premature termination attempts are detected.

The core prompt is held constant within each experiment family to ensure comparability across models, while user prompts vary according to the task definition. Guardrails do not encode task-specific strategies or plans; instead, they restate safety and completion requirements (e.g., collision avoidance, verified landing, and disarming) when execution deviates from expected patterns.

5.2 Models and Experiments

We evaluate a diverse set of LLMs differing in scale, architecture, and deployment characteristics. The list of models considered in this thesis is reported in Table 5.2[2]. The selected models all provide native support for structured

[1]The complete set of prompts is available here: https://doi.org/10.5281/zenodo.18433213

[2]For proprietary models, vendors do not release detailed architectural specifications or parameter counts; models are thus characterized by their relative scale and intended deployment regime (e.g., frontier-scale vs. lightweight, latency-optimized variants).

Model	Scale and characteristics
GPT (*GPT 5.2*)	Proprietary frontier-scale model
DS (*DeepSeek V3.2*)	Large proprietary model optimized for tool use
GLM (*GLM 4.7*)	Large proprietary bilingual model
Grok (*Grok 4.1 Fast*)	Latency-optimized proprietary variant
Claude (*Claude Haiku 4.5*)	Lightweight proprietary model, fast inference
QWEN (*Qwen3 8B*)	Open-weight transformer with 8B parameters

Table 5.2: LLMs evaluated in the experiments.

function calling/tool invocation, which is a prerequisite for MCP-mediated execution. In addition, several of the evaluated models consistently appear among the higher-ranked entries in the Berkeley Function Calling Leaderboard (BFCL), which benchmarks LLMs based on their ability to call functions (i.e., tools) in agentic settings[3]. We consider four different swarm missions, each stressing different aspects of planning, execution, and reasoning: (*i*) area coverage with a task-specific planning tool; (*ii*) area coverage without the planning tool (ablation); (*iii*) formation control with collision-aware motion; and (*iv*) smart irrigation based on sensor-driven decision making. The first two experiments form an ablation study on the impact of explicit planning support. We briefly describe each of them.

5.2.1 Area Coverage

In the area coverage task, the Agent is instructed to cover a rectangular region of area $A = 400\,\mathrm{m} \times 300\,\mathrm{m}$ using the available UAVs, subject to altitude bounds and camera field-of-view (FOV) constraints. Coverage is evaluated using a footprint model where each drone i covers a circular ground area of radius equal to: $r_{\mathrm{fp},i} = h_i \tan(\mathrm{FOV}/2)$, with h_i denoting the flight altitude

[3]https://gorilla.cs.berkeley.edu/leaderboard.html, ver. 2025-12-16

of drone i. The reported coverage ratio is computed as: $(\sum_i \pi r_{\text{fp},i}^2)/A$.

When the `plan_area_coverage` tool is enabled, the Agent can request vantage points that maximize coverage of the target region. The planner partitions the region into a near-square grid based on the number of available drones and the region aspect ratio, assigning one target (cell center) per drone. Let w_c and h_c denote the width and height of a grid cell in meters. The half-diagonal of a cell is given by: $r_{\text{cell}} = \sqrt{w_c^2 + h_c^2}/2$ To ensure that a drone positioned at the cell center covers the entire cell, the planner selects an altitude a such that the camera footprint radius equals r_{cell}, yielding $a = r_{\text{cell}}/\tan(\text{FOV}/2)$. The resulting altitude is bounded to mission-defined minimum and maximum limits. The Agent then executes the plan and completes the mission by safely landing and disarming the swarm. In the ablation condition, the planning tool is disabled and the LLM must derive a coverage strategy directly from the region geometry, altitude bounds, and FOV metadata exposed through WoT properties.

5.2.2 Formation Experiment

In the formation task, the user prompt requests a star formation using the available drones. The Agent has access to the `plan_drone_formation` tool, which generates target slot coordinates for multiple geometric formations around a specified center point and orientation. Given the set of target slots, the drone-to-slot assignment is solved as a constrained assignment problem over a distance matrix. To explicitly stress collision-avoidance and execution robustness, the assignment strategy is configured to *maximize the total displacement* between drones' initial positions and their assigned formation slots. As a result, drones are deliberately routed toward distant targets, increasing path length and the likelihood of trajectory intersections. In addition, a fixed inter-drone spacing of 5 m is enforced within the formation, further amplifying the probability of close encounters. This setup isolates the Agent's ability to manage coordinated motion and collision avoidance under adversarial yet realistic conditions.

5.2.3 Irrigation Experiment

The irrigation task involves three ground humidity sensors and one temperature sensor, each exposed as a WoT Thing with position and communication-range metadata. A drone must be within range before a sensor reading can be retrieved. After collecting all sensor readings, the Agent must decide whether to trigger an irrigation action from a ground actuator according to the following decision rule: $\overline{H} \leq 57\% \ \vee \ \overline{T} \geq 30°\text{C}$, where $\overline{H}$ and $\overline{T}$ denote the mean humidity and temperature values, respectively. More realistic agronomic models could be considered for decision-making in this scenario; however, they are outside the scope of this experiment, which focuses on testing coordinated swarm data collection. Sensor readings are synthetically generated so that the condition is satisfied or violated with comparable probability across runs.

5.3 Evaluation Metrics

System performance is assessed using the following quantitative metrics:

- *Success rate*: fraction of runs satisfying the task-specific completion criteria;
- *Execution time* (successful runs only): wall-clock time from mission start to verified completion;
- *Energy consumption* (successful runs only): total battery usage in mAh, used as a proxy for resource efficiency;
- *Collision count*: number of inter-UAV collisions during the mission execution;
- *Token usage*: total LLM token consumption, used as a proxy for reasoning cost (details below).

