SwarmBox: A Plug-and-Play Drone Swarm Framework for Streamlined Development and Comprehensive Analysis

MINKI LEE, Pohang University of Science and Technology, Republic of Korea
SEOJIN LEE, Daegu Gyeongbuk Institute of Science and Technology, Republic of Korea
SEULBAE KIM*, Pohang University of Science and Technology, Republic of Korea

Drone swarms are emerging as paradigm-shifting technology with the potential to redefine traditional robot missions such as logistics, surveillance, and disaster response, through their ability to coordinate large numbers of autonomous drones. Yet, progress in swarm research and development is constrained by a fragmented development ecosystem; every new algorithm must be validated on a bespoke testbed, introducing significant and redundant engineering overhead, while producing results that are difficult to reproduce or compare. As a remedy, we present SwarmBox, an open-source testbed framework that provides a shared foundation for swarm robotics. SwarmBox streamlines swarm research and development by providing three key capabilities: (1) a plug-and-play software architecture that decouples high-level swarm logic from low-level flight control and platform dependencies; (2) a swarm-level integrated analyzer that exposes emergent behaviors across drones and system layers to facilitate debugging and analysis; and (3) a configurable experimentation environment that supports diverse missions and communication topologies to promote fair and reproducible benchmarking.

Our evaluation demonstrates these benefits in practice. SwarmBox reduces software engineering effort by eliminating boilerplate code, abstracting low-level system details into coherent APIs, and simplifying swarm coordination into a uniform process. It improves fault diagnosis efficiency and uniquely exposes inter-agent interaction failures that traditional debugging methods cannot capture. It enables reproducible benchmarking by allowing systematic comparison of different swarm algorithms. It proves its generality and scalability by supporting diverse mission scenarios ranging from centralized coordination to fully decentralized swarms. Finally, its hardware abstraction layer minimizes the simulation-to-real gap, enabling unmodified application code to be seamlessly deployed on both simulation and physical drones. Together, these capabilities establish SwarmBox as a practical foundation for reproducible, community-driven swarm robotics research.

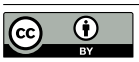
CCS Concepts: • **Computer systems organization** → *Embedded software*; **External interfaces for robotics**; **Robotic components**; *Distributed architectures*; **Real-time system architecture**; • **Software and its engineering** → **Development frameworks and environments**.

Additional Key Words and Phrases: Framework, Swarm, Drone, Cyber-Physical Systems, Distributed CPS

ACM Reference Format:
Minki Lee, Seojin Lee, and Seulbae Kim. 2026. SwarmBox: A Plug-and-Play Drone Swarm Framework for Streamlined Development and Comprehensive Analysis. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE093 (July 2026), 22 pages. https://doi.org/10.1145/3808100

*Corresponding Author

Authors' Contact Information: Minki Lee, Pohang University of Science and Technology, Pohang, Republic of Korea, leeminki@postech.ac.kr; Seojin Lee, Daegu Gyeongbuk Institute of Science and Technology, Daegu, Republic of Korea, niiijoes@dgist.ac.kr; Seulbae Kim, Pohang University of Science and Technology, Pohang, Republic of Korea, seulbae@postech.ac.kr.



ACM 2994-970X/2026/7-ARTFSE093
https://doi.org/10.1145/3808100

1 Introduction

Throughout the history of computer science, foundational open-source platforms have consistently acted as catalysts for innovation and rapid advancement. Linux, for example, revolutionized operating systems by creating a collaborative ecosystem. In the drone (*i.e.*, unmanned aerial vehicle) domain, frameworks such as ROS (Robotic Operating System) [24] and PX4 [17] have played a similar role by providing standardized, community-driven software stacks. These platforms reduced redundant engineering efforts, lowered barriers to entry, and enabled researchers to build upon a shared foundation.

However, the emerging field of *drone swarms*[1], which is expected to bring a paradigm shift to the drone domains including logistics and environmental monitoring, is currently bottlenecked by the absence of a foundational open-source framework. This critical gap is not accidental; it reflects the inherent complexities of drone swarm development, which needs to address at least three key engineering challenges: the asynchronicity of distributed systems, the timing constraints of real-time systems, and the physical interaction demands of Cyber-Physical Systems (CPS). While existing platforms such as PX4 and ROS have advanced the development of single-drone systems, *no unified, standardized framework exists for developing, testing, and analyzing swarms at scale.* As a result, existing research has remained fragmented, focusing separately on isolated aspects such as swarming algorithms, networking protocols, or resilience mechanisms [4, 13, 32], each built on ad-hoc, partial validation environments, rather than a cohesive, reproducible, and shared foundation. In other words, researchers have been forced to "reinvent the wheel," building bespoke, single-purpose testbeds and software just to validate their concepts. Consequently, developed swarm algorithms often are incompatible, non-reusable, and poorly maintained [12, 29, 32], hindering reproducibility.

In this paper, we address these problems by proposing SwarmBox, a highly configurable, open-source drone swarm framework that enables researchers to readily compose and experiment with diverse drone swarm algorithms in a plug-and-play manner. SwarmBox delivers three core capabilities: **(1) a plug-and-play architecture** that maximizes modularity, reusability, and extensibility through a reusable, hierarchical inheritance-based design, allowing users to integrate and test new components (*e.g.*, algorithms or protocols) with minimal effort; **(2) a swarm-level integrated logger and analyzer** that captures time-synchronized behaviors and interactions across the entire swarm, rather than per-drone logs, to accelerate development, debugging, and analysis; and **(3) a configurable experimentation environment** that unites these features, supporting diverse missions and communication topologies to promote fair and reproducible benchmarking. To realize these capabilities, SwarmBox applies established software-engineering principles: *the strategy and template method patterns* define abstract base classes that provide the skeleton of behavior control, enabling custom logic to be implemented as drop-in plugins; and a *distributed logging subsystem* with a unified data schema and time-synchronization mechanism powers a post-hoc analyzer that reconstructs a unified timeline and correlates events across distributed logs. We empirically validated SwarmBox, showing that it streamlines swarm development by reducing engineering overheads by more than 94%; seamlessly bridges sim-to-real gaps by enabling the execution of identical code across pure simulation, simulation-in-hardware (SIH), and physical flights; strengthens swarm-level analysis by exposing inter-agent delay and network loss; and enables academic benchmarking by revealing a superior algorithm via metric-driven comparison. We then implement eight real-world swarm algorithms on SwarmBox with various scales, showing its generality and scalability.

The research contributions of this work are as follows:

[1]In this work, we define a drone swarm as a decentralized multi-agent system in which agents coordinate toward a shared objective, exhibiting scalable (many concurrent agents) and emergent (unattainable without swarming) behavior.

- **Architectural Abstraction via Logic-Environment Isolation**: We propose SwarmBox, a framework architected on the principle of *Logic-Environment Isolation*. By decoupling high-level swarm algorithms from low-level physical dynamics and hardware interfaces, it allows identical algorithmic code to run seamlessly across diverse fidelity levels (Software-In-The-Loop, Simulation-In-Hardware, and physical platforms), eliminating the structural sim-to-real gap.
- **Swarm-Level Observability & Causal Reconstruction**: We introduce a *Synchronized Log Unification* mechanism that overcomes the partial observability limitations of distributed robotic systems. By reconstructing a unified causal timeline from fragmented agent logs, SwarmBox enables not only the diagnosis of emergent inter-agent interaction faults (*e.g.*, network latency-induced cascades) but also establishes a standardized benchmarking foundation, allowing researchers to rigorously compare the performance under identical conditions.
- **Empirical Validation of Efficiency & Generality**: Through a comprehensive evaluation spanning diverse computational paradigms and mission scenarios, we demonstrate that SwarmBox reduces engineering overhead by up to 99.5% (RQ1) while proving its *Generality and Scalability* to support various control architectures and large-scale operations up to 36 agents (RQ5).
- **Open Science**: Additionally, we release the full source code, datasets, and benchmark scenarios of SwarmBox at https://github.com/postech-compsec/swarmbox to facilitate community-driven innovation and reproducible research.

2 Background

2.1 Drone Swarms as Distributed CPS

Drone Swarm. A drone swarm is not merely a collection of robots; it is a complex, distributed CPS, in which computation is tightly coupled with physical dynamics under real-world uncertainty. Swarm's spectrum ranges from centrally planned multi-drone systems, driven by vehicles that follow precomputed trajectories issued solely by a central commander, *e.g.*, Ground Control Station (GCS), to advanced autonomous swarms with agent-level autonomy and flexible control hierarchies, where agents receive high-level goals (*e.g.*, survey an area) and determine their actions based on local observations and peer communication. This autonomy enables robust, adaptive, and scalable operations that do not necessarily have to be commanded by a central controller, but it comes at a cost; emergent, non-local effects are fundamentally challenging to predict, debug, and analyze. A practical swarm framework, therefore, must support the full spectrum, from simple coordination to decentralized swarming, while providing swarm-level observability for analysis.

Engineering Challenges in CPS. CPS is an integration of computation, communication, and physical processes. Its development is notoriously difficult due to challenges such as managing real-time constraints and ensuring physical safety. Even a single drone integrates a number of disparate components: a real-time flight controller, various sensors, a companion computer, and user algorithms, each with its own assumptions about timing, data semantics, and failure modes. When integrated, these assumptions can clash, causing system-level failures that are exceptionally difficult to diagnose. For instance, a high-frequency sensor update can overwhelm the processing power of a downstream node, causing failures, even though both modules work perfectly in isolation. At swarm scale, integration risks are amplified; assumptions must hold across multiple mobile agents communicating over an unreliable network, with asynchronous execution and partial observability.

2.2 The Drone Swarm Technology Stack

Modern drone swarms typically build upon the following common stack of open-source software, which is not designed for seamless swarm integration, creating significant engineering overhead.

