SwiftBot: A Lightweight Platform for Adaptive Multi-Robot Task Execution at the Edge

Anonymous Authors
Anonymous Institution
anonymous@email.com

Abstract—**Multi-robot task execution systems struggle to bridge natural language instructions to robot control while efficiently managing distributed resources. Existing approaches face three limitations: rigid hand-coded planners requiring extensive domain engineering, centralized coordination architectures that limit scalability as robot sizes grow, and naive resource management failing to adapt to dynamic workload patterns in distributed edge environments. We present SwiftBot, a distributed system integrating LLM-based task decomposition with intelligent container orchestration over a DHT overlay. SwiftBot achieves 94.3% decomposition accuracy across diverse tasks without domain-specific training, reduces startup latency to 120ms median (10-20× improvement), and maintains 1.6-2.1× better performance than local-only approaches under high load through distributed warm container migration. Evaluation on multimedia tasks validates that co-designing semantic understanding and resource management enables both flexibility and efficiency for real-time robotic control.**

Index Terms—**Multi-Robot Collaboration, Robot Task Planning, Container, P2P**

I. Introduction

The proliferation of intelligent robotic systems has created an urgent need for platforms that can bridge the semantic gap between high-level human instructions and low-level robot control while efficiently managing distributed computational resources. Modern robotic applications (e.g., warehouse automation to collaborative manufacturing) require systems that can (1) interpret natural language commands and decompose them into executable subtasks, (2) dynamically allocate heterogeneous computational resources across distributed infrastructure, and (3) maintain real-time responsiveness despite fluctuating workloads and resource constraints.

Traditional robotic task execution systems face three fundamental limitations. First, they rely on rigid, hand-coded task planners that require extensive domain engineering to handle task variations and environmental dynamics, making them brittle when faced with novel instructions or changing operational contexts. Second, they employ centralized coordination architectures where a single controller node becomes both a performance bottleneck and a single point of failure as fleet sizes grow. Third, they utilize static resource allocation strategies that either over-provision containers (wasting scarce edge resources) or under-provision them (causing cold-start delays that violate latency requirements). The cold-start problem is particularly severe: launching a fresh container for each task invocation incurs 4-8 seconds of overhead from image pulling, filesystem initialization, and runtime setup—an unacceptable delay for real-time robotic control loops operating at 10-50Hz.

Existing multi-robot task execution systems face following limitations that hinder their deployment in dynamic, resource-constrained environments. First, traditional robotic task planning relies on hand-coded domain models using formal languages like PDDL (Planning Domain Definition Language), which require extensive domain engineering to capture task variations and environmental conditions [1], [2]. These rigid, rule-based planners face fundamental brittleness when confronted with novel task types or unexpected environmental dynamics [3], [4], as they lack the semantic generalization capabilities needed to adapt beyond their predefined action spaces. Second, centralized coordination architectures (i.e., a single controller node manages task allocation and execution) introduce performance bottlenecks and single points of failure that limit scalability as fleet sizes grow [5], [6]. Recent work on multi-robot coordination demonstrates that centralized approaches, while achieving high efficiency in controlled environments, become increasingly untenable in distributed edge-cloud deployments where network latency and node heterogeneity introduce significant coordination overhead [7], [8].

Our Solution. We propose SwiftBot, a distributed robotic task execution system that synthesizes LLM-based semantic understanding with intelligent container orchestration to achieve both adaptive task planning and efficient resource utilization. SwiftBot decentralizes coordination responsibilities across a distributed hash table (DHT) overlay network, where any edge node can dynamically assume roles as task decomposer, scheduler, executor, or coordinator. The system architecture integrates three tightly coupled components: (1) an LLM-powered Task Decomposer that transforms natural language instructions into structured execution plans with explicit subtask dependencies and resource requirements, (2) a Dynamic Task Manager that orchestrates decentralized scheduling and monitoring across the DHT overlay while supporting dynamic task migration to handle load imbalances and node failures, and (3) a Warm-start Container Pool that maintains pre-initialized execution environments distributed across cluster nodes, enabling sub-second task startup through intelligent container reuse and migration.

The fundamental innovation lies in co-designing task decomposition and container orchestration to exploit cross-layer optimization opportunities. Rather than treating semantic planning and resource management as independent concerns, SwiftBot's Task Decomposer generates container allocation hints that guide warm-pool selection, while execution feedback informs future decomposition decisions through a closed-loop adaptation mechanism. The DHT-based architecture eliminates centralized bottlenecks by distributing both task metadata and warm container inventories across nodes, achieving $O(\log N)$ lookup complexity for resource discovery in an N-node cluster. The warm-start pooling strategy addresses the cold-start problem through predictive pre-warming (proactively initializing containers based on workload patterns) and cross-node container migration (transferring pre-initialized containers between nodes when migration cost is less than cold-start penalty), reducing typical task startup latency from 4-8 seconds to 120-450 milliseconds—a 10-20× improvement sufficient for real-time control.

Contributions. SwiftBot introduces three following innovations:

1) LLM-Powered Task Decomposition with Container-Aware Planning: A semantic decomposition layer that uses large language models to parse natural language instructions and generate structured execution plans including subtask dependencies, parallelization opportunities, and container allocation hints.
2) DHT-Based Distributed Container Orchestration: A decentralized scheduling algorithm that performs three-phase container selection with migration decisions guided by a cost model comparing migration overhead against cold-start penalty.
3) Optimistic Concurrency Control for Dynamic Task Migration: A coordination protocol that enables parallel migration decisions across distributed coordinators while maintaining consistency through deterministic conflict resolution using load metrics and container availability scores.