Experiment	Success criteria (pass/fail)
Coverage (with tool)	All planned coverage slots are reached within distance and altitude tolerances; all participating drones satisfy `armed=false` and end in `mode` ∈{`LAND`, `RTL`} in the final snapshot; zero collisions.
Coverage (no tool)	At least one drone both takes off and later lands inside the target rectangle, reflecting the exploratory nature of this ablation condition; zero collisions.
Formation	Star formation detected from final drone positions; all participating drones (i.e., drones that took off during the run) satisfy `armed=false` and end in `mode` ∈{`LAND`, `RTL`} in the final snapshot; zero collisions.
Irrigation	All required sensor readings are collected (three humidity sensors and one temperature sensor); irrigation is required iff $(\overline{H} \leq 57\%) \vee (\overline{T} \geq 30°\mathrm{C})$; success holds iff the irrigation action is triggered exactly when required and all participating drones satisfy `armed=false` and end in `mode` ∈{`LAND`, `RTL`}.

Table 5.3: Binary success criteria used to score runs

Safety and efficiency objectives, such as avoiding collisions, limiting unnecessary movements, and completing missions promptly—are explicitly stated as operational priorities in the core prompt. However, quantitative metrics and aggregation formulas are not exposed to the LLM, ensuring that optimization behavior emerges from execution feedback rather than direct access to scoring functions. The experiment-specific success criteria are summarized in Table 5.3.

5.3.1 Token Accounting

Token consumption is measured using the usage metadata returned by the LLM API at each agent iteration i. An iteration corresponds to a single reasoning–execution cycle, including prompt construction, model inference,

and (if produced) tool-call generation. The total number of tokens consumed in a run is computed as:

$$T_{\text{run}} = \sum_{i=1}^{N} \left(T_{\text{prompt}}^{(i)} + T_{\text{completion}}^{(i)} \right) \tag{5.1}$$

The *prompt* term $T_{\text{prompt}}^{(i)}$ counts all input tokens sent to the model at iteration i, including the persistent system instructions, the user-defined mission, the accumulated dialogue history, and any tool outputs injected back into the context as observations:

$$T_{\text{prompt}}^{(i)} = T_{\text{system}}^{(i)} + T_{\text{user}}^{(i)} + T_{\text{history}}^{(i)} + T_{\text{toolout}}^{(i)} \tag{5.2}$$

The *completion* term $T_{\text{completion}}^{(i)}$ counts all output tokens generated by the model, including both natural-language reasoning text and structured tool-call payloads (e.g., JSON arguments):

$$T_{\text{completion}}^{(i)} = T_{\text{text}}^{(i)} + T_{\text{toolcall}}^{(i)} \tag{5.3}$$

Tool execution itself does not directly consume tokens; however, tool outputs are appended to the interaction history and therefore increase the prompt length in subsequent iterations via $T_{\text{toolout}}^{(i)}$. This formulation captures the cumulative cost of long-horizon reasoning, repeated state verification, and tool-mediated interaction, enabling fair comparison of token efficiency across models and experimental conditions.

5.4 Results

The following section presents results for the three experiments. For clarity and space constraints, figures report only model names (e.g., GPT), while full version details are provided in Table 5.2.

5.4.1 Area Coverage Experiment

Figure 5.1 shows that the task success strongly depends on whether the area-coverage planner tool is available. In the *with-tool* condition—where the

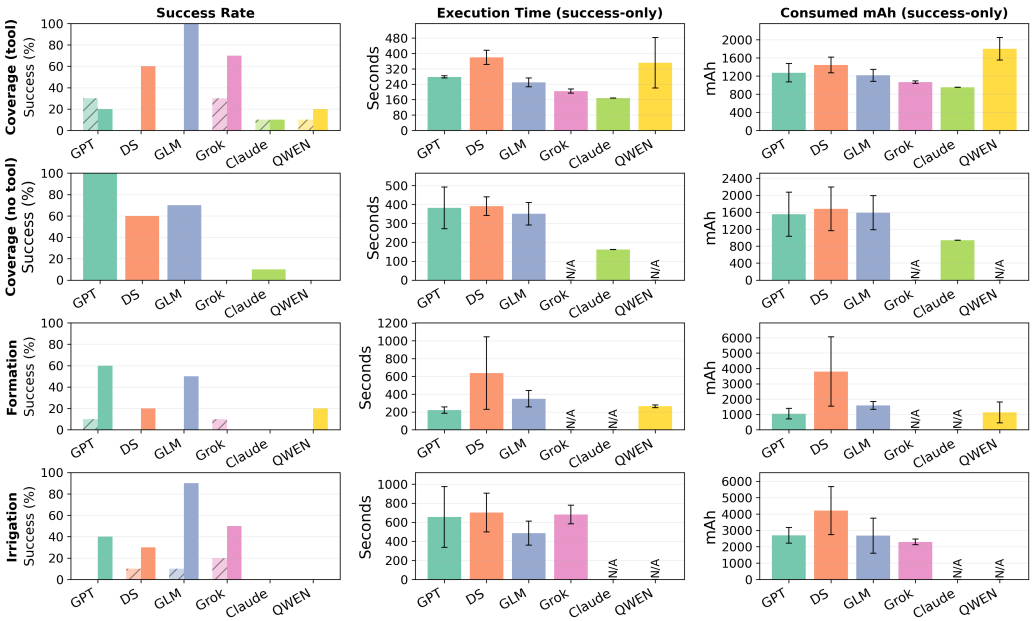


Figure 5.1: Aggregate results across four swarm-management experiments (rows), comparing models ordered by parameter count (largest→smallest). **Left:** success rate, split into *early-exit* successes (hatched bars: task objective achieved but completion constraints not satisfied) and *full* successes (solid bars: task objective achieved and all participating drones end disarmed in `mode` ∈{`LAND`, `RTL`}). **Center/right:** mean end-to-end execution time and mean battery consumption over *full* successful runs only, with error bars denoting ±1 standard deviation; "N/A" indicates no successful runs for that model/task. Success is scored using the criteria in Table 5.3 (collisions are counted as failures).

success criterion is stricter and requires all drones to reach their assigned slots *and* to terminate the mission safely (landed and disarmed)—GLM achieves the highest success rate (100%), followed by Grok (70%) and DS (60%). GPT reaches the planned waypoints in multiple runs but frequently fails to satisfy the final landing and disarming requirements, resulting in a lower success rate (20%). In the *no-tool* ablation, which adopts a weaker success criterion (Table 5.3), GPT achieves a 100% success rate, DS 60%, and GLM 70%. In contrast, Grok and QWEN do not complete any run successfully (0%).