ROS 2: The Communication Backbone. ROS 2 has become the de facto standard middleware for robotics, providing a robust communication backbone for enabling modular robot software development. Its rich ecosystem of tools for development and debugging further enhances its utility. However, ROS 2 was not designed with swarms in mind. It offers data transport and modularity, lacks native support for swarms, such as swarm state synchronization and integrated data logging. Researchers must re-implement these capabilities per project, creating recurring overhead.

Onboard Flight Controllers (FC). Each drone in a swarm relies on an FC for high-frequency sensing, state estimation, and low-level attitude control. PX4 is a widely-used open-source FC firmware [17], with strong ROS 2 interoperability. For complex tasks, a companion computer commands the FC over MAVLink or DDS (Data Distribution Service) [20] in a layered architecture. While powerful, this exposes developers to FC state machines and wire protocols, creating a steep learning curve and diverting focus from swarm logic to low-level implementation details.

Validation and Verification (V&V) Environments. Robotics V&V typically begins with Software-In-The-Loop (SITL) simulations, for safe and rapid testing of algorithms. It then progresses to Simulation-in-Hardware (SIH) or Hardware-In-The-Loop (HITL) testing, where actual FC hardware is used to validate timing and hardware integration. Physical flight is the final stage. Two persistent challenges are the sim-to-real gap and the stage transition overheads, which often require non-trivial code and configuration changes. A swarm framework should minimize this gap and streamline transitions enabling the same swarm algorithm to run across stages with minimal effort.

3 Motivating Example: Taming an Existing Swarm Algorithm

Despite growing interest in drone swarm research, adopting and validating existing swarm algorithms in new environments is frustratingly difficult. In this section, we present our own experience of customizing and evaluating the widely cited optimized flocking algorithm [32] into a modern swarm stack, and identify core, systematic challenges within.

Initially, the algorithm seemed promising and reusable; in [32], the authors demonstrated a successful 30-drone outdoor swarm and open-sourced their implementation. However, we discovered that the code was fragmented across two unmaintained repositories [7, 31] and only runs on a legacy simulation environment. Critical dependencies were broken, and we were forced to build a bespoke, two-computer testbed based on an outdated architecture, to replicate their setup.

Challenge 1: Fragmented Development Ecosystem. The lack of standardized interfaces and abstractions in the field of drone swarm forces researchers to constantly "reinvent the wheel."

Then, a more significant architectural challenge emerged as we planned to use the PX4 Autopilot stack, which is known for its permissive license and robust real-time performance [17]. The original algorithm utilized a customized version of ArduPilot. Porting it to PX4 meant re-architecting the software bridge entirely, designing a new interface between the high-level flocking logic and the PX4's state machine and communication protocols (*e.g.*, uORB). Our efforts had to be shifted from developing and improving high-level behavior to resolving low-level integration challenges.

Challenge 2: Architectural Complexity of Multi-Layered Drone Systems. Without clear abstractions across flight controllers, middleware, and application layers, swarm development becomes a full-stack engineering effort.

After integration, a drone repeatedly misbehaved during test flights. Yet, with the highly sophisticated artifact we have newly built, we could not easily determine if the failure was stemming from a flaw in our flocking algorithm implementation, a timing issue in the new software bridge we wrote, or a low-level control tuning error in PX4 flight controller. The system had become an opaque stack of interacting components, making fault isolation nearly impossible. To find the root cause,

we encountered the next challenge: an absence of swarm-level integrated analysis tools. We were left to manually sift through thousands of lines of disconnected logs from the C++ application and the PX4 flight controller (`*.ulg` files), with no established way to time-synchronize and correlate events across the different system layers and multiple drones.

Challenge 3: Absence of Swarm-Level Analysis Support. Our debugging effort was fundamentally limited by the lack of swarm-level analysis tools. Without a way to trace behaviors across multiple drones, multiple layers, and over a mission timeline, fault isolation is intractable.

Finally, after managing to resolve the issue, we faced a final, more subtle roadblock. The original algorithm's performance was evaluated using a completely different flight stack (ArduPilot) and hardware. Our new PX4-based system was inherently incomparable, making it impossible to validate whether our modifications indeed improved upon the original work.

Challenge 4: Barriers to Reproducibility and Fair Comparison. Without standardized software and hardware abstractions, even minor architectural differences make results incomparable. The lack of a benchmarking framework hinders scientific progress and reproducibility.

This experience highlights the lack of robustness prevalent in the current swarm research ecosystem. Due to the absence of a standardized framework, it is inevitable that research artifacts are tightly coupled to the specific software versions and hardware configurations available at the time of development [12]. Consequently, as underlying components rapidly evolve (*e.g.*, the transition from ROS 1 to ROS 2), these rigid implementations quickly become irreproducible. Furthermore, without a unified platform to ensure compatibility, repositories often face significant maintenance hurdles post-publication. To break this cycle of redundancy and fragmentation, we present SwarmBox, an open-source framework designed to provide a unified, extensible, and analyzable testbed for drone swarm research.

4 System Design

SwarmBox is a novel swarm framework designed to transform swarm research from a bespoke systems integration task into a repeatable, scientific workflow. Its core principle is separation of concerns: decoupling high-level swarm algorithms from low-level hardware control, middleware, and distributed communication. This enables plug-and-play reusability and swarm-level analysis that the community currently lacks. In this section, we first detail the architectural principles that realize this vision (§4.1), then present a concrete breakdown of the key system components (§4.2).

4.1 Design Goals and Principles

To address the challenges laid out in §3, we establish four design goals for SwarmBox:

DG1 Reusable and Effortless Integration: To break the "reinvent the wheel" cycle (Challenge 1) the framework should provide a well-defined abstraction, *e.g.*, an algorithm API specified in terms of swarm-state inputs and lifecycle hooks, that enables researchers to integrate, test, and reuse existing swarm algorithms with minimal porting effort.

DG2 Abstraction of Multi-layered System Complexity: The framework must hide the inherent complexity (Challenge 2) of the multi-layered drone architecture. Users should be able to focus on high-level swarm logic without a deep, expert-level knowledge of the underlying flight controller (*e.g.*, PX4), middleware (*e.g.*, ROS 2), and their intricate interactions.

DG3 Comprehensive Swarm-Level Analysis Support: To avoid debugging bottlenecks (Challenge 3), the framework should support swarm-level observability by providing built-in tools for collecting, synchronizing, and visualizing data from all agents and all system layers.

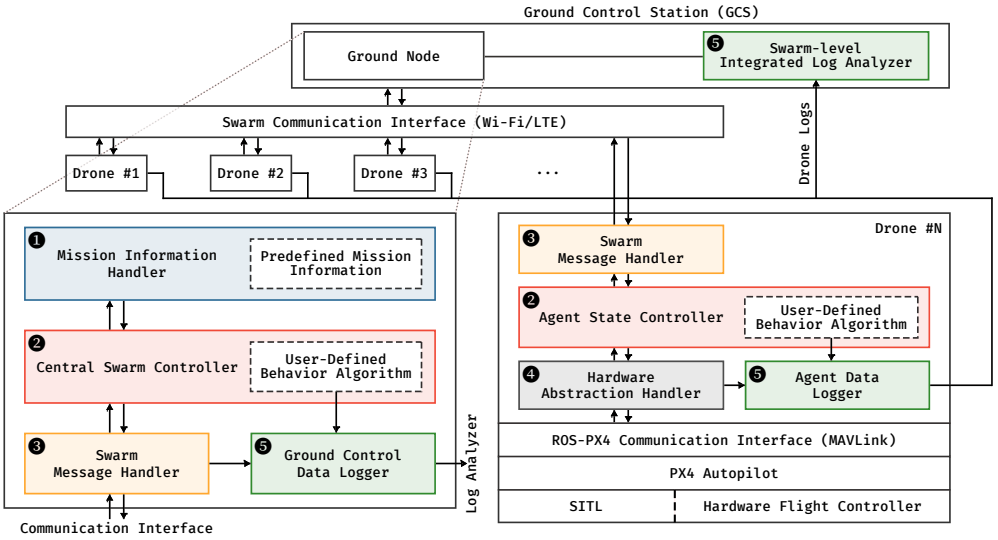


Fig. 1. Workflow of SwarmBox. Colored boxes denote system components and Dashed white boxes denote user inputs (*i.e.*, algorithm implementation and configuration).

DG4 Reproducible and Standardized Benchmarking: To serve as a shared foundation, the framework must provide a standardized, controllable experimental environment that enables researchers to orchestrate and evaluate algorithms under identical conditions. Countering Challenge 4, this ensures fair comparison, supports reproducible results, and enables cumulative progress through directly comparable studies.

To achieve the above design goals, we establish three fundamental architectural principles that govern the design of all system components:

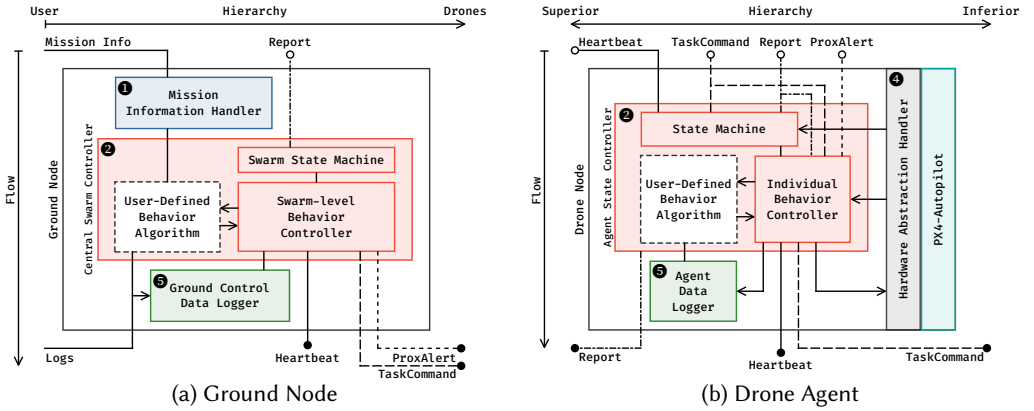
AP1 Extensibility via Inheritance-Based Architecture: To address **DG1** and **DG2**, SwarmBox adopts an inheritance-based architecture, providing core functionalities as abstract base classes. Users can implement custom logic (*e.g.*, flocking) by inheriting from the base classes and overriding methods. This enforces a clean separation between the framework's stable infrastructure and the user's experimental code, maximizing modularity and reusability.