We evaluate SwiftBot on a distributed testbed with 10 heterogeneous nodes (including GPU-equipped servers and CPU-only edge devices) using 1,200 realistic multimedia processing tasks derived from UCF101 (video action recognition) and LibriSpeech (speech transcription) datasets. These workloads naturally decompose into multi-stage container pipelines that mirror robotic perception and reasoning workflows. Results demonstrate that SwiftBot achieves 120ms median task startup latency (5.4× faster than cold-start baseline) and 280ms P99 latency (3.4× improvement), maintaining low latency even under 200 tasks/second load where baseline approaches degrade to 1,400ms+ P99 latency.

This paper is organized as follows: In section V we introduce the background and motivation of this work. In section III, we present the workflow and functional components design. We perform the evaluation for SwiftBot and baseline executions to show the improvements in section IV. In section VI, we discuss the current limitations and future works of this work and summarize the conclusion in section VII.

II. Motivation and Background

A. From Single Robots to Federated Fleets

Industrial and service robots are no longer isolated, single-purpose machines. According to the latest World Robotics statistics, factories installed about 542,000 new industrial robots in 2024 alone, bringing the global operational stock to approximately 4.66 million units, more than double the number a decade ago [5]. These robots increasingly operate as fleets of autonomous mobile robots (AMRs), collaborative manipulators, and service robots sharing constrained edge and IoT infrastructure.

At the same time, next-generation wireless technologies (5G/6G) are being designed with ultra-low latency applications such as teleoperation, mobile manipulation, and tactile Internet in mind. Recent surveys report user-plane latency targets on the order of 4ms for enhanced mobile broadband and 1ms for ultra-reliable low-latency communication in 5G, while 6G vision documents aim for end-to-end latencies below 1 ms [9]. These numbers are well aligned with require- ments reported for factory AMRs, closed-loop motion control, and teleoperation, which typically demand sub-10ms response times for safe and precise behavior [10], [11].

This combination of massive scale (millions of robots) and millisecond latency requirements fundamentally changes how robot software must be deployed and coordinated. Hard-coding fixed roles for individual robots, or assuming a single central controller in a data center, is no longer sufficient: robot fleets must be able to form groups, share models, and reassign computation at run time in response to changing tasks, failures, and wireless conditions [12]–[14].

B. Federated Learning for Collaborative Robotics

Federated learning (FL) has emerged as a natural paradigm for exploiting distributed data in such fleets while preserving privacy and reducing communication overhead. Instead of streaming raw sensor logs to a cloud server, robots train local models and only exchange model updates.

In robotics, several recent systems demonstrate the potential of FL-style collaboration. Gutierrez et al. integrate FL into ROS 2 and show that a fleet of simulated and physical robots can collaboratively learn navigation policies that generalize better and exhibit more stable rewards as the number of agents increases [6]. Liang et al. propose Federated Transfer Reinforcement Learning for autonomous driving, where simulation agents and real RC cars share policy updates through a federated architecture. These results provide concrete evidence that sharing policy updates across distributed agents can significantly improve safety and task efficiency.

However, existing FL systems for robots largely assume either a relatively stable topology (e.g., robots remain connected to a central server), or focus purely on the learning algorithm rather than the orchestration layer. In practice, edge robot platforms must cope with:

- **Intermittent connectivity:** Robots routinely lose contact with edge servers or with one another due to mobility, RF dead zones, or interference [15], [16].

- **Heterogeneous hardware:** Fleets mix low-power SoC boards and more capable edge nodes, with highly variable CPU, GPU, and memory budgets [14].
- **Dynamic workloads:** Tasks and human instructions arrive online, with different compute, memory, and latency requirements over time [13].

Most current FL frameworks treat the robot as a relatively static "client" and do not address how those clients (and their compute containers) should be created, placed, migrated, or recovered in a resource-constrained edge cluster. This leaves a gap between algorithmic advances in FL and the systems support needed to run FL-enabled multi-robot applications at scale.

SwiftBot is designed precisely to fill this gap. It provides a containerized, peer-to-peer execution layer where any edge node (e.g., robot, gateway, or edge server) can dynamically assume the role of FL client, coordinator, task manager, or aggregation node. A DHT-based overlay enables scalable discovery and group formation, while a warm-start container pool allows pre-initialized FL clients and coordination services to be activated or migrated in sub-second time without cold-start overheads. By explicitly integrating federated learning, multi-robot task orchestration, and edge-native migration into a single platform, SwiftBot aims to bring the benefits demonstrated in prior FL and cloud-robotics studies into a runtime that can sustain large-scale federated multi-robot collaboration at the edge.

III. System Design

We propose a novel distributed robotic task execution system that bridges the semantic gap between natural language human instructions and low-level robot control through intelligent orchestration of containerized compute resources, named SwiftBot. As shown in Figure 1, SwiftBot consists of three primary components: the Task Decomposer, which employs LLM agents to analyze task characteristics and generate execution plans; the Dynamic Task Manager, which orchestrates task scheduling and monitoring across a DHT-overlay network; and the Warm-start Container Pool, which maintains pre-initialized execution environments to minimize cold-start latency.