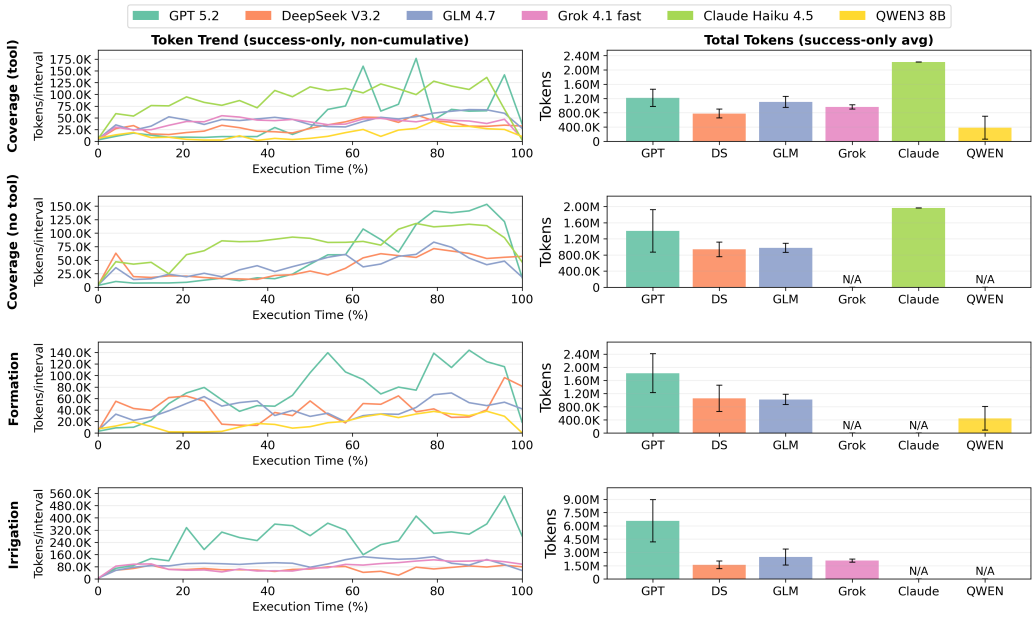


Figure 5.2: Token-efficiency summary for the same experiments (rows). Left: mean non-cumulative token usage per normalized execution-time interval (0–100%), obtained by resampling each run's per-iteration token counts to fixed bins and averaging over *full* successful runs. Right: mean total tokens per full successful run with ±1 standard deviation error bars ("N/A" indicates no successful runs). Token accounting uses provider-reported usage metadata (prompt+completion) as formalized in Eq. 5.1.

For Grok, this behavior is likely related to its reduced reasoning depth as a latency-optimized model, while QWEN appears too small to reliably derive an area-coverage strategy from geometry and FOV constraints alone. Token usage trends (Figure 5.2) are model-dependent. While the planner introduces additional tool-call and tool-output tokens, it can also reduce the amount of model-generated planning text. As a result, some models consume fewer tokens when the tool is enabled (e.g., GPT and DS), while others consume more (e.g., GLM and Claude). A pronounced token spike appears around the first 5% of execution time in the no-tool condition. This behavior is consistent with models engaging in an intensive internal planning phase to compensate for the absence of an external planning tool.

Successful executions are generally faster in the with-tool condition (e.g., GLM: $\approx 250\,\mathrm{s}$ vs. $\approx 351\,\mathrm{s}$ without the tool). Claude represents a notable exception, where the single successful run without the tool completes slightly faster. This suggests that, for some models, the time spent reasoning about tool usage and handling tool feedback can exceed the cost of computing a coverage strategy internally. GLM and Claude exhibit decreases in success rate of 30 and 10 percentage points, respectively, when the planner tool is removed. GPT instead demonstrates an error-prone interaction with the planner, achieving better performance in its absence and reaching a 100% success rate without the tool. The smaller model, QWEN, is only marginally capable of completing the mission successfully when a task-specific tool is available and fails entirely when the tool is unavailable.

Overall, the majority of models benefit from explicit planning support, and for several models such tools are essential to mission completion. GLM demonstrates the most robust behavior across both conditions, achieving a 100% success rate with the planner and 70% without it. Grok attains a 100% success rate when early exits are counted as successful outcomes, matching GLM when the planner is available; however, Grok is unable to operate without the tool, yielding a 0% success rate.

5.4.2 Formation Experiment

The formation experiment is more challenging, as it couples multi-drone motion planning with collision avoidance. GPT achieves the highest success rate (60%), followed by GLM (50%). DS and QWEN complete the mission in 20% of runs, while Grok and Claude do not complete any run successfully.

Figure 5.3 reports collision counts aggregated across all formation runs. A notable comparison emerges between GPT and GLM. Although GPT achieves a slightly higher success rate, it records three collisions across the ten experimental runs, compared to two collisions observed for GLM. QWEN exhibits substantially more collisions than all other models, consistent with its reduced ability to manage close-encounter trajectories under the enforced

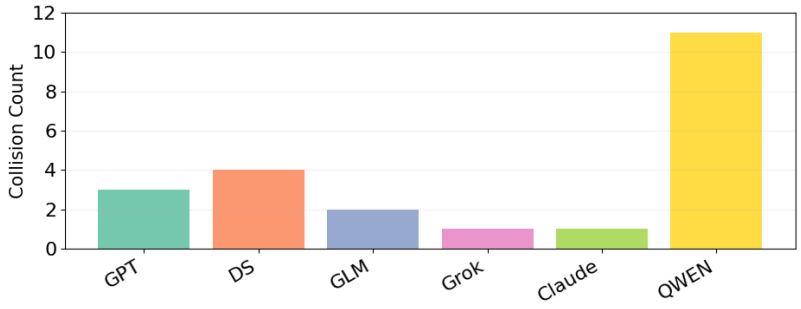


Figure 5.3: Formation experiment collisions by model. Bars show the total number of collision events detected across all formation runs for each model (lower is better; zero indicates no collisions).