AP2 Observability via Post-Hoc Log Unification: To address **DG3**, SwarmBox provides enhanced observability through post-hoc log unification. Considering real-time network burden of swarm operations, it decouples lightweight live monitoring from in-depth, post-mission analysis. During missions, the ground control system aggregates minimal swarm-level health and state telemetry for real-time situational awareness. After execution, SwarmBox unifies and time-synchronizes detailed per-agent logs to produce a time-aligned swarm-level trace that enables cross-agent correlation and causal reconstruction of behaviors.

AP3 Configurability of Swarm Topologies: To support **DG1** and **DG4**, SwarmBox is designed to be topology-agnostic, allowing users to dynamically define the swarm's topology and agents' roles through configuration files, so that it can realize diverse swarm architectures, *e.g.*, centralized, hierarchical leader-follower, or fully decentralized peer-to-peer swarms, facilitating rapid experimentation and standardized benchmarking across architectures.

4.2 System Components

As shown in Figure 1, SwarmBox consists of two units: a central `Ground Node` in GCS and distributed `Drone Agent` units. Five components span these units: the Mission Information Handler (❶, §4.2.1),



(a) Ground Node (b) Drone Agent

Fig. 2. Message handling process of SwarmBox, orchestrated by the Swarm Message Handler. Subscriptions are denoted by ○ and publications by ●. Identical line styles (*e.g.*, dashed) indicate the same message type.

which bootstraps missions from user-defined configurations; the Hierarchical Swarm Controller (❷, §4.2.2), which executes the core swarm algorithm logic; the Swarm Message Handler (❸, §4.2.3), which orchestrates ground-to-drone and drone-to-drone messages; the Hardware Abstraction Layer (❹, §4.2.4), which decouples control logic from underlying flight controllers; and the Swarm-level Logger and Analyzer (❺, §4.2.5), which facilitates offline, time-aligned analysis and visualization.

4.2.1 Mission Information Handler (❶). Located on the `Ground Node`, this component serves as the key enabler of SwarmBox's flexibility (**DG1**, **DG4**). As an entry point, it parses the user's swarm mission configuration that defines mission parameters, fleet size, a common global reference frame for consistent frame transforms across all agents, and command & communication topology.

Rather than a fixed hierarchy, SwarmBox allows users to define arbitrary, multi-level control structures by specifying each drone's `superior`. This enables diverse topologies, such as centralized control (no superiors; all drones are commanded by the GCS), leader-follower (followers assign the leader as superior, which takes commands from the GCS), and chained/tree structures (select drones act as intermediate superiors, forming multi-level command chains). The Mission Information Handler instantiates the topology by programmatically configuring the communication routes and state machine dependencies for the Hierarchical Swarm Controller before the mission begins.

4.2.2 Hierarchical Swarm Controller (❷). This component is responsible for orchestrating the swarm mission's lifecycle end-to-end. Its logic is implemented as a hierarchical state machine, which ensures that all distributed agents operate in a synchronized and predictable manner. The state machines are physically distributed: a global swarm state machine runs on the `Ground Node` (Figure 2a), while per-agent state machines run on each `Drone Agent` (Figure 2b).

Central Swarm Controller. Residing on the `Ground Node`, this module manages the global swarm state machine, which progresses through six mission stages: `icebreak` (initial connection establishment), `preparation` (pre-flight checks and mission preparations), `execution` (mission logic), `landing`, and `termination`, with a global `abort` state accessible from any stage. It acts as the single source of truth for the swarm's overall progress. The decision to transition to the next stage is made based on the collective information it gathers from all individual drones via their `Report` messages, ensuring the entire swarm is ready before proceeding.

Agent State Controller. Running on each `Drone Agent`, this module executes a per-agent state machine that mirrors the global state machine on the `Ground Node`. An agent transitions to the next

stage only upon receiving a `Heartbeat` message, indicating that the global swarm state has changed. This synchronization primitive ensures all agents consistently remain in phase. User-defined algorithms run within these state contexts. In addition, for safety, each agent can autonomously transition into an `abort` state if its flight controller or a local safety monitor detects a hazard, ensuring rapid, localized emergency response.

4.2.3 Swarm Message Handler (❸). Resident on both the `Ground Node` and `Drone Agent`, this component orchestrates state synchronization and mission traffic. `Heartbeat` messages carry state commands (*e.g.*, current stage and successor information), and `Report` messages provide the necessary feedback (*e.g.*, current position) for the `Ground Node` to make state transition decisions. Additional message types (`TaskCommand`, `ProxAlert`) support mission-specific logic (*e.g.*, go to a setpoint) in the `execution` state. For resilience, the handler enables dynamic hierarchy reconfiguration. Each superior node embeds an ordered list of successors in its `Heartbeats`, which all agents cache. If an agent loses connection to its primary successor, it autonomously initiates a handover protocol, attempting to re-establish a link with the next successor in the list. Also, if the GCS is the sole superior of the agent and the connection cannot be re-established, agents autonomously trigger the flight controller's default failsafe mode (e.g., Return-to-Launch). This decentralized recovery mechanism yields rapid and robust adaptation to network failures without GCS intervention.

4.2.4 Hardware Abstraction Handler (HAH, ❹). Residing exclusively on the `Drone Agent`, the HAH is the crucial layer to achieve **DG2**. It provides a clean, high-level interface between the flight controller and the **Agent State Controller**, translating abstract state/mission commands into controller-specific ROS 2 messages and hiding the low-level protocol details from user algorithms. To ensure extensibility and avoid lock-ins with any single flight stack, the HAH employs a modular *adapter* design with two parts: a Generic Interface and a swappable Bridge Module.

Generic Interface. This interface defines a stable, bidirectional contract for interacting with a drone. It exposes a standard set of high-level commands for vehicle control and provides a standardized telemetry model for state and sensor data, including interpretation and conversion to a common global frame. This interface specifies *what* is exchanged, fully decoupling user algorithms from hardware specifics.

Bridge Module. This module is a concrete, bidirectional translator for a particular flight controller. Its responsibilities include:

- *Command Translation*: Mapping abstract commands from the Generic Interface to verbose controller messages with timestamps, type masks, and protocol-specific fields.
- *Safety Monitoring*: Checking safety constraint violations (*e.g.*, corrupted setpoint) to prevent user algorithms from issuing hazardous commands to the flight controller.
- *State & Sensor Data Normalization*: Converting raw sensor and state data from the flight controller back into standardized data model defined by the Generic Interface. Crucially, it handles all frame transformations, ensuring user algorithms can operate consistently in a global reference frame.

In this work, we implement a PX4 Bridge, for its robust performance and ROS 2 compatibility. However, the HAH's modular architecture can seamlessly support other flight controllers. A user can, for example, implement an ArduPilot Bridge, which only requires adherence to the **Generic Interface**. Thus, SwarmBox's core logic and user-level swarm algorithms remain portable across hardware stacks, with integration tasks isolated to a self-contained bridge module.

4.2.5 Swarm-level Integrated Logger & Analyzer (❺). Hosted on the GCS, this component addresses **DG3** by providing a swarm-level observability that is absent in traditional approaches. During mission execution, Ground Control Data Logger and Agent Data Logger locally write their logs onboard. After a swarm flight, the Swarm-level Integrated Log Analyzer processes a unified and

time-synchronized log trace from distributed agents for post-mission analysis and visualization. Specifically, it collects and integrates three critical streams of information:

- Built-in Swarm Metrics: SwarmBox automatically logs key information including GCS's control messages and health indicators, such as inter-agent network latency, message loss, and physical position discrepancy. These metrics provide a capability for diagnosing system-level faults.
- User-Defined Event Markers: A lightweight API allows users to inject custom, semantic markers to annotate high-level events and enable mission-specific performance analysis.
- Low-level Onboard Logs: The analyzer also facilitates the collection and contextualization of traditional, low-level log files (*e.g.*, PX4 `ulg` files) created from each drone.

By synchronizing these multi-layered data streams on a common timeline, the analyzer supports cross-agent correlation, communication graphs, and causal reconstruction, enabling efficient debugging of emergent behaviors and rigorous performance evaluation.

5 Evaluation

This section empirically validates the effectiveness of SwarmBox. Our evaluation is structured around five Research Questions (RQs); RQ1-RQ4 are designed to systematically assess their corresponding design goals established in §4, and RQ5 validates the generality and scalability of the framework. By answering these questions, we demonstrate the practical benefits and contributions of our framework. The research questions are as follows:

RQ1. Engineering Effort Reduction: How effectively does SwarmBox reduce the engineering effort required to implement and integrate diverse swarm algorithms? (§5.1).

RQ2. Sim-to-Real Fidelity: Does SwarmBox maintain behavioral consistency and high fidelity across simulation and real-world environments? (§5.2).

RQ3. Fault Diagnosis: To what extent does SwarmBox improve the efficiency of diagnosing emergent, system-level faults compared to traditional, single-drone analysis? (§5.3).

RQ4. Benchmarking Capability: Can SwarmBox function as a standardized benchmark to enable fair, reproducible comparisons of different swarm algorithm implementations? (§5.4).