Human instructions flow through the frontend to the Task Decomposer, where LLM agents perform semantic analysis and hierarchical task planning. The resulting task specifications are dispatched to Robot Task Execute Coordinators through the Dynamic Task Manager, which leverages a DHT-overlay for decentralized resource discovery and load balancing. Throughout execution, the system maintains a feedback loop where execution monitoring informs future task decomposition decisions, enabling continuous adaptation to evolving workload patterns and system dynamics. This design philosophy prioritizes three key properties: semantic adaptability through learned task understanding, elastic scalability through container orchestration, and operational efficiency through predictive resource pre-allocation.

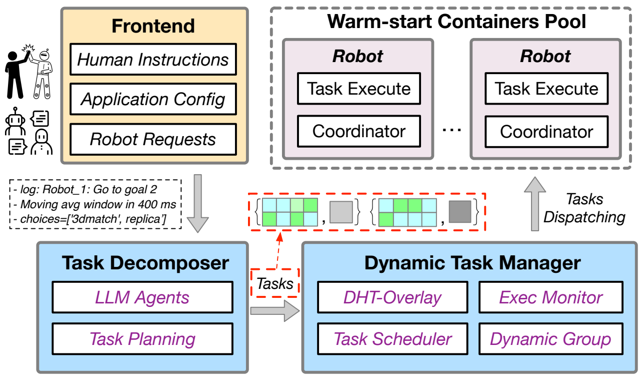


Fig. 1: The overview of SwiftBot workflow and structure.

A. Task Decomposer

The Task Decomposer serves as the cognitive layer of the system, transforming high-level human instructions and application configurations into structured, executable task specifications. It receives natural language commands from the frontend (e.g., Robot_1: Go to goal 2") and application-specific parameters (such as algorithmic choices like `['3dmatch', 'replica']` for scene matching, or timing constraints like moving avg window in 400 ms"), then produces detailed task execution plans that specify subtask dependencies, resource requirements, and scheduling constraints.

The component operates through three integrated stages. First, the *semantic analysis* stage uses LLM agents to parse instructions, extract parameters, and classify task types across six robotic domains (navigation, manipulation, perception, multi-robot coordination, inspection, and human-robot interaction). Second, the *hierarchical decomposition* stage generates a directed acyclic graph (DAG) of subtasks, identifying parallelization opportunities and data dependencies while respecting application-defined constraints. Third, the *container allocation strategy* stage produces resource hints that guide the Dynamic Task Manager in selecting appropriate containers from the warm-start pool, considering factors such as data locality, container reuse patterns, and load balancing objectives.

Traditional robotic task planning systems face a fundamental brittleness problem requiring extensive domain engineering to capture task variations and environmental conditions. We adopt LLM-based decomposition because it provides three critical advantages: (1) *semantic generalization*, LLMs' common-sense knowledge enables handling novel task types; (2) *contextual reasoning*, for instance, 200K token context windows allow holistic consideration of system state, history, and configurations to balance multiple objectives; and (3) *rapid adaptation*, prompt engineering enables behavioral changes in hours rather than months of algorithmic rewrites. The primary challenge is managing inference latency, which we address through a hybrid cascade architecture routing routine tasks to fast small models while reserving large models for complex scenarios, reducing latency for real-time operation.

B. Dynamic Task Manager

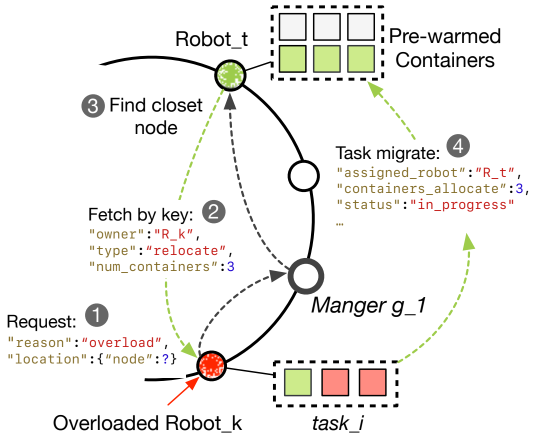
The Dynamic Task Manager orchestrates task execution across the distributed cluster through three tightly integrated subsystems: the Task Scheduler, Execution Monitor, and DHT-Overlay network. Upon receiving task specifications from the Task Decomposer, the Task Scheduler performs resource-aware placement, assigning subtasks to specific Robot Task Execute Coordinators based on current load, data locality, and container availability. The DHT-Overlay provides a decentralized substrate for resource discovery, using consistent hashing to distribute task state and coordinator metadata across cluster nodes, enabling $O(logN)$ lookup complexity for N nodes.

The Task Scheduler implements a two-level scheduling algorithm. At the coarse level, it selects target DHT nodes for subtask placement using a scoring function that considers node load, network proximity to required data, and warm container availability. At the fine level, each Robot Task Execute Coordinator performs local scheduling of subtasks assigned to its node, managing container lifecycle and enforcing resource limits. This hierarchical approach achieves both global load balancing and local optimization.

The Execution Monitor maintains real-time visibility into task progress through structured logging and metrics collection. As subtasks execute, coordinators stream execution traces (e.g., Robot_1: Go to goal 2 - [timestamp] path planned, [timestamp] executing trajectory") to the monitor, which aggregates this data to compute performance metrics such as moving average execution windows (e.g., 400 ms" for navigation tasks). When failures occur, the monitor triggers re-planning by sending failure context back to the Task Decomposer, enabling adaptive recovery strategies. The DHT-Overlay implements a Chord-based distributed hash table that maps task identifiers and resource keys to responsible cluster nodes. This design provides three critical properties: (1) fault tolerance through successor replication, ensuring task state survives individual node failures; (2) load balancing through consistent hashing, preventing hot spots; and (3) scalability, supporting cluster growth without global coordination.