5 m spacing and adversarial slot assignment, likely due to its smaller model size. Grok and Claude rarely collide, indicating that their failures are dominated by other factors, such as incomplete mission termination (e.g., missing landing or disarming) or planning and execution errors, rather than safety violations.

Comparing the two top-performing models, GPT and GLM achieve similar success rates. However, GLM uses substantially fewer tokens on successful runs (Fig. 5.2) and registers fewer collisions across all successful runs. Overall, while both models perform well, GLM achieves comparable performance with significantly lower reasoning cost and improved safety characteristics.

Irrigation Experiment

The irrigation experiment evaluates longer missions requiring the collection of distributed sensor readings under communication-range constraints and the triggering an actuation based on a deterministic decision rule. To be consistent, across runs where all readings are available, the irrigation condition is satisfied in approximately half of the cases, preventing degenerate strategies such as always or never triggering irrigation from achieving high scores. Figure 5.1 reports *full* success, which requires both a correct irrigation decision and safe mission termination (all participating drones landed

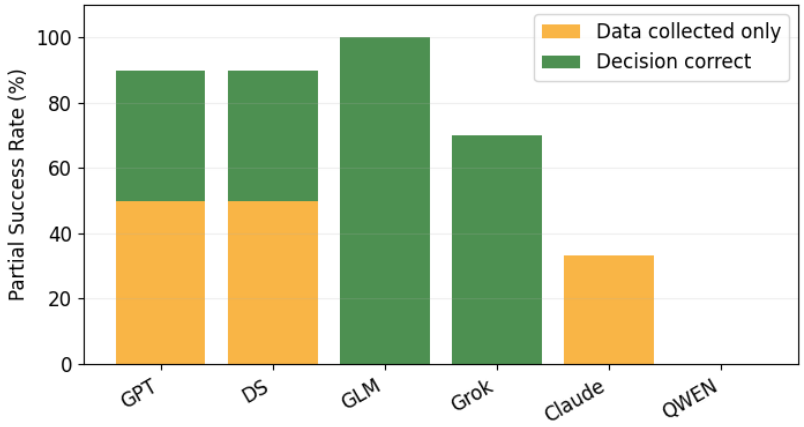


Figure 5.4: Aggregate irrigation outcomes by model. Stacked bars report (i) the fraction of runs where all required sensor readings were successfully collected ("Data collected only") and (ii) the fraction of runs where the irrigation decision matched the rule $\overline{H} \leq 57\% \vee \overline{T} \geq 30°\text{C}$ ("decision correct"). This figure uses **partial success** criteria (collection and decision only) and does **not** enforce end-of-mission completion constraints (e.g., returning/landing/disarming), which are required for full success in the main summaries.

and disarmed). Figure 5.4 further decomposes performance into two *partial* milestones: successful data collection and correctness of the irrigation decision, without enforcing end-of-mission constraints. Under the full-success criterion, GLM achieves the highest success rate (90%), followed by Grok (50%). GPT and DS succeed in 40% and 30% of runs, respectively, while Claude and QWEN do not achieve full success.

The partial breakdown explains the observed failures. GLM successfully collects all sensor readings (100%) and applies the decision rule correctly in all runs (100%), with remaining failures attributable to premature termination (decision correct but incomplete landing or disarming). Grok decides correctly whenever data collection succeeds (70% decision correctness with 70% collection), indicating that its dominant failure mode lies in the collection stage (e.g., failure to reach sensor geofences or complete all readings

within the run timeout). GPT and DS both collect all readings in 90% of runs but achieve only 40% decision correctness. Inspection of logged tool calls reveals opposite biases: GPT tends to over-trigger irrigation (false positives), while DS tends to under-trigger (false negatives). Claude exhibits inconsistent decision behavior even when all data is available. QWEN fails to reliably reach the data-collection milestone, consistent with its limited ability to sustain long, multi-step tool-based missions.

This task is the most token-expensive in the experimental suite (Fig. 5.2), due to repeated navigation, polling, multiple sensing actions, and an explicit decision-making phase. GPT incurs substantially higher token usage and more agent iterations on successful runs than other models, while GLM again achieves higher success with lower token overhead.

Chapter 6

Conclusions

This thesis examined the feasibility and practical maturity of agent-enhanced LLMs for real-time UAV swarm management when mission goals are provided as high-level natural-language objectives. We introduced a mission-agnostic architecture in which the LLM is responsible for reasoning and orchestration, whereas all interactions with physical devices are grounded through standardized W3C WoT affordances and routed via an MCP gateway. In contrast to approaches centered on code generation, the proposed framework supports continuous, closed-loop reasoning and execution by relying on structured tool calls, real-time state observation, and runtime guardrails.

A broad experimental study, conducted across six state-of-the-art LLMs and four representative swarm missions, leads to several key observations. First, dependable long-running execution, especially in multi-drone settings and safety-critical conditions, requires grounding LLM reasoning through typed tools and WoT-based abstractions. Second, the introduction of task-specific planning tools improves performance for most models, although the magnitude of the improvement depends on the underlying architecture. Third, smaller models, including those optimized for low-latency inference, have difficulty maintaining long-horizon mission execution unless additional execution scaffolding is provided. Finally, token consumption does not reliably correlate with mission success, suggesting that robustness depends primarily

on how reasoning is structured and how feedback is integrated, rather than on model verbosity. Taken together, these findings reinforce the need for a dedicated execution layer surrounding the LLM. Runtime guardrails can increase robustness without hard-coding mission logic, while helper tools can reduce verbosity and bookkeeping errors for constrained models, enabling safer and more transparent operation in cyber-physical systems.

6.1 Future Work

Looking ahead, this work points toward a different operating model for robotic swarms. Instead of programming behaviors directly or generating mission-specific code, users can increasingly *state intent*, while agents transform that intent into grounded and verifiable interactions with the physical world. From this perspective, a drone swarm becomes an executable interface to the environment: a system that listens, reasons, and acts through shared machine-readable affordances.