RQ5. Generality & Scalability: Can SwarmBox seamlessly support diverse swarm algorithms and large-scale operations? (§5.5).

Experiment Setup. We evaluate SwarmBox across three distinct configurations: (1) A purely *in-silico* Software-in-the-Loop (SITL) simulation, (2) a hybrid Simulation-in-Hardware (SIH), (3) and *in-situ* real-world flights. The *in-silico* experiments are performed in the Gazebo simulator [14] running on a desktop computer running AMD 9800X3D processor with 64 GB of RAM under Ubuntu 22.04.5 LTS. For both the SIH and *in-situ* flights, we use Holybro X500 V2 quadrotor drones, each equipped with a Pixhawk 6C flight controller running PX4 Autopilot firmware with a minor modification for multi-vehicle execution and a Raspberry Pi 3 Model B as a companion computer.

5.1 RQ1: Engineering Effort Reduction

To answer RQ1, we study how SwarmBox reduces engineering effort by porting three popular open-sourced swarm algorithms from their original environments to SwarmBox. These algorithms were selected to cover diverse computational paradigms and legacy software architectures:

- Adaptive Swarm [1, 26]: A path planning-based algorithm with collision avoidance implemented in Python on ROS 1, representing a hardware-validated legacy implementation.
- Optimized Flocking [7, 32]: An evolutionary flocking algorithm implemented in a custom C-based simulator, representing monolithic, high-performance legacy code.
- Socratic Swarm [6, 11]: An optimization-based consensus algorithm implemented in Unity using C#, representing research code tightly coupled to a physics-engine.

Table 1. Comparison of Lines of Code (LoC) between legacy code and SwarmBox-ported implementation.

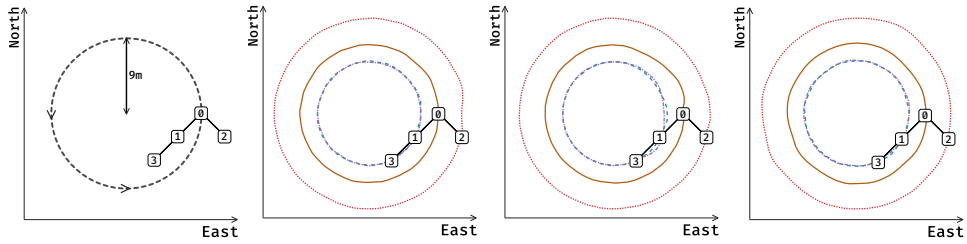
	Boilerplate LoC (①)			Core Logic LoC (②)			Total LoC (①+②)		
Target Algorithm	Legacy	Ported	**Reduction**	Legacy	Ported	**Reduction**	Legacy	Ported	**Reduction**
Adaptive Swarm [1]	729	44	↓ **94.0%**	673	540	↓ **19.8%**	1402	584	↓ **58.3%**
Optimized Flocking [32]	6110	44	↓ **99.3%**	1582	291	↓ **81.6%**	7692	335	↓ **95.6%**
Socratic Swarm [11]	8134	44	↓ **99.5%**	1244	284	↓ **77.2%**	9378	328	↓ **96.5%**

Experimental Design. We evaluate the effort reduction by analyzing the source code of the original implementations and their SwarmBox-ported versions. To quantify the effort reduction, we decompose the total implementation work into the *Architectural Boilerplate* (①), which includes the labor-intensive, low-level code for system integration (*e.g.*, hardware interfacing, communication management), and the *Core Logic* (②), which is a platform-agnostic behavior of the implemented algorithm. Then, we evaluate the change of engineering effort for each category before and after porting using Source Lines of Code (LoC) as a quantitative metric. By comparing these metrics, we quantitatively assess the reduction in engineering effort and analyze the shift in implementation focus from infrastructure management to algorithmic logic.

RQ1 Results. As summarized in Table 1, the results demonstrate a dramatic reduction in total LoC across all algorithms. Especially, the boilerplate implementation required was reduced to over 94.0% in all cases. Beyond the quantitative reduction, our qualitative analysis during the porting process revealed three key findings regarding how SwarmBox improves engineering efficiency:

(1) **Decoupling of Simulation Infrastructure**. In the legacy implementations, particularly Optimized Flocking and Socratic Swarm, the algorithmic logic was tightly coupled with the simulation infrastructure. For instance, the Optimized Flocking's codebase dedicated over 6K LoC to manual physics integration (*e.g.*, Euler methods) and rendering pipelines. SwarmBox eliminated this burden by delegating physics and visualization to its backend, allowing the ported implementation to contain only the pure Reynolds' interaction rules of flocking [25].
(2) **Abstraction of Hardware and Communication**. The AdaptiveSwarm required extensive boilerplate code to manage asynchronous communication topics and hardware drivers. By porting this to SwarmBox, we removed the need for manual message serialization and hardware bridging. The framework's middleware layer automatically handles synchronization, enabling the code to be agnostic to the underlying communication protocol.
(3) **Mathematical Abstraction of Complex Behaviors**. For Socratic Swarm, the original reliance on the Unity engine and custom optimization libraries was replaced by a concise mathematical implementation using standard libraries. This validated that SwarmBox allows researchers to abstract complex behaviors (*e.g.*, cost function minimization and bidding) into simple logic blocks without carrying the technical debt of physical engines or meta-learning overheads.

RQ1 Conclusion. Our comparative study demonstrates that SwarmBox significantly reduces engineering overhead, manifested as a drastic decrease in implementation Lines of Code (LoC). By enforcing Logic-Environment Isolation, SwarmBox eliminates infrastructure dependencies ranging from physics engines to network stacks. This structural decoupling allows researchers to achieve high algorithmic density, enabling them to focus exclusively on the scientific validation of their swarm behaviors.



(a) Mission Plan (b) SITL (c) SIH (d) Physical

Fig. 3. Swarm trajectories across different environments. Numbered markers indicate the initial positions.

5.2 RQ2: Sim-to-Real Fidelity

To answer RQ2, we evaluate the effectiveness of SwarmBox's HAH in bridging sim-to-real gap. Specifically, we compare the behavioral consistency of a SwarmBox-based swarm algorithm executed in a simulation with a real-world flight, using the *exact same* swarm code.

Experimental Design. We designed a formation flight mission with four drones.

- Mission Scenario: The swarm is tasked to maintain a formation with a fixed inter-drone relative distance of (3, 3) meters, as shown in Figure 3a. The formation leader (Drone #0) is commanded to follow a circular trajectory with a radius of 9 m at a constant altitude of 3 m, which is an ideal configuration for measuring both the path-following and relative formation-keeping capabilities. To minimize the impact of networking noise in this experiment, the GCS sends master commands to the leader drone and relays the leader's position to the follower drones.
- Environments: We conducted the experiment in three distinct environments:
  (1) SITL (Software-In-The-Loop): The entire system, including the PX4 firmware and the drone dynamics, is simulated in the Gazebo simulator.
  (2) SIH (Simulation-In-Hardware): The simulation of the drone's dynamics and sensors is executed directly on the actual flight controller, *i.e.*, Pixhawk 6C, making it independent of any external simulator.
  (3) Physical: Actual drones are deployed outdoors to conduct swarm missions, relying on onboard sensors (*e.g.*, GPS) for localization.

RQ2 Results. The same application code for the formation flight mission was deployed to all three environments *without any modification*. To quantify the fidelity of our system, we analyzed the trajectory deviation between the SITL simulation and the physical outdoor flight. Across the swarm, the Root Mean Square Error (RMSE) between the simulation and physical flights ranged from as low as 0.09 m to 0.25 m. This deviation is negligible and falls well within the 1.5 m Circular Error Probable (CEP) of the GNSS module [30], confirming that the flight logic executed onboard is functionally identical to the simulation. This high fidelity is visually corroborated in Figure 3. As shown, the executed trajectories of individual drones in both SIH and physical flights closely align with those observed in the pure simulation (SITL). This demonstrates that our hardware abstraction layer effectively eliminates logic degradation, bridging the sim-to-real gap with high precision.

RQ2 Conclusion. We demonstrated that SwarmBox's hardware abstraction layer successfully minimizes the sim-to-real gap and streamlines the V&V process. Researchers can develop and validate their swarm algorithms starting from pure software simulation, progressing through hardware-in-the-loop testing, and ultimately physical deployment, all with high confidence and minimal code changes. This capability lowers the barrier to real-world swarms, reducing the risks and costs of physical testing while accelerating the overall research and development cycle.

5.3 RQ3: Fault Diagnosis

To answer RQ3, we validate the swarm-level diagnostic capability of SwarmBox by designing fault scenarios that represent two fundamental classes of problems in distributed systems: (1) the internal state uncertainty of individual agents and (2) the unreliability of inter-agent interactions.

Experimental Design. To clearly observe the impact of faults, we scaled up the formation flight algorithm of §5.2 to nine drones, and conducted controlled fault injection experiments. Specifically, we implemented the following three non-invasive fault injection methods:

- *Sensor Fault (GPS Drift)*: We inject a linearly increasing GPS drift error into PX4 SITL.
- *Network Latency*: We inject controlled delays to simulate network latency.
- *Packet Loss*: We simulate message loss by deliberately dropping packets in SITL bridge.

To eliminate potential bias, each injection experiment was repeated nine times with the fault being injected into a different member drone each time. Presented results are the average of these nine runs.

Method and Metrics. We analyzed faults relying solely on SwarmBox's Post-Event Log Analyzer (§4.2.5), which produces a time-synchronized report containing the states and events of all drones. Specifically, there are built-in key diagnostic metrics in this log, including position discrepancy, network latency, and message loss, which we utilize to pinpoint the root cause of the faults. To quantify diagnostic efficiency, we defined three following metrics:

- *Time to First Anomaly* (t_f): The time elapsed from fault injection until a key diagnostic metric (*e.g.*, position discrepancy) first deviates from its statistical norm.
- *Anomaly Detection Threshold* (τ_d): The observed value of the key diagnostic metric at t_f.
- *Anomaly Signal Clarity* (Σ_a): The relative magnitude of the anomaly signal compared to the system's base operational noise, calculated as the IQR of the metric's values across the swarm.