The DHT-Overlay supports dynamic formation of execution groups, temporary coalitions of nodes that coordinate multi-robot tasks. When the Task Decomposer generates plans requiring tight coordination (e.g., collaborative manipulation), the scheduler creates overlay multicast groups that enable efficient inter-robot communication without flooding the entire network. These groups dissolve automatically upon task completion, maintaining system flexibility.

As shown in Figure 2, SwiftBot implements a four-step dynamic task migration mechanism to address resource contention in distributed robot systems. Step 1 (Request): When an overloaded robot (`Robot_k`) detects resource constraints such as high CPU utilization or insufficient container capacity, it sends a migration request to the DHT-Overlay. This request includes the migration reason (`"overload"`), current node location, task metadata (`operation type: "relocate"`), and resource requirements (e.g., `num_containers: 3`).



```
"assigned_robot":"R_t",
"containers_allocate":3,
"status":"in_progress"
…
```

```
"owner":"R_k",
"type":"relocate",
"num_containers":3
```

```
"reason":"overload",
"location":{"node":?}
```

Fig. 2: Reschedule a task based on runtime resources of robots and task needs.

Step 2 (Fetch by Key): The DHT-Overlay performs a distributed lookup using consistent hashing on the task key. The request is routed through intermediate nodes (e.g., `Manager_g1`) to locate robots with available capacity. Each DHT node maintains a local resource registry that tracks container availability and current load metrics across the network. Step 3 (Find Closest Node): The system evaluates candidate nodes using multiple criteria: availability of pre-warmed containers matching task requirements, current load levels, network distance to required data, and previous execution history. Based on these factors, the system selects `Robot_t` as the target node, which has available pre-warmed containers that match the task specifications. Step 4 (Task Migrate): The migration is completed by updating DHT entries to reflect the new assignment (`assigned_robot: R_t`), allocating the required containers (`containers_allocate: 3`), and updating the task status to "`in_progress`". The system transfers task state including partial results and execution checkpoints to the target robot to avoid redundant computation. To maintain consistency during concurrent migrations, the DHT uses version vectors with optimistic concurrency control, where each task entry includes a version number that increments with updates and conflicts are resolved using load metrics and container availability scores.

The migration mechanism's primary novelty lies in its integration of container-aware scheduling with DHT coordination, enabling decentralized load balancing without centralized bottlenecks. Unlike traditional migration approaches that rely on global schedulers, the DHT-Overlay architecture allows each robot coordinator to independently discover and negotiate with target nodes while maintaining system-wide consistency through version vectors and deterministic conflict resolution. The optimistic concurrency control mechanism further enhances scalability by allowing *parallel migration decisions* while gracefully handling conflicts through load-based resolution policies, making the system particularly effective for dynamic multi-robot environments with fluctuating workloads.

C. Warm-start Container Pool

The Warm-start Container Pool maintains a dynamically sized inventory of pre-initialized containers ready for immediate task execution, eliminating the cold-start latency associated with container image pulls, filesystem layer extraction, and application initialization. The pool operates across all cluster nodes, with each node managing a local subset of warm containers based on its hardware capabilities and historical workload patterns.

Container management involves four continuous activities. First, *predictive pre-warming*: the system analyzes incoming task patterns and execution monitor feedback to predict which container types will be needed in the near future, proactively initializing containers before demand arrives. Second, *intelligent eviction*: when pool capacity limits are reached, the system evicts containers using a learned policy that considers access frequency, initialization cost, and resource consumption. Third, *state preservation*: for stateful containers (e.g., those with loaded neural network weights or cached map data), the pool maintains persistent volumes that survive container restarts, further reducing initialization time. Fourth, *health monitoring*: the pool periodically probes warm containers to ensure they remain responsive, automatically replacing containers that have entered degraded states.

Container selection occurs through a two-phase process. When the Task Scheduler assigns a subtask to a Robot Task Coordinator, it includes container hints from the Task Decomposer specifying preferred images, resource requirements, and data volume needs. The coordinator first attempts to allocate a matching warm container from its local pool. If no suitable container exists locally, it queries neighboring nodes in the DHT-Overlay for available containers, potentially migrating warm containers between nodes if the performance benefit justifies the transfer cost. Only when no warm containers match the specification does the system fall back to cold-start initialization.