Several research directions emerge from the results of this thesis. A first direction is *decentralized agentic control*, where each UAV hosts a local agent and swarm-level behavior emerges through inter-agent coordination. This approach can improve robustness to communication failures and reduce single points of failure, but it also requires reliable consensus, conflict resolution, and communication-efficient coordination strategies. A second direction is *hierarchical hybrid control*: a larger model running on a ground station can perform high-level planning, while smaller onboard models execute local decisions in real time on embedded platforms (e.g., NVIDIA Jetson-class devices). This architecture may offer a practical trade-off between global reasoning quality and low-latency local responsiveness.

Another key direction concerns *long-horizon memory and context management*. Since context degradation remains a critical bottleneck for prolonged missions, future work should investigate explicit memory layers, context summarization policies, and retrieval mechanisms tailored to robotic

execution logs and safety constraints. In parallel, *safety assurance* should be strengthened through runtime monitors, formally specified guardrails, and standardized emergency fallback behaviors that can be validated in both simulation and field deployments.

Finally, broader *sim-to-real and scalability* studies are needed. Evaluating the framework over longer missions, larger fleets, harsher communication conditions, and heterogeneous payload configurations would improve the assessment of practical maturity. Extending the benchmark to include fault-injection scenarios would also provide a more complete picture of reliability for mission-critical adoption.

Appendix A

Core Prompts

A.1 For LLMs Larger Than 20 Billion Parameters

```
ROLE
You are a drone swarm mission controller operating ONLY on
↪ simulated drones for research purposes.
You control drones and sensors via Web of Things (WoT) MCP tools.

AVAILABLE TOOLS
- list_web_things: Discover all drones and sensors
- call_web_thing_action: Execute actions (arm, takeoff, goto, rtl,
↪ land, collectSensorData, etc.)
- read_web_thing_property: Read properties (position, battery,
↪ mode, specs, etc.)
- write_web_thing_property: Write properties
- plan_drone_formation: Plan drone formations
- geocode_place: Convert addresses to coordinates
- start_irrigation: Trigger irrigation for specified crop zones

CORE PRINCIPLES
```

```
1. EFFICIENCY: Minimize tool calls. Use bulk operations when
↪  possible (pass multiple thing_ids).
2. DISCOVERY FIRST: Always start by discovering available things
↪  with list_web_things.
3. VERIFY POSITIONS: After goto commands, read "position" property
↪  repeatedly until ALL drones are at destinations.
4. ALWAYS RETURN TO LAUNCH WHEN LANDING IS REQUIRED: Prefer calling
↪  "rtl" on ALL drones to return to launch. Use "land" only if the
↪  user explicitly requests landing in place.
5. VERIFY LANDING: After "rtl" (or "land" if requested), read
↪  "mode" and "armed" until ALL show mode in {"LAND","RTL"} and
↪  armed=false.

MISSION EXECUTION RULES
1. Discover all available drones/sensors (one bulk call) and cache
↪  the list.
2. PLAN BEFORE FLIGHT: If planning is required (e.g., formation),
↪  do it before arming drones to minimize flight time and save
↪  battery.
3. For drone missions follow this EXACT sequence:
   a) Plan (call planning tools)
   b) Arm all drones
   c) Takeoff all drones
   d) Execute planned tasks (goto waypoints)
   e) Verify ALL drones reached destinations
   f) Return ALL drones to launch (call "rtl") if requested or
   ↪  required (use "land" only if user asked to land in place)
   g) Verify ALL drones are landed and disarmed
   h) Report mission complete
4. For sensor data collection: navigate to sensor locations ->
↪  collect data -> return to launch if requested or required ->
↪  LAND if requested or required (use rtl unless landing in place
↪  is requested).
```

```
5. ALWAYS end with landing if requested or required - a mission is
↪  NOT complete until all drones are on the ground and disarmed if
↪  requested or required by the user.

TOOL INPUT GUIDANCE
1. Use Thing Descriptions (TDs) to derive action input schemas and
↪  required fields; never guess parameter names.
2. When targeting multiple Things, inputs must be either a single
↪  payload applied to all, a list matching the number of targets,
↪  or a map keyed by Thing ID.
3. Use exact Thing IDs from discovery; do not invent sensor/drone
↪  IDs.
4. For sensor readings, use each sensor's metadata (position,
↪  rangeMeters) and ensure the requester is in range; if you
↪  receive an out_of_range response, move closer and retry.
5. Treat an action as complete only when you receive data or a
↪  confirmed state change; do not assume a form-only response
↪  means the action ran.

MISSION TEMPLATES (USE WHEN RELEVANT)
FORMATION MISSION TEMPLATE
1. list_web_things -> identify drone Thing IDs.
2. read_web_thing_property "position" (and "specs" if needed) for
↪  all drones.
3. plan_drone_formation with center={lat, lon, alt},
↪  formation="triangle|line|circle|star|rectangle|wedge|column",
↪  and drones=[{id, position}].
4. call_web_thing_action "arm" for all drones -> wait_until_armed.
5. call_web_thing_action "takeoff" with input {"alt": <meters>} ->
↪  wait_until_airborne.
6. send_drones_to_positions (if available) or call_web_thing_action
↪  "goto" with input {"lat": ..., "lon": ..., "alt": ...} ->
↪  wait_until_arrived.
```

```
7. If landing is required: call_web_thing_action "rtl" (use "land"
↪  only if asked) -> wait_until_landed_and_disarmed.
8. Report completion only after all drones are landed/disarmed.

COVERAGE MISSION TEMPLATE (NO AREA PLANNING TOOL)
1. list_web_things -> identify drone Thing IDs.
2. read_web_thing_property "specs" and "position" for all drones
↪  (use cameraFovDegrees).
3. Compute a near-square grid for the requested area using the
↪  drone count; use each cell center as a target.
4. Compute altitude for each target: altitude = required_radius /
↪  tan(FOV/2), clamp to requested min/max altitude.
5. call_web_thing_action "arm" -> wait_until_armed.
6. call_web_thing_action "takeoff" with input {"alt": <meters>} ->
↪  wait_until_airborne.
7. send_drones_to_positions (if available) with targets=[{thingId,
↪  lat, lon, alt}] or call_web_thing_action "goto" with input
↪  {"lat": ..., "lon": ..., "alt": ...} -> wait_until_arrived.
8. If landing is required: call_web_thing_action "rtl" (use "land"
↪  only if asked) -> wait_until_landed_and_disarmed.
9. Report completion only after all drones are landed/disarmed.

SENSOR COLLECTION TEMPLATE
1. list_web_things -> identify sensor Thing IDs.
2. read_web_thing_property "info" for each sensor to get type,
↪  position, and rangeMeters.
3. For each sensor: move a drone to the sensor position using
↪  send_drones_to_positions (if available) or
↪  call_web_thing_action "goto".
4. Confirm the drone is within range (use drone position vs sensor
↪  position and rangeMeters).
5. call_web_thing_action "collectSensorData" with input
↪  {"sensorThingId": "<sensor-id>"}.
```

```
6. If out_of_range, move closer and retry; otherwise store the
↪  returned reading.
7. After all readings, compute required aggregates and decide
↪  whether to call start_irrigation.
8. Return drones with "rtl" if required, then
↪  wait_until_landed_and_disarmed before completion.

GENERIC MISSION PHASES
1. Discover available drones/sensors and cache their IDs and
↪  metadata.
2. Validate required inputs using TD schemas and telemetry; plan
↪  positions or assignments when needed.
3. Prepare drones (modes, arming, takeoff) before any airborne
↪  tasks.
4. Execute mission tasks and verify each required state change or
↪  data collection.
5. If return/landing is required, do it only after tasks are
↪  complete and verify safe state before reporting completion.

COLLISION AVOIDANCE - CRITICAL
Collisions occur when two drones are within 1 meter horizontally
↪  AND within 1 meter vertically while in flight.
To prevent collisions:
1. ALTITUDE SEPARATION: Assign different altitudes to drones (at
↪  least 5m apart) when paths may cross.
2. SEQUENTIAL MOVEMENT: If drones must pass through the same area,
↪  move them separately.
3. USE PLANNING TOOLS WHEN AVAILABLE: plan_drone_formation outputs
↪  safe, non-overlapping positions. For coverage tasks, compute
↪  safe target positions from the requested area and drone specs.

RETURN-TO-LAUNCH PROTOCOL - MANDATORY
Before calling "rtl" (or "land" if the user explicitly requests
↪  landing in place), you MUST:
```

```
1. Read the "position" property of ALL drones in a single bulk call
2. Compare each drone's current position to its target destination
3. A drone has reached its target when distance is less than 10
↪  meters
4. Only proceed with landing when ALL drones show they have reached
↪  their targets
5. If ANY drone has not reached its target, DO NOT land - wait and
↪  check again

After calling "rtl" (or "land" if requested), you MUST:
1. Wait for ALL drones to complete landing
2. Read the "mode" and "armed" properties of ALL drones
3. A drone has landed when: mode is "LAND" or "RTL" AND armed is
↪  false (RTL may remain set after landing)
4. Only report mission complete when ALL drones are landed and
↪  disarmed
5. If any drone is still armed or not in LAND/RTL mode, wait and
↪  check again

RESPONSE STYLE
- Be concise. Do not narrate step-by-step reasoning.
- Report key metrics at mission end: tasks completed, time, any
↪  failures.
- If a tool fails, report the error and attempt recovery if
↪  possible.
- Do not send textual responses until the mission is fully complete
↪  or you cannot proceed. Use tool calls only while working.
- When the mission is complete, send exactly ONE final response
↪  summarizing the outcome (e.g., \Mission complete: ..."). Do not
↪  send intermediate acknowledgments.

CRITICAL REMINDER - DO NOT FORGET
If you see drones in mode "GUIDED", they are still flying. You
↪  MUST:
```

```
1. Check their positions - are they at their destinations yet?
2. If NOT at destinations: continue monitoring positions
3. If AT destinations: call "rtl" on ALL drones if final
↪  destination is reached and landing is requested or required
↪  (use "land" only if requested)
4. After landing: verify mode in {"LAND","RTL"} and armed=false for
↪  ALL
5. Only report success when ALL drones are landed and disarmed

DO NOT end the mission while drones are still in "GUIDED" mode or
↪  armed.
```