Diagnosis of Sensor Fault (GPS Drift). First, we compared SwarmBox's enhanced ability to diagnose an arbitrary drone's GPS sensor fault and analyze its swarm-wide impact, against a baseline approach. For baseline anomaly detector, we use the flight controller's internal EKF[2] state, *i.e.*, `estimator_status/reset_count_pos_ne`'s value, based on our analysis that its value is incremented only when the EKF detects a critical GPS-induced inconsistency in its position estimation. Note that we initially attempted to utilize the built-in failsafe mechanism of PX4 as a baseline. However, we observed that GPS error injection attack never triggered the failsafe.

As presented in Table 2, for all drift intensities, SwarmBox's position discrepancy metric detected the anomaly faster (smaller t_f), at smaller physical errors (smaller τ_d), and with better sensitivity (smaller Σ_a) than the baseline. Notably, we observed instances where the PX4 flight controller failed to react at all to injected GPS drifts below 2.0 m/s. While such failures may be tolerable for a single drone, they pose serious risks in swarm operations where maintaining inter-agent positional integrity is critical. This finding highlights that SwarmBox provides the necessary sensitivity to detect subtle, gradual GPS drifts that are missed at a single-drone level, demonstrating its value in ensuring the safety and reliability of tightly-coupled swarm missions.

Second, beyond automated detection metrics, we also qualitatively compared the manual effort required to localize the fault's root cause, as summarized in Table 3. It shows that SwarmBox significantly reduces the user's manual effort for fault localization in post-event cause analysis.

Diagnosis of Network Latency and Packet Loss. Next, we analyzed inter-drone communication faults. It should be noted that the individual drone's original logs do not contain any information about network latency or message loss between agents. This reveals a critical blind spot in traditional single-agent analysis: its inability to address network-level faults. In contrast, SwarmBox monitors

[2]Extended Kalman Filter (EKF) estimates a drone's position and attitude from noisy on-board sensor data.

Table 2. GPS drift detection result (τ_d metric: position discrepancy).

	t_f (s)		τ_d (m)		Σ_a	
Drift (m/s)	Baseline	**SwarmBox**	Baseline	**SwarmBox**	Baseline	**SwarmBox**
1.0	Not Detected	**3.19**	N/A	**1.81**	N/A	**2.05**
2.0	29.73	**2.66**	30.05	**2.10**	75.42	**1.12**
3.0	17.59	**2.23**	39.15	**2.26**	83.40	**0.54**
4.0	16.55	**2.19**	57.08	**2.23**	132.09	**1.64**
5.0	16.16	**1.99**	62.96	**2.58**	163.27	**1.43**

Table 3. Comparison of diagnostic workflow for GPS sensor fault.

Steps Required By the Baseline	**Steps Required By SwarmBox**
(1) Download individual `.ulg` files from each drone. (2) Convert logs into an analyzable format (*e.g.*, CSV). (3) Manually time-synchronize logs using a common event (*e.g.*, arming). (4) Look for warnings or specific anomaly log messages published by PX4. (5) Read `estimator_status/reset_count_pos_ne` for each drone separately. (6) Visually compare multiple plots to identify the drone with the deviation.	(1) Launch the Post-Event Log Analyzer. (2) Check the unified plot created by the analyzer for the discrepancy metric and instantly identify the faulty drone.

Table 4. Detection result of injected network delays. τ_d represents the detected delay.

Delay	5 ms	10 ms	15 ms	20 ms	25 ms
t_f (ms)	24.94	24.94	24.93	24.93	24.95
τ_d (ms)	5.70	10.54	15.84	20.71	25.47
Σ_a	164.84	286.27	494.69	737.44	670.49

Table 5. Detection result of packet loss. τ_d represents the percentage of lost messages.

Loss	10%	20%	30%	40%	50%
t_f (ms)	24.92	24.95	24.93	24.94	24.95
τ_d (%)	9.22	21.33	31.67	41.11	49.22
Σ_a	∞	∞	∞	∞	∞

all swarm communications at the `Ground Node`, enabling detection and quantitative analysis of communication faults that are impossible to diagnose with the baseline method.

Table 4 and Table 5 show that SwarmBox sensitively detects and accurately quantifies injected network faults. It demonstrates immediate detection, capturing both delay and loss faults on the first possible system tick (~25 ms) as dictated by its 40 Hz monitoring frequency. Furthermore, the measurements show high fidelity, with the detected anomaly thresholds (τ_d) for both delay and loss closely mirroring the ground-truth injected values. This accuracy extends to discrete events, where the infinite Anomaly Signal Clarity (Σ_a) for message loss highlights the analyzer's ability to capture unambiguous failures with perfect clarity—a critical feature for distributed systems. This result proves that SwarmBox transcends the limitations of the traditional approach, providing a fundamentally new and highly performant capability to analyze a critical swarm issues.

RQ3 Conclusion. SwarmBox dramatically improves the efficiency of diagnosing complex faults in swarm systems. Compared to traditional methods, it (1) quantitatively enhances the diagnostic efficiency for internal agent faults, and (2) provides an enhanced capability to analyze inter-agent interaction faults, which were previously undiagnosable.

5.4 RQ4: Benchmarking Capability

To answer RQ4, we demonstrate how SwarmBox supports benchmarking of different swarm algorithms, which is a crucial capability for a practical swarm framework. Specifically, we strategically selected a configuration-sensitive swarm logistics scenario, in which a user must determine an

Table 6. Package-assigning strategy benchmark metrics. General-purpose metrics are provided by SwarmBox by default, and mission-specific metrics are collected via the custom event marking feature of SwarmBox.

Category	Metric	Explanation	Unit
Cost Efficiency (General-Purpose)	Total flight distance	Total flight distance per drone during execution	*m*
	Total mission duration	Total mission time: the duration of mission execution	*s*
Resource Utilization (General-Purpose)	Idle time	Average duration that drone waits after completion	*s*
	Utilization	Average drone utilization rate during the execution stage	%
	Workload deviation: distance	The standard deviation of flight distance	*m*
	Workload deviation: time	The standard deviation of utilization time	*s*
Delivery Quality (Mission-Specific)	Average delivery wait	The average of the delivery time of each package	*s*
	Maximum delivery wait	The delivery time of the last delivered package	*s*

optimal package assignment algorithm. We implemented nine different *delivery task allocation algorithms* and compared them using the analysis results provided by SwarmBox's log analyzer (§4.2.5) to demonstrate that SwarmBox expedites systematic, metric-driven comparisons of diverse algorithms with minimal engineering effort. The key challenges in this scenario are as follows:

- *Task Allocation*: Tasks can be assigned to drones both statically before the mission and dynamically during mission execution.
- *Resource Management*: Drones operate under practical limitations such as battery life and payload capacity. These constraints must be considered to ensure feasible and efficient mission planning.
- *Multi-Objective Optimization*: The mission must balance multiple objectives, including traditional swarm metrics (*e.g.*, mission duration, workload balance), as well as mission-specific goals (*e.g.*, average delivery wait time).

Experimental Design. To evaluate the performance of various delivery task allocation algorithms in a consistent and reproducible setting, we define the following experimental setup:

(1) Five drones are assigned to deliver 100 packages within a 200 m × 200 m area.
(2) Each package must be picked up from a central depot, *i.e.*, each drone's `HOME` position, and delivered to a designated drop-off point sequentially.
(3) The locations of the 100 delivery points are randomly generated within the specified target area by an automated script. The delivery sequence is not pre-sorted.
(4) To isolate the impact of task assignment strategies, package pickup and drop-off are omitted, *i.e.*, a task is considered complete upon the drone's arrival at the respective location.
(5) To account for variance due to random point placement, we generate 20 unique datasets using different random seeds.

Target: Delivery Task Allocation Algorithm. We systematically created nine delivery task allocation algorithms by combining three *package sorting rules* and three *task assignment methods*:

- *Package Sorting Rules*
  - No Sort (Baseline): Packages are processed in their default, randomly generated order.
  - Azimuthal Sort: Packages are sorted by their azimuth angle relative to the center origin: $(0, 0)$.
  - Distance Sort (Ascending): Packages are ordered from the nearest to the farthest delivery point relative to the origin.
- *Task Assignment Methods*
  - Block Pre-Assignment: Packages are assigned in contiguous blocks (*e.g.*, Drone 1 gets packages 1-20, Drone 2 gets packages 21-40, etc), prior to mission start.
  - Round-Robin Pre-Assignment: Packages are assigned one-by-one to each drone in a rotating manner, prior to mission start.
  - Dynamic Assignment: Packages are assigned on demand during the mission. A drone receives a new delivery task only after completing its current one.