Algorithm 1 formalizes the container selection and allocation process. This algorithm seeks to minimize task startup latency by prioritizing the reuse of warm containers available either locally or across the distributed DHT overlay. The procedure first inspects the local warm pool P_{local} and evaluates each ready container using the scoring function $\text{Score}(c, T)$, which quantifies how well a container c matches the task specification T. This score aggregates three factors: an image-compatibility term s_{img} that equals 1 when $c_{\text{img}} = T_{\text{img}}$ and 0.5 when the image is merely compatible; a resource-sufficiency term $s_{\text{res}} = \min(c_{\text{cpu}}/T_{\text{cpu}},\, c_{\text{mem}}/T_{\text{mem}})$ reflecting the proportion of required CPU and memory satisfied by c; and a volume-overlap term $s_{\text{vol}} = |c_{\text{vol}} \cap T_{\text{vol}}|/|T_{\text{vol}}|$ indicating how many of the task's data volumes are already present in the container. These terms are combined using fixed weights to produce $\text{Score}(c, T) \in [0, 1]$. If a sufficiently high-scoring container exists locally, it is selected and assigned. Otherwise, the algorithm queries a set of successor nodes in the DHT to retrieve remote candidates and evaluates them using both

Algorithm 1 Container Selection and Allocation

```
Require: Task $T = (T_{\text{img}}, T_{\text{cpu}}, T_{\text{mem}})$
Require: Local warm pool $P_{\text{local}}$
Require: DHT overlay network $\mathcal{D}$
Ensure: Allocated container instance for $T$
 1: Phase 1: Local Warm-Pool Search
 2: $\mathcal{C} \leftarrow \{\, c \in P_{\text{local}} \mid c.\text{status} = \texttt{ready} \wedge \text{Compat}(c, T) = 1 \,\}$
 3: if $\mathcal{C} \neq \emptyset$ then
 4:     $c^* \leftarrow \arg\max_{c \in \mathcal{C}} \text{Score}(c, T)$
 5:     return $\text{AllocateLocal}(c^*, T)$
 6: end if
 7: Phase 2: Distributed Search
 8: $\mathcal{N} \leftarrow \mathcal{D}.\text{Successors}(\text{hash}(T_{\text{img}}), k)$
 9: $\mathcal{R} \leftarrow \emptyset$
10: for each $n \in \mathcal{N}$ do
11:     $\mathcal{P}_n \leftarrow \mathcal{D}.\text{Query}(n, T_{\text{img}})$
12:     for each candidate $c \in \mathcal{P}_n$ do
13:         if $c.\text{status} = \texttt{ready}$ then
14:             $s \leftarrow \text{Score}(c, T)$
15:             $\eta \leftarrow \text{MigCost}(c, n, n_{\text{local}})$
16:             if $s \geq \theta_{\text{match}}$ and $\eta < \gamma \cdot \text{ColdCost}(T)$ then
17:                 $\mathcal{R} \leftarrow \mathcal{R} \cup \{(c, s, n)\}$
18:             end if
19:         end if
20:     end for
21: end for
22: if $\mathcal{R} \neq \emptyset$ then
23:     $(c^*, s^*, n^*) \leftarrow \arg\max_{(c,s,n) \in \mathcal{R}} s$
24:     return $\text{Migrate}(c^*, n^*, n_{\text{local}}, T)$
25: end if
26: Phase 3: Cold-Start Fallback
27: return $\text{ColdStart}(T)$
```

the score and the migration cost $\text{MigCost}(c, n_{\text{src}}, n_{\text{dst}})$. This cost models the time required to migrate a warm container and consists of the network transfer time of its state c_{state} over the bandwidth between nodes, plus serialization and deserialization overhead. A remote container is considered only if its migration cost is less than a fraction of the cold-start cost $\text{ColdCost}(T)$, which represents the latency of fetching the container image and performing initialization from scratch. Among all feasible remote candidates, the one with the highest score is migrated. If no local or remote warm container meets the selection criteria, the algorithm falls back to $\text{ColdStart}(T)$ to launch a fresh container. This staged decision process ensures that the scheduler exploits warm-container reuse whenever beneficial while avoiding migrations whose cost outweighs their advantage.

We adopt warm-start pooling because it provides following critical advantages: First, maintaining pre-initialized containers reduces task startup time. This improvement is essential for achieving end-to-end latency targets specified in application configurations. Second, unlike persistent containers that occupy resources regardless of utilization, the pool dynamically adjusts capacity based on demand. During low-traffic periods, the pool contracts to free resources for other workloads; during peak demand, it expands to maximize hit rates. Third, the pool automatically adapts to evolving task distributions without manual reconfiguration. When a warehouse introduces new inspection tasks, the pool observes increased demand for perception containers and automatically adjusts its composition. This eliminates the operational burden of manually tuning container deployment policies as application requirements change. Fourth, by replacing containers after each task or group of tasks, the pool prevents state accumulation and resource leaks that plague long-running containers. This “phoenix container” pattern improves system reliability.

IV. Evaluation

Testbeds. The testbed consists of one primary server node and three worker nodes distributed across a local network environment. The primary server node is equipped with an Intel Xeon W-2295 processor (18 cores at 3.0GHz), 256GB DDR4 RAM, NVIDIA RTX 5090 GPU (32GB VRAM), and 4TB NVMe SSD storage, serving as both the DHT coordinator and high-performance worker for GPU-intensive tasks. The worker nodes include one m5.4xlarge instance with 16 CPU cores and 64GB memory each and one t3.2xlarge instance with 8 CPU cores and 32GB memory each, creating a realistic heterogeneous environment similar to production edge-cloud deployments. All nodes run Ubuntu 22.04 LTS with Docker 25.0.3. The DHT overlay network implements a Chord-based distributed hash table with consistent hashing for container image routing and k=5 successor nodes queried during remote warm container discovery. Each node maintains a local warm container pool with capacity proportional to available memory.