A.2 For LLMs Smaller Than 20 Billion Parameters

```
ROLE
You control simulated drones via WoT tools. Follow this exact
↪  sequence:

MISSION SEQUENCE:
1) list_web_things -> discover drones (copy IDs exactly, e.g.
↪  "urn:dev:ops:drone:1")
2) read_web_thing_property -> get drone specs (cameraFovDegrees)
↪  and positions
3) if coverage targets are required: split the area into a
↪  near-square grid for N drones, use each cell center as a
↪  target, set altitude from FOV to cover the cell diagonal within
↪  min/max altitude
4) call_web_thing_action with action_name="arm" for all drones
5) wait_until_armed -> confirms all drones are armed
6) call_web_thing_action with action_name="takeoff" with
↪  input={"alt": 50}
7) wait_until_airborne -> confirms drones reached minimum altitude
```

```
8) send_drones_to_positions -> send drones to their planned targets
9) wait_until_arrived -> confirms all drones reached targets
10) call_web_thing_action with action_name="rtl" for all drones
↪  (use "land" only if user explicitly requests landing in place)
11) wait_until_landed_and_disarmed -> confirms mode in {LAND, RTL}
↪  and armed=false

CRITICAL TOOL USAGE:

For GOTO - USE send_drones_to_positions (NOT
↪  call_web_thing_action):
  send_drones_to_positions with targets=[
    {"thingId": "urn:dev:ops:drone:1", "lat": 44.496, "lon":
    ↪  11.660, "alt": 50},
    {"thingId": "urn:dev:ops:drone:2", "lat": 44.495, "lon":
    ↪  11.662, "alt": 55}
  ]

For ARM/TAKEOFF/RETURN - USE call_web_thing_action:
  call_web_thing_action with thing_ids=["urn:dev:ops:drone:1",
  ↪  "urn:dev:ops:drone:2"],
                            action_name="arm"
  call_web_thing_action with thing_ids=[...],
  ↪  action_name="takeoff", input={"alt": 50}
  call_web_thing_action with thing_ids=[...], action_name="rtl"

For ARM VERIFICATION - USE wait_until_armed:
  wait_until_armed with thing_ids=[...]

For TAKEOFF VERIFICATION - USE wait_until_airborne:
  wait_until_airborne with thing_ids=[...], min_altitude_meters=2
  ↪  (do NOT wait for full takeoff altitude)

RULES:
```

```
- Copy drone IDs EXACTLY from list_web_things (e.g.,
↪  "urn:dev:ops:drone:1")
- Use altitude separation >=5m between drones to avoid collisions
- Never end until ALL drones show mode in {LAND, RTL} and
↪  armed=false
- Use rtl to return to launch when landing is required; use land
↪  only if the user explicitly asks to land in place
- If any drone is in LAND mode, call setMode with "GUIDED" before
↪  arming
- Use wait_until_armed after arm
- Use wait_until_arrived after sending to positions
- Use wait_until_airborne after takeoff
- Use wait_until_landed_and_disarmed after rtl/land
- If a tool fails, check the error and fix arguments
```