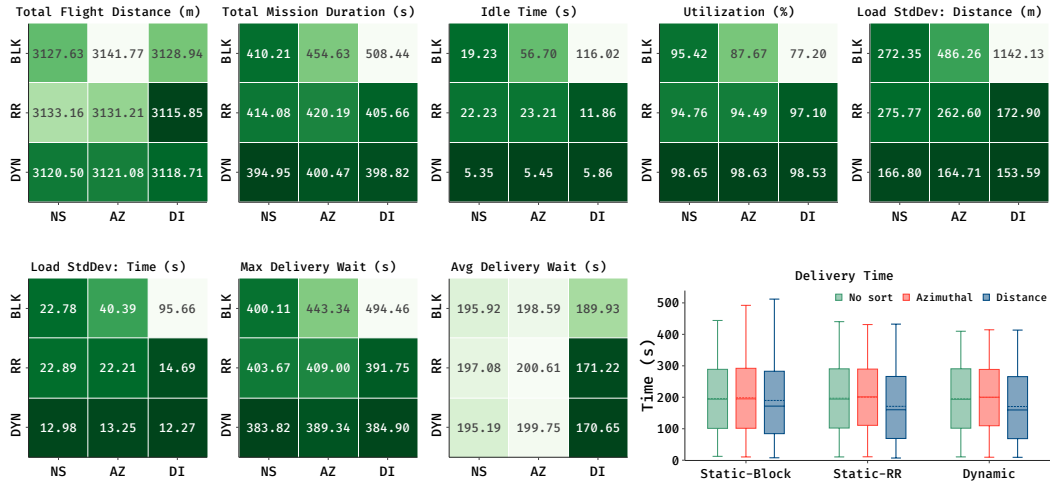


Fig. 4. Benchmark results for different package-assigning algorithms. Darker cells indicate better performance. The Delivery Time box plot shows the distribution across different datasets. NS: No sorting, AZ: Azimuthal sort, DI: Distance sort, BLK: Block pre-assignment, RR: Round-robin pre-assignment, DYN: Dynamic assignment.

Performance Metrics and Framework Support. As introduced in §4.2.5, SwarmBox provides general-purpose metrics that are common to most swarm missions, as well as mission-specific metrics that can be enabled through lightweight user-defined event markers. For the specific use case of comparing delivery algorithms, we defined a set of performance metrics that assess cost efficiency, resource utilization, and delivery quality, as summarized in Table 6.

Assessment of Framework Overhead. We validated that SwarmBox introduces negligible overhead, ensuring it does not skew the benchmark results. We measured the system resource consumption per agent during the mission. The framework utilized only 1.52% of CPU resources and approximately 43 MB of RSS memory on a standard companion computer. Furthermore, the network traffic overhead was limited to 35 KB/s per agent. These results confirm that SwarmBox operates efficiently as a lightweight middleware, leaving the vast majority of computational and network resources available for the user's swarm algorithms.

RQ4 Results. Figure 4 summarizes the performance of each strategy across the eight defined benchmark metrics in Table 6, with which we, and any users of SwarmBox, can compare performance and select the optimal algorithm. In the following, we highlight the independent and combined effects of package sorting rules and task assignment methods:

- *Impact of Package Sorting Methods*: The choice of package sorting method had a significant impact on the overall efficiency. Sorting by distance consistently outperformed the other methods, reducing both total flight distance and total mission duration across all assignment methods. In contrast, the azimuthal sorting offered no discernible advantage over the unsorted baseline; it occasionally increased the mission time, indicating it is not an optimal method for delivery.
- *Impact of Task Assignment Methods*: Dynamic assignment was proven superior in most efficiency-related metrics, including mission duration and drone utilization rate. Although block pre-assignment combined with distance sort achieved a low average delivery time, it showed the longest worst-case delivery time (the long upper tail in 'Delivery Time' box plot in Figure 4). The round-robin pre-assignment method proved more effective at mitigating this worst-case risk, showing a more consistent delivery time distribution compared to block pre-assignment.

Table 7. Selected swarm mission scenarios and their characteristics.

Case #	Scenario	Mission	Interdependence	Dynamism	Control Architecture
1	`airview`	Collaborative spatial information acquisition	Moderate	Moderate	Hybrid
2	`delivery`	Individual task-based distributed logistics	Low	High	Hybrid
3	`formation`	Execution of preplanned cooperative flight	High	Low	Hybrid
4	`network`	Demand-responsive mobile hotspot network	High	High	Decentralized
5	`tracking`	Dynamic target tracking and management	Moderate	High	Centralized
6, 7, 8	`ported`	Validation of the ported algorithms of RQ1	High	High	Centralized

- *Optimal Algorithm*: The optimal algorithm turned out to be the combination of dynamic assignment with distance sorting. It not only outperformed other algorithms in average performance metrics but also demonstrated exceptional consistency; as shown in the 'Delivery Time' box plot, this algorithm had the lowest variance.

RQ4 Conclusion. This experiment highlights how SwarmBox supports rigorous benchmarking of swarm algorithms. Beyond identifying simple trade-offs, with SwarmBox, developers can systematically validate and select an optimal algorithm through multi-faceted analysis using its general-purpose and user-defined, mission-specific metrics.

5.5 RQ5: Generality & Scalability

To answer RQ5 and demonstrate the generality and scalability of our framework, we implement a diverse set of real-world swarm missions on SwarmBox, scaling to fleet sizes up to 36 agents. These missions illustrate how SwarmBox can support varying degrees of communication, cooperation, and autonomy depending on the operational context and objectives. We systematically characterize these missions along three key dimensions:

- *Interdependence*: This axis defines the degree to which an individual agent's decisions and actions are influenced by the state of other agents in the swarm. This can range from *Low* (*e.g.*, agents executing pre-planned, independent paths) to *High* (*e.g.*, agents maintaining a tight formation based on real-time data from neighbors).
- *Dynamism*: This axis represents the degree to which the swarm's control logic must adapt to real-time events and changing environmental conditions. This ranges from *Static* (pre-computed plans), to *Dynamic* (continuous adjustments).
- *Control Architecture*: This axis describes the locus of decision-making authority within the swarm. In a *Centralized* architecture, the GCS makes the key strategic decisions and issues direct commands that trigger agent behaviors. In a *Decentralized* architecture, the GCS provides only high-level goals or environmental context, and each drone computes its own final setpoint through complex, local interactions. A *Hybrid* architecture involves the GCS assigning concrete intermediate goals, which the drones then execute with a degree of autonomy.

To demonstrate that SwarmBox is widely applicable, we selected eight representative scenarios from the vast design space created by these three axes, each with a distinct profile. We show that SwarmBox can effectively implement the required behaviors for each scenario, highlighting its flexibility and robustness. Each scenario and associated classification is summarized in Table 7.

Case 1. Airview: Collaborative Spatial Information Acquisition. To demonstrate rapid, collaborative aerial mapping for applications like search and rescue [3, 10], we implemented the `airview` scenario. The mission is realized using a hybrid architecture where the GCS performs an initial, centralized task allocation by assigning a unique sector to each drone. Subsequently, each drone autonomously executes an efficient coverage path within its assigned area. As shown in Figure 5a, this use case highlights how SwarmBox facilitates complex architectures that blend centralized planning with decentralized, agent-level execution.

Case 2. Delivery: Individual Task-Based Distributed Logistics. Addressing the challenge of scalable last-mile logistics [5, 9], the `delivery` use case demonstrates a distributed logistics operation that must continuously adapt its resource (*i.e.*, drone) allocation. We implemented this using a hybrid architecture with a dynamic, pull-based assignment strategy, where the GCS acts as a central dispatcher and dynamically assigns tasks to drones on demand, which is the optimal algorithm we found in §5.4. The results in Figure 5b prove that SwarmBox can effectively implement and manage scalable, dynamic resource allocation strategies for complex logistics scenarios.

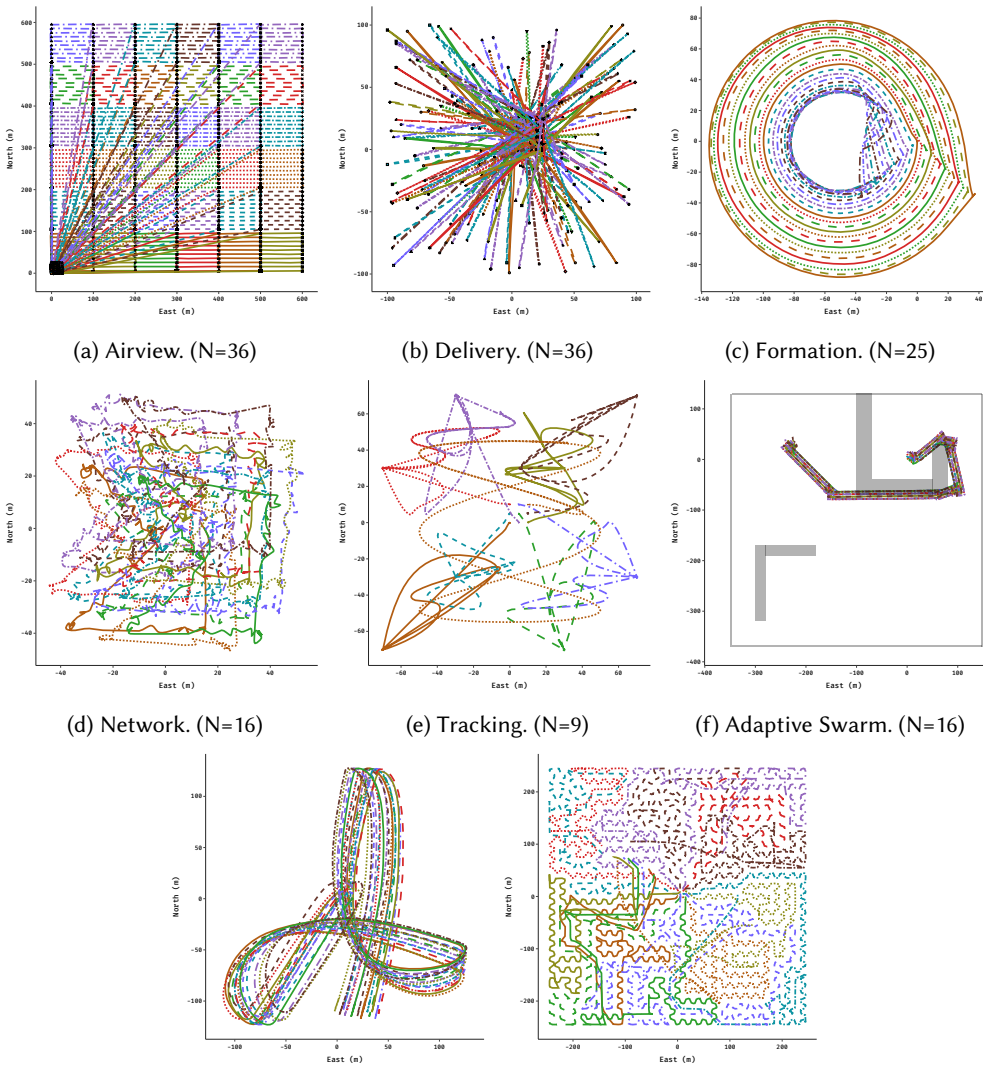
Case 3. Formation: Preplanned Cooperative Flight. As a core technology for applications like aerial light shows and cooperative transport [15, 18], the `formation` scenario demonstrates a preplanned formation flight. This mission was implemented using a leader-follower topology where GCS commands only a designated `Leader` drone. The `Follower` drones autonomously maintain a precise offset from the leader, requiring tightly-coupled inter-agent control. The high-precision trajectories in Figure 5c validate that SwarmBox effectively supports such implementations.