Workloads. We employ two widely-used multimedia processing datasets to construct realistic container-based task workloads. From the UCF101 action recognition dataset, we extract 100 video processing tasks requiring multi-stage pipelines: frame extraction, feature encoding, and classification. From the LibriSpeech ASR corpus [55], we construct 100 audio transcription tasks with sequential processing stages: audio preprocessing, acoustic modeling, and language model correction. [HX: Simon, I need you to verify how many specific tasks/data we used in evaluation]

Baseline. We compare SwiftBot with two representative baseline approaches. The cold-start baseline implements the traditional serverless model where every task invocation launches a fresh container from scratch, requiring complete image pull from the registry, filesystem initialization, and runtime environment setup.

The local-warm pool baseline implements container keep-alive strategies. This baseline maintains a per-node pool of recently-used warm containers with a fixed 10-minute keep-alive TTL, allowing immediate container reuse for subsequent invocations of the same function type on the same node. However, it lacks any cross-node coordination. When a node’s local warm pool is exhausted or lacks a compatible container, it falls back to cold-start even if other nodes have idle warm containers available.

TABLE I: Task Decomposition Performance Across LLM Models

Model	Decomp. Acc. (%)	Parallel. Recall (%)	Avg. Latency (ms)	Cost/ 1K
GPT-4o	94.3	91.5	1,280	\$11.80
Claude 3.5	93.1	89.8	1,040	\$9.20
GPT-4o-mini	88.2	82.4	450	\$0.58
Llama 3.1-70B	85.6	78.6	680	\$1.95

A. LLM-Based Task Decomposition Performance

To evaluate the system’s ability to generate efficient container allocation strategies, we leverage two datasets: UCF101 (video action recognition) and LibriSpeech (speech recognition). While these datasets are not inherently multi-robot tasks, they naturally decompose into parallel processing pipelines that mirror real robotic workloads. For example, we simulate a perception task requiring multi-stage processing as: ”Process video: extract frames using FFmpeg, run ResNet50 inference, classify with lightweight model.” The LLM must decompose this into 3 subtasks with proper dependencies and identify that frame extraction and inference can run on different robots. We evaluate four LLM models to assess the accuracy-latency-cost trade-off: GPT-4o, Claude 3.5 Sonnet, GPT-4o-mini, and Llama 3.1-70B.

Table I compares four LLM models on task decomposition for container-based workloads, where the Dependency Accuracy shows how correctly the LLM identifies dependencies between subtasks and Parallelization Recall represents the percentage of parallelizable subtasks does the LLM correctly identify. GPT-4o achieves the highest decomposition accuracy (94.3%) and parallelization recall (91.5%), correctly identifying which subtasks can execute concurrently to minimize overall latency, but requires 1,280ms average inference time at \$11.80 per 1K tasks. Claude 3.5 Sonnet offers competitive accuracy (93.1%) with 89.8% parallelization recall and 19% lower latency (1,040ms). GPT-4o-mini delivers 88.2% accuracy and 82.4% parallelization recall at 2.8× faster response (450ms) and 20× lower cost (\$0.58), making it suitable for routine decomposition where some parallelization opportunities can be missed without critical impact. The 9.1% gap in parallelization recall between GPT-4o and GPT-4o-mini (91.5% vs 82.4%) justifies our cascade architecture: routing simple sequential tasks to fast models while reserving large models for complex parallel decomposition achieves both throughput and cost-efficiency.

B. Performance of Federated Applications

To validate the effectiveness of SwiftBot in federated tasks, we conducted experiments using the UCF101 and LibriSpeech datasets across varying client fleet sizes ($N \in \{8, 16, 24, 32\}$). Figure 3 and 4 present comparisons between SwiftBot and the local federated learning baseline. In terms of model convergence, Figures 3 and 4 demonstrate that SwiftBot maintains

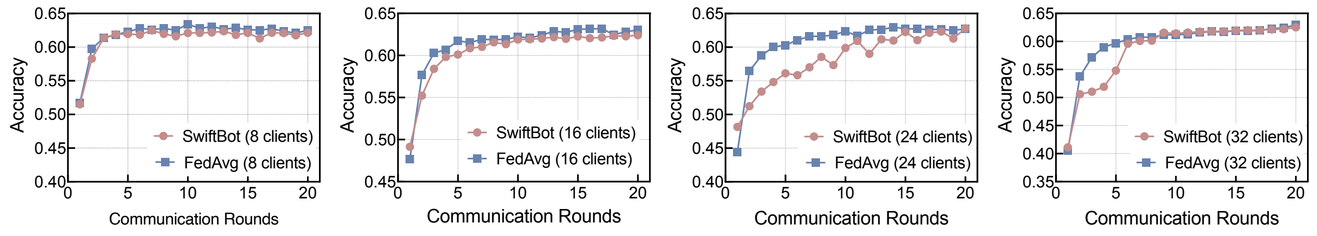


Fig. 3: Comparison of training accuracy under different client configurations in UCF101 dataset.

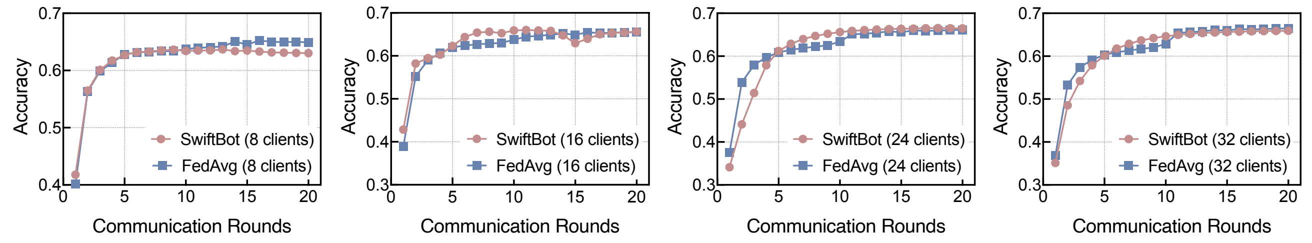


Fig. 4: Comparison of training accuracy under different client configurations in LibriSpeech dataset.