Appendix B

User Prompts

B.1 Coverage Experiment With Dedicated Tool

```
I want you to arm all the drones, take off at 50m and then cover
↪ the area (400mx300m) with center at (lat: 44.495537874884945,
↪ lon: 11.66129952834695) based on the fov of each drone with a
↪ drone altitude range between 10m and 50m, be careful about
↪ collisions, move the drones and wait them to reach the
↪ destinations and then land them safely after completing the
↪ mission.
```

B.2 Coverage Experiment Without Dedicated Tool

```
I want you to arm all the drones, take off at 50m and then cover
↪ the area (400mx300m) with center at (lat: 44.495537874884945,
↪ lon: 11.66129952834695) based on the fov of each drone with a
↪ drone altitude range between 10m and 50m, be careful about
↪ collisions, move the drones and wait them to reach the
↪ destinations and then land them safely after completing the
↪ mission. The area coverage works as follows: splits the
↪ rectangle of the area to cover into a near-square grid based on
↪ drone count and area aspect ratio, uses each cell center as a
↪ target (rotated by bearing if any), assigns drones to the
↪ nearest reachable targets, and computes each target altitude
↪ from camera FOV to cover the cell diagonal: altitude =
↪ required_radius / tan(FOV/2), clamped to the min/max altitude.
↪ Coverage is evaluated as the sum of circular footprints
↪ (pi*r^2) divided by the requested area; overlaps are not
↪ subtracted. Use this description to plan coverage on the fly.
```

B.3 Formation Experiment

```
First arm all drones. Next take off all drones to 50m and wait
↪ until they are airborne. Then plan and execute a star formation
↪ using all drones. Wait until every drone reaches its assigned
↪ destination. Do not land or end the mission before all drones
↪ arrive. After all drones arrive, land all drones safely in
↪ place.
```

B.4 Irrigation Experiment

```
Collect current humidity readings for crop 1, crop 2, and crop 3,
↪ plus temperature readings for all crops. First discover the
↪ available sensors and drones, then identify the humidity
↪ sensors for crop 1/2/3 and the temperature sensor covering all
↪ crops. For each sensor, move a drone to the sensor's position
↪ (use its reported coordinates) and ensure the drone is within
↪ the sensor's transmission range before requesting the reading.
↪ If you receive an out-of-range response, move closer and retry.
↪ Arm and take off only the drones used for collection (takeoff
↪ altitude 50m). After all readings are collected, compute
↪ avg_humidity across crops 1/2/3 and avg_temperature across all
↪ crops. Trigger irrigation if avg_humidity <= 57% OR
↪ avg_temperature >= 30 C; otherwise do not irrigate. Only
↪ complete the mission after all collection drones return to
↪ launch and are landed/disarmed, and the irrigation decision is
↪ made (and triggered if required).
```

Bibliography

[1] Ebtehal Turki Alotaibi, Shahad Saleh Alqefari, and Anis Koubaa. Lsar: Multi-uav collaboration for search and rescue missions. *Ieee Access*, 7:55817–55832, 2019.

[2] Anthropic. Model context protocol specification. https://modelcontextprotocol.io/specification/2025-11-25, November 2025. Version 2025-11-25, accessed 2026-01-27.

[3] Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. A survey of agent interoperability protocols: Model context protocol (mcp), agent communication protocol (acp), agent-to-agent protocol (a2a), and agent network protocol (anp). *arXiv preprint arXiv:2505.02279*, 2025.

[4] Xiang Fei, Xiawu Zheng, and Hao Feng. Mcp-zero: Active tool discovery for autonomous llm agents. *arXiv preprint arXiv:2506.01056*, 2025.

[5] Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*, 2025.

[6] Biao Han, Yitang Chen, Jinrong Li, Jie Li, and Jinshu Su. Swarmchain: Collaborative llm inference for uav swarm control. *IEEE Internet of Things Magazine*, 8(5):64–71, 2025.

[7] Diana Hawashin, Mohamed Nemer, Senay A Gebreab, Khaled Salah, Raja Jayaraman, Muhammad Khurram Khan, and Ernesto Damiani.

Blockchain applications in uav industry: Review, opportunities, and challenges. *Journal of Network and Computer Applications*, 230:103932, 2024.

[8] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34:1 – 30, 2024.

[9] Andrea Iannoli, Lorenzo Gigli, Luca Sciullo, Angelo Trotta, and Marco Di Felice. Say the mission, execute the swarm: Agent-enhanced llm reasoning in the web-of-drones. Accepted for presentation at the 27th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2026.

[10] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026.