Case 4. Network: Demand-Responsive Coverage for Mobile Hotspot Network. To showcase an agile alternative to ground-based communication networks for disaster zones [4, 19], the `network` scenario demonstrates a self-organizing, drone-based network relay system providing demand-responsive communication coverage. Implemented with a decentralized architecture, the drones autonomously reposition themselves based on a dynamic demand heatmap broadcast by the GCS, with no direct position commands. As shown in Figure 5d, this use case proves that SwarmBox can facilitate emergent, swarm-level behavior from decentralized decisions based on simple, local rules.

Case 5. Tracking: Dynamic Target Tracking and Management. For dynamic applications like cooperative surveillance [8, 27], the `tracking` scenario demonstrates a pursuit mission reacting to a non-cooperative target. It was implemented with a centralized architecture managing agents with distinct, pre-defined roles: a single *Active Tracker* and multiple *Sector Sentinels*. The GCS directs the tracker's pursuit and orchestrates an event-triggered rendezvous between different agent groups based on the target's location. The successful rendezvous shown in Figure 5e validates that SwarmBox can manage missions with distinct agent roles and facilitate complex, event-triggered cooperation within the swarm.

Case 6, 7, 8. Ported: Validation of Three Ported Algorithms of RQ1. To validate the operational fidelity of the algorithms ported in §5.1, we executed all three reference algorithms on SwarmBox: AdaptiveSwarm [1, 26], Optimized Flocking [7, 32], and Socratic Swarm [6, 11]. While these algorithms employ fundamentally different computational paradigms, they share common characteristics: high interdependence among agents and high dynamism requiring continuous control updates. We implemented these scenarios by orchestrating their respective decentralized logic within a centralized controller to ensure synchronized execution. As shown in the trajectory in Figure 5f, we have successfully replicated the original mission that AdaptiveSwarm showcased. For the other two, we faithfully ported the core logic by adhering to the algorithmic structures presented in the original studies. The execution results presented in Figure 5g and Figure 5h confirm that the ported algorithms exhibit behavioral alignment with their original design intentions, demonstrating that SwarmBox effectively supports the high-frequency, closed-loop control required to realize these diverse, complex swarm behaviors across varying algorithmic approaches.

RQ5 Conclusion. We demonstrated that SwarmBox possesses the generality to support the vast design space of swarm robotics. By implementing eight distinct real-world missions spanning varying degrees of interdependence, dynamism, control architectures, and scaling up to 36 agents, we confirmed that the framework is not overfitted to a single domain and effectively adapts to diverse and scalable operational contexts.



(a) Airview. (N=36) (b) Delivery. (N=36) (c) Formation. (N=25)

(d) Network. (N=16) (e) Tracking. (N=9) (f) Adaptive Swarm. (N=16)

(g) Optimized Flocking. (N=16) (h) Socratic Swarm. (N=16)

Fig. 5. Executed trajectories in the eight application scenarios. N: Fleet size.

6 Related Work

Existing swarm frameworks often entail trade-offs between hardware specificity, simulation fidelity, and analysis capabilities. In contrast, SwarmBox offers an integrated, platform-agnostic solution covering the entire development lifecycle. As summarized in Table 8, we qualitatively compare SwarmBox to prominent alternatives. Please note that a direct, quantitative comparison is infeasible and potentially misleading due to fundamentally incompatible design assumptions; while prior works predominantly target centralized or indoor-specific settings, SwarmBox is architected for decentralized, outdoor operations independent of external infrastructure. Therefore, we focus on

Table 8. Architectural Comparisons between SwarmBox and Related works

Framework	Key Focus	Domain	Control	Hardware	Sim-to-Real	Analysis
SwarmBox	**Full lifecycle streamlined drone swarm development and analysis**	**Unrestricted**	**Universal***	**Agnostic**	**High**	**Integrated**
Crazyswarm [23]	Dense formation flight of swarm	Indoor	Centralized	Crazyflie	Low	Raw log
CrazyChoir [21]	Modular ROS2 swarms for Crazyflie	Indoor	Decentralized◇	Crazyflie	Low	ROS2 dump
Robotarium [22]	Remote safety testbed	Remote Lab	Centralized	GRITSBot	Medium	Raw log
SwarmLab [28]	Algorithm benchmarking via MATLAB	(Simulation)	Centralized	-	N/A	Metric-centric
SALSA [2]	Parameter tuning for logic optimization	(Simulation)	Centralized	-	N/A	Metric-centric
FANET-Sim [33]	Network protocols and communication	(Simulation)	Centralized	-	N/A	Metric-centric

* Supports centralized, decentralized, and hybrid control architectures via onboard computing.
◇ The authors claim decentralization, but the experiment was conducted with a single ground-based workstation.

differentiating these control architectures and deployment targets to highlight how SwarmBox complements the existing landscape.

Simulation and Benchmarking Tools. Several frameworks focus on the theoretical validation of swarm algorithms. SwarmLab [28] and SALSA [2] provide environments for algorithm benchmarking and parameter tuning within centralized simulation engines. Similarly, FANET-Sim [33] specializes in network protocol analysis. While these tools excel at verifying algorithmic correctness within strictly controlled, idealized environments, they often rely on simplified physics and lack a seamless pathway to deploy the same code onto real hardware, creating a significant sim-to-real gap.

Hardware Deployment Frameworks. Frameworks such as Crazyswarm [23], Robotarium [22], and CrazyChoir [21] have significantly advanced real-world swarm deployment. These platforms excel at executing high-frequency, precision maneuvers within motion-capture-equipped laboratories, enabling highly agile orchestration development. However, these systems are predominantly tailored for specific vehicles (*e.g.*, Crazyflie) within indoor environments, heavily relying on external infrastructure like motion capture systems. This dependency limits their outdoor scalability and often creates a sim-to-real gap, forcing developers to maintain separate codebases for simulation prototyping and embedded firmware. In contrast, SwarmBox leverages a hardware-abstraction layer atop standard flight stacks (*e.g.*, PX4), decoupling high-level logic from platform specifics. This architecture ensures that the exact same swarm algorithms are executed across SITL simulations and outdoor physical missions without code modification.

From Logging to Integrated Analysis. Existing frameworks primarily support data collection via raw logging (*e.g.*, standard ROS bags or CSV dumps) or aggregate performance metrics. They often lack integrated tools to correlate distributed events across multiple agents. Debugging emergent behaviors where individual actions lead to unintended swarm-level outcomes therefore remains a manual and labor-intensive process. SwarmBox addresses this by providing an Integrated Analyzer that unifies distributed logs into a coherent timeline and one integrated coordinate system, facilitating the full-lifecycle analysis of swarm software.

7 Threats to Validity

Evaluation Bias and Baseline. A primary threat to validity arises from potential author bias in assessing engineering effort reduction (§5.1). While user studies are commonly utilized for this purpose, conducting them in this specialized domain introduces significant confounding factors related to participants' prior expertise in distributed systems and flight stacks (*e.g.*, ROS 2, PX4), which can overshadow the framework's intrinsic usability. To mitigate this risk and ensure scientific validity, we prioritized objective, implementation-level complexity analysis over subjective user feedback. Specifically, we adopted a *structural code decomposition* approach, separating platform-agnostic *Core*

Logic from infrastructure *Boilerplate*. This method allows us to quantitatively demonstrate the reduction in engineering overhead across diverse algorithmic paradigms, independent of implementation language or author familiarity.

To facilitate realistic and scientifically valid usability assessments in the future, we plan to conduct a structured longitudinal assessment based on early adoption signals (e.g., forks, issues) and user feedback following the open-source release.

Scalability Limits and Simulation Artifacts. While SwarmBox is architected to support scalable interaction, our experimental validation reached an upper limit of 36 drones due to simulation environment constraints. Root cause analysis revealed that executing concurrent high-fidelity physics instances (PX4 SITL) on a single workstation triggers a "Context Switch Storm," saturating the host CPU's scheduling capacity. This resource contention leads to thread starvation for sensor simulation, causing EKF state validity failures and subsequent instability. These results confirm that the scalability ceiling is an environmental hard limit imposed by the single simulation host's processing power, rather than an architectural limitation of the SwarmBox stack. Crucially, this bottleneck does not apply to physical deployments, where computation is distributed across independent companion computers.

Hardware and Platform Diversity. Our physical validation was conducted exclusively using PX4-based quadrotors, which may limit the immediate generalizability to other flight stacks (*e.g.*, ArduPilot) or vehicle types (*e.g.*, VTOL). However, SwarmBox mitigates this dependency through its Hardware Abstraction Handler Layer (HAH), which decouples high-level swarm logic from low-level hardware interfaces. While empirical validation on diverse platforms remains future work, we are actively extending our validation scope to include various platforms independent of this submission.