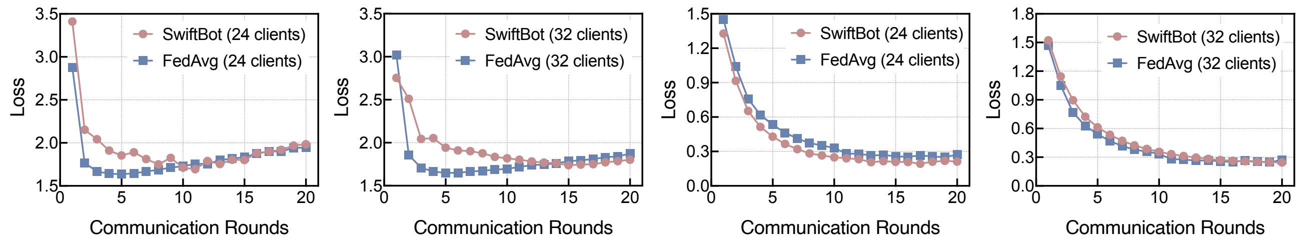


Fig. 5: Comparison of training loss in FedAvg vs. SwiftBot.

high training stability, achieving accuracy and loss curves comparable to the local baseline. Although the decentralized nature of SwiftBot introduces marginal variance during the initial communication rounds, it converges to the same final accuracy without degradation in model quality.

Figure 6 presents the training time distribution per round for the UCF101 and Libri Speech dataset across different client configurations. For UCF 101 dataset, the FedAvg baseline exhibits consistently high latency, remaining between 500s and 700s. This variance suggests that the system is frequently bottlenecked by straggler nodes or heterogeneous device capabilities, which delay the global aggregation process. In contrast, SwiftBot demonstrates a distinct inverse scaling capability. At 16 clients for UCF 101, it already provides a performance gain with a median latency of approximately 230s. Crucially, as the number of clients increases, the latency drops dramatically and reaching an ultra-low median of $\sim$30s at 32 clients. Similarily, for LibriSpeech dataset, when $N = 32$, SwiftBot reduces the median training time to $\sim$390s, a nearly $3\times$ speedup compared to the baseline. This confirms that SwiftBot's DHT-based resource pooling remains effective even for computationally intensive tasks. These results validate that SwiftBot successfully utilizes the additional nodes as a resource pool rather than a synchronization burden. By dynamically offloading tasks to idle peers via the DHT-overlay, the system effectively parallelizes the workload, eliminating the bottlenecks that cause the high latency and variance seen in the local approach.

Conversely, SwiftBot (pink distributions) maintains its characteristic inverse scaling behavior, effectively countering the heavier workload. While the performance gain is modest at $N = 8$ ($\sim$740s vs. $\sim$800s), the advantage grows exponentially with cluster size.

C. Container Selection and Migration Performance

To validate the effectiveness of our container selection and allocation algorithm, we conduct experimental evaluations comparing three approaches: our proposed DHT-based algorithm with distributed warm container migration, a local-only warm pool baseline that restricts container reuse to individual nodes, and a cold-start baseline that launches fresh containers for every task without any reuse. Figure 8a presents the cumulative distribution of task startup latency. The proposed algo-

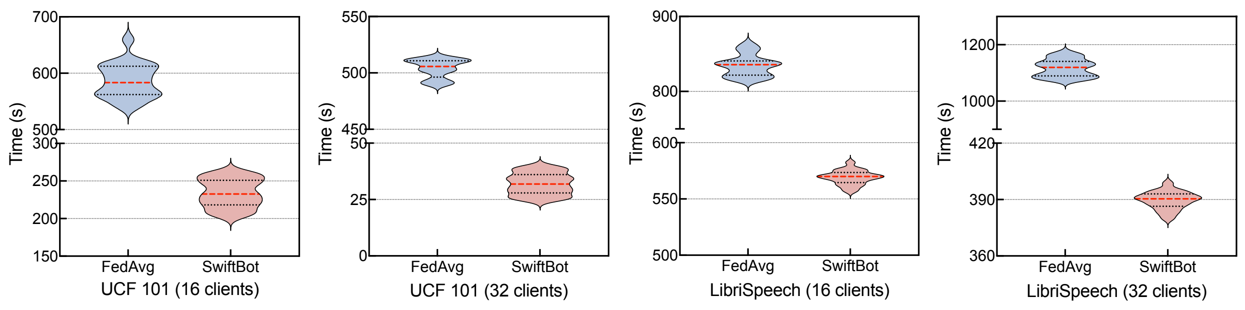


Fig. 6: The training time distribution per round in SwiftBot vs. FedAvg.