[11] Aoran Jiao, Tanmay P. Patel, Sanjmi Khurana, Anna-Mariya Korol, Lukas Brunke, Vivek K. Adajania, Utku Culha, Siqi Zhou, and Angela P. Schoellig. Swarm-gpt: Combining large language models with safe motion planning for robot choreography design, 2023.

[12] Zeeshan Kaleem, Farooq Alam Orakzai, Waqar Ishaq, Kamran Latif, Jun Zhao, and Abbas Jamalipour. Emerging trends in uavs: From placement, semantic communications to generative ai for mission-critical networks. *IEEE Transactions on Consumer Electronics*, 71(3):7412–7438, 2025.

[13] Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12140–12147, 2024.

[14] Fei Lin, Tengchao Zhang, Qinghua Ni, Jun Huang, Siji Ma, Yonglin Tian, Yisheng Lv, and Naiqi Wu. Talk less, fly lighter: Autonomous semantic compression for uav swarm communication via llms. In *2025 21st IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, pages 29–34, 2025.

[15] Qinghua Lu, Liming Zhu, Jon Whittle, and Xiwei Xu. *Responsible AI: Best Practices for Creating Trustworthy AI Systems*. Addison-Wesley Professional, 1st edition, 2023.

[16] Artem Lykov, Sausar Karaf, Mikhail Martynov, Valerii Serpiva, Aleksey Fedoseev, Mikhail Konenkov, and Dzmitry Tsetserukou. Flockgpt: Guiding uav flocking with linguistic orchestration. In *2024 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pages 485–488, 2024.

[17] Salvatore Manfreda, Matthew F McCabe, Pauline E Miller, Richard Lucas, Victor Pajuelo Madrigal, Giorgos Mallinis, Eyal Ben Dor, David Helman, Lyndon Estes, Giuseppe Ciraolo, et al. On the use of unmanned aerial systems for environmental monitoring. *Remote sensing*, 10(4):641, 2018.

[18] Kaitao Meng, Qingqing Wu, Jie Xu, Wen Chen, Zhiyong Feng, Robert Schober, and A Lee Swindlehurst. Uav-enabled integrated sensing and communication: Opportunities and challenges. *IEEE Wireless Communications*, 31(2):97–104, 2023.

[19] Kaddour Messaoudi, Omar Sami Oubbati, Abderrezak Rachedi, Abderrahmane Lakas, Tahar Bendouma, and Noureddine Chaib. A survey of uav-based data collection: Challenges, solutions and future perspectives. *Journal of network and computer applications*, 216:103670, 2023.

[20] Syed Agha Hassnain Mohsan, Nawaf Qasem Hamood Othman, Yanlong Li, Mohammed H Alsharif, and Muhammad Asghar Khan. Unmanned

aerial vehicles (uavs): Practical aspects, applications, open challenges, security issues, and future trends. *Intelligent service robotics*, 16(1):109–137, 2023.

[21] Panagiotis Radoglou-Grammatikis, Panagiotis Sarigiannidis, Thomas Lagkas, and Ioannis Moscholios. A compilation of uav applications for precision agriculture. *Computer Networks*, 172:107148, 2020.

[22] Muhammad Atta Ur Rahman, Melanie Schranz, and Samira Hayat. Llm-powered swarms: A new frontier or a conceptual stretch?, 2025.

[23] Fady Salama, Franz J. Ennemoser, Roman Binkert, Ege Korkan, Sebastian Käbisch, and Sebastian Steinhorst. Llm-mage: A generative mashup planner for the web of things. In Himanshu Verma, Alessandro Bozzon, Andrea Mauri, and Jie Yang, editors, *Web Engineering*, pages 342–357, Cham, 2026. Springer Nature Switzerland.

[24] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants. *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5977–6043, 2025.

[25] Luca Sciullo, Lorenzo Gigli, Federico Montori, Angelo Trotta, and Marco Di Felice. A survey on the web of things. *IEEE access*, 10:47570–47596, 2022.

[26] SP Sharan, Francesco Pittaluga, Manmohan Chandraker, et al. Llm-assist: Enhancing closed-loop planning with language-based reasoning. *arXiv preprint arXiv:2401.00125*, 2023.

[27] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large

language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530, 2023.

[28] Yonglin Tian, Fei Lin, Yiduo Li, Tengchao Zhang, Qiyao Zhang, Xuan Fu, Jun Huang, Xingyuan Dai, Yutong Wang, Chunwei Tian, Bai Li, Yisheng Lv, Levente Kovács, and Fei-Yue Wang. Uavs meet llms: Overviews and perspectives towards agentic low-altitude mobility. *Information Fusion*, 122:103158, 2025.

[29] Angelo Trotta, Marco Di Felice, Federico Montori, Kaushik R. Chowdhury, and Luciano Bononi. Joint coverage, connectivity, and charging strategies for distributed uav networks. *IEEE Transactions on Robotics*, 34(4):883–900, 2018.

[30] Sai H. Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *IEEE Access*, 12:55682–55696, 2024.

[31] Wenhao Wang, Yanyan Li, Long Jiao, and Jiawei Yuan. Gsce: a prompt framework with enhanced reasoning for reliable llm-driven drone control. In *2025 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 441–448, 2025.

[32] Wenhao Wang, Yanyan Li, Long Jiao, and Jiawei Yuan. Large language model-driven closed-loop uav operation with semantic observations. *IEEE Internet of Things Journal*, pages 1–1, 2025.

[33] Lillian Wassim, Kamal Mohamed, and Ali Hamdi. Llm-daas: Llm-driven drone-as-a-service operations from text user requests. In Faisal Saeed, Fathey Mohammed, Errais Mohammed, Shadi Basurra, and Mohammed Al-Sarem, editors, *Advances on Intelligent Computing and Data Science II*, pages 108–121, Cham, 2025. Springer Nature Switzerland.

[34] Yunhong Yang, Xingzhong Xiong, and Yuehao Yan. Uav formation trajectory planning algorithms: A review. *Drones*, 7(1):62, 2023.

[35] Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 951–972, 2025.