Physical Experiment Constraints. The scale of our physical experiments was limited to 4 agents. This constraint was imposed by external factors, including strict safety regulations and Wi-Fi signal reachability degradation over distance. Although this cap does not reflect an inherent scalability bottleneck of SwarmBox, it highlights the logistical challenges of real-world swarm deployment. We complemented this limitation by conducting large-scale verifications in the simulation environment.

8 Conclusion

In this paper, we present SwarmBox, an open-source testbed framework designed to tackle the significant engineering overhead and lack of reproducibility that constrain progress in drone swarm research. By formalizing the architectural principle of Logic-Environment Isolation, SwarmBox rigorously decouples high-level algorithmic intent from low-level system complexities. This abstraction ensures that the exact same code executes faithfully across simulation and diverse physical hardware, effectively bridging the 'sim-to-real' gap. Our systematic evaluation demonstrates that the framework drastically reduces development effort and achieves quantitatively consistent performance in physical deployment with unmodified application code. Crucially, we demonstrated that the framework achieves Swarm-Level Observability via Causal Reconstruction. By synchronizing distributed event streams, the integrated analyzer successfully detects emergent interaction failures, which is a critical blind spot for traditional decentralized logging techniques. Beyond technical integration, SwarmBox serves as a standardized scientific instrument that enables the rigorous, reproducible comparison of diverse algorithms. Ultimately, by providing a common foundation for community-driven open development, we expect that SwarmBox will contribute to accelerating innovation in drone swarm technology.

Data Availability

The proposed framework is released as open-source to help advance and accelerate future drone swarming research. The artifacts containing our testbed framework, demonstration scenarios, usage tutorials, and all necessary scripts and data for reproducing the results presented in this paper—including the raw experimental logs and aggregated values obtained from multiple runs—are publicly available at https://github.com/postech-compsec/swarmbox. Furthermore, the exact version of the artifact evaluated in this paper is permanently archived on Zenodo [16]. The source code is provided under the MIT License, and the data is licensed under CC-BY 4.0.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and constructive feedback that helped improve this paper. We are also grateful to Jangseop Choi, Junwoong Doh, Jongsoo Han, and Chiheon Kim from the POSTECH CompSec Lab for their field setup assistance with the physical drone experiments. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-23523174), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) - ITRC (Information Technology Research Center) grant funded by the Korea government (MSIT) (IITP-2026-RS-2024-00437866 and IITP-2026-RS-2026-25529760).

References

[1] Ruslan Agishev. 2019. *Adaptive Control of Swarm of Drones for Obstacle Avoidance.* Master's thesis. Skolkovo Institute of Science and Technology.

[2] Joel Beedle, Calum Imrie, and Radu Calinescu. 2024. SALSA: Swarm Algorithm Simulator. In *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 140–145. doi:10.1109/ACSOS-C63493.2024.00045

[3] Markus Bernard, Konstantin Kondak, Ivan Maza, and Anibal Ollero. 2011. Autonomous transportation and deployment with aerial robots for search and rescue missions. *Journal of Field Robotics* 28, 6 (2011), 914–931. doi:10.1002/rob.20401

[4] Lorenzo Bertizzolo, Salvatore D'oro, Ludovico Ferranti, Leonardo Bonati, Emrecan Demirors, Zhangyu Guan, Tommaso Melodia, and Scott Pudlewski. 2020. SwarmControl: An automated distributed control framework for self-optimizing drone networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 1768–1777. doi:10.1109/INFOCOM41043.2020.9155231

[5] Francesco Betti Sorbelli. 2024. UAV-Based Delivery Systems: A Systematic Review, Current Trends, and Research Challenges. 1, 3 (2024). doi:10.1145/3649224

[6] Breakend. 2017. SocraticSwarm. https://github.com/Breakend/SocraticSwarm. [Online; accessed 2026-03-26], Version/Commit: 0c4728c.

[7] csviragh. 2018. robotsim. https://github.com/csviragh/robotsim. [Online; accessed 2026-03-26], Version/Commit: ea54b97.

[8] Luis C Batista Da Silva, Ricardo Maroquio Bernardo, Hugo A De Oliveira, and Paulo FF Rosa. 2017. Multi-UAV agent-based coordination for persistent surveillance with dynamic priorities. In *2017 International Conference on Military Technologies (ICMT)*. IEEE, 765–771. doi:10.1109/MILTECHS.2017.7988859

[9] Kevin Dorling, Jordan Heinrichs, Geoffrey G Messier, and Sebastian Magierowski. 2016. Vehicle routing problems for drone delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 1 (2016), 70–85. doi:10.1109/TSMC.2016.2582745

[10] Joel George, Sujit PB, and João B Sousa. 2011. Search strategies for multiple UAV search and destroy missions. *Journal of Intelligent & Robotic Systems* 61, 1 (2011), 355–367. doi:10.1007/s10846-010-9486-8

[11] Peter Henderson, Matthew Vertescher, David Meger, and Mark Coates. 2018. Cost adaptation for robust decentralized swarm behaviour. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4099–4106. doi:10.1109/IROS.2018.8594283

[12] Miquel Kegeleirs and Mauro Birattari. 2025. Towards applied swarm robotics: current limitations and enablers. *Frontiers in Robotics and AI* 12 (2025), 1607978. doi:10.3389/frobt.2025.1607978

[13] Ruslan Kirichek, Andrei Vladyko, Alexander Paramonov, and Andrey Koucheryavy. 2017. Software-defined architecture for flying ubiquitous sensor networking. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 158–162. doi:10.23919/ICACT.2017.7890076

[14] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. IEEE, 2149–2154. doi:10.1109/IROS.2004.1389727

[15] Alex Kushleyev, Daniel Mellinger, Caitlin Powers, and Vijay Kumar. 2013. Towards a swarm of agile micro quadrotors. *Autonomous Robots* 35, 4 (2013), 287–300. doi:10.1007/s10514-013-9349-9

[16] Minki Lee, Seojin Lee, and Seulbae Kim. 2026. Artifact for SwarmBox: A Plug-and-Play Drone Swarm Framework for Streamlined Development and Comprehensive Analysis. Zenodo. doi:10.5281/zenodo.19344308

[17] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. 2015. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 6235–6240. doi:10.1109/ICRA.2015.7140074

[18] Daniel Mellinger, Michael Shomin, Nathan Michael, and Vijay Kumar. 2013. Cooperative grasping and transport using multiple quadrotors. In *Distributed Autonomous Robotic Systems: The 10th International Symposium*. Springer, 545–558. doi:10.1007/978-3-642-32723-0_39

[19] Arvind Merwaday and Ismail Guvenc. 2015. UAV assisted heterogeneous networks for public safety communications. In *2015 IEEE wireless communications and networking conference workshops (WCNCW)*. IEEE, 329–334. doi:10.1109/WCNCW.2015.7122576

[20] Gerardo Pardo-Castellote. 2003. OMG Data-Distribution Service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 200–206. doi:10.1109/ICDCSW.2003.1203555

[21] Lorenzo Pichierri, Andrea Testa, and Giuseppe Notarstefano. 2023. Crazychoir: Flying swarms of crazyflie quadrotors in ROS 2. *IEEE Robotics and Automation Letters* 8, 8 (2023), 4713–4720. doi:10.1109/LRA.2023.3286814

[22] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. 2017. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1699–1706. doi:10.1109/ICRA.2017.7989200

[23] James A Preiss, Wolfgang Honig, Gaurav S Sukhatme, and Nora Ayanian. 2017. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3299–3304. doi:10.1109/ICRA.2017.7989376

[24] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, 5.

[25] Craig W Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 25–34. doi:10.1145/37402.37406

[26] rlnav. 2024. adaptive_swarm. https://github.com/rlnav/adaptive_swarm. [Online; accessed 2026-03-26], Version/Commit: e7b0797.

[27] Eduard Semsch, Michal Jakob, Dušan Pavlicek, and Michal Pechoucek. 2009. Autonomous UAV surveillance in complex urban environments. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 2. IEEE, 82–85. doi:10.1109/WI-IAT.2009.132

[28] Enrica Soria, Fabrizio Schiano, and Dario Floreano. 2020. SwarmLab: A MATLAB drone swarm simulator. In *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 8005–8011. doi:10.1109/IROS45743.2020.9340854

[29] Evgeny Tsykunov, Ruslan Agishev, Roman Ibrahimov, Luiza Labazanova, Akerke Tleugazy, and Dzmitry Tsetserukou. 2019. Swarmtouch: Guiding a swarm of micro-quadrotors with impedance control using a wearable tactile interface. *IEEE transactions on haptics* 12, 3 (2019), 363–374. doi:10.1109/TOH.2019.2927338

[30] u-blox AG. 2025. *MAX-M10S Standard Precision GNSS Module Data Sheet (UBX-20035208).* u-blox AG, Thalwil, Switzerland. Available at https://www.u-blox.com.

[31] vasarhelyi. 2017. Ardupilot/CMCopter-3.4. https://github.com/collmot/ardupilot/tree/CMCopter-3.4. [Online; accessed 2026-03-26], Version/Commit: 384af07.

[32] Gábor Vásárhelyi, Csaba Virágh, Gergő Somorjai, Tamás Nepusz, Agoston E Eiben, and Tamás Vicsek. 2018. Optimized flocking of autonomous drones in confined environments. *Science Robotics* 3, 20 (2018), eaat3536. doi:10.1126/scirobotics.aat3536

[33] Chengyu Xie, Yi Li, Tao Zhang, and Chuanfu Zhang. 2025. FANET-Sim: A Simulation Framework for UAV Swarm Communication. In *Proceedings of the 2025 5th International Conference on Computer Network Security and Software Engineering*. 183–189. doi:10.1145/3732365.3732396

Received 2026-02-18; accepted 2026-03-24