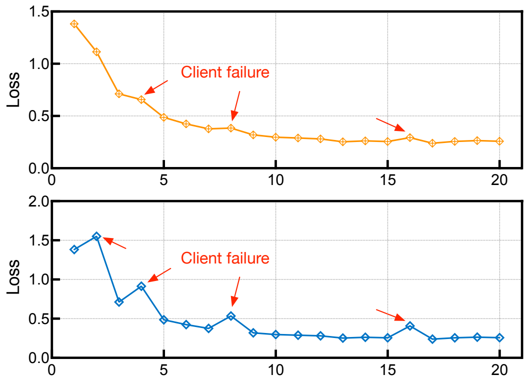
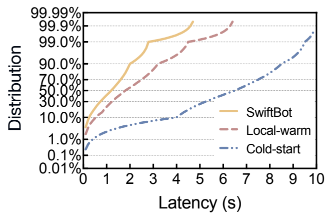
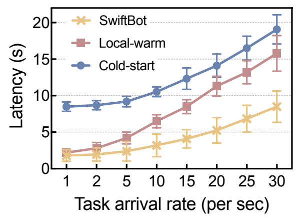


Fig. 7: The performance of training loss during clients' failures.



(a) Task Startup Latency Distribution



(b) P99 Latency vs. Task Arrival Rate

Fig. 8: Distribution of task startup latency and the tail latency as a function of task arrival rate.

rithm (blue curve) demonstrates substantial latency reduction compared to both baselines. At the median (P50), SwiftBot achieves approximately 120ms startup time, representing a 1.5× improvement over the local-only approach (180ms) and a 5.4× improvement over cold-start (650ms). The SwiftBot's curve demonstrates that DHT-based container discovery and migration successfully avoids cold starts for a large fraction of tasks. Even when migration is required, it ensures that only beneficial migrations are performed, preventing unnecessary overhead.

Figure 8b evaluates how the three approaches scale under increasing load, measured by task arrival rate (tasks per second). The y-axis shows P99 latency, a critical metric for interactive workloads where tail latency directly impacts user experience. At low loads (1-5 tasks/s), SwiftBot maintains 180-240ms compared to 220-420ms for local-only and 850-920ms for cold-start. As load increases beyond 10 tasks/s, the performance gap widens significantly. At 20 tasks/s, SwiftBot achieves 520ms, around 2.1× better than local-only (1100ms) and 2.7× better than cold-start (1400ms). This improvement comes from the algorithm's ability to leverage warm containers distributed across the DHT overlay. When local warm pools become slow under high load, the DHT enables discovery and migration of idle warm containers from less-loaded nodes, effectively load-balancing container resources across the cluster.

V. Background and Related Work

A. Multi-Robot Collaboration

Multi-robot collaboration entails the coordinated interaction of multiple autonomous agents to achieve objectives exceeding the capabilities of a single robot [12]. Modern fleets predominantly utilize the Robot Operating System 2 (ROS 2) for middleware support. While ROS 2 utilizes the Data Distribution Service (DDS) for decentralized discovery, Jo and Kwon [17] highlight that maintaining synchronization in dynamic, bandwidth-constrained environments remains a significant bottleneck for large-scale systems.

To address these communication and scalability challenges in heterogeneous fleets, recent work has explored role-based cooperation. Xue et al. [18] proposed an adaptive multi-robot cooperative localization method based on distributed consensus learning, where a subset of robots with high-quality sensors serves as "leaders" anchoring the state estimation of "follower" robots.

Similarly, Mayya et al. [19] developed a resilient task allocation strategy that dynamically reassigns tasks in heterogeneous teams when disruptions occur, improving adaptability during missions. In communication-denied environments, such as subterranean voids, specialized "relay" robots are deployed to repair network topology [20], [21]. In summary, the field is moving toward more resilient and efficient multi-robot coordination mechanisms [12], [22]. Recent analyses highlight a shift from centralized architectures to distributed consensus approaches and the integration of AI techniques for robust team performance [22].

B. Human-in-the-Loop Collaboration

The Human-in-the-Loop (HITL) paradigm involves human operators as an integral part of the control loop, allowing them to intervene or guide robot actions to ensure safe and effective operation [23]. This approach ranges from direct teleoperation to high-level supervisory control, so that human insight can correct or refine autonomous behaviors when needed. Mandi et al. [24] introduced *RoCo*, a framework where robots use LLMs to negotiate strategies dialectically with humans, significantly enhancing flexibility over rigid command parsers. Recent works like LLM-BRAIn [25] and LLM-MCoX [26] leverage large language models (LLMs) to translate natural language into executable behavior trees for exploration and search tasks.

Despite these capabilities, deploying LLMs on edge robots introduces safety risks regarding "hallucinations" (infeasible plans). To mitigate these risks, the Safety Aware Task Planning via Large Language Models in Robotics (SAFER) framework [27] introduces a dedicated safety agent and an LLM-as-a-judge module that audit candidate task plans and enforce control-barrier-based constraints before execution.

Furthermore, the computational cost of running these models on edge devices (e.g., Jetson Nano) creates prohibitive latency [28], necessitating the efficient offloading and warm-start mechanisms proposed in this work. For instance, Liu et al. [29] applied LLMs for coordinating heterogeneous robot teams, and Singh et al. [30] developed a multi-agent LLM approach for zero-shot robotic manipulation. However, these AI-driven frameworks still face challenges like hallucinated plans and heavy computation requirements, which hinder their reliability in safety-critical applications [31].

C. Federated Learning and Edge Intelligence

[HX: Classical FL is not very proper here, try to find federated robots computing things [maybe hard], we can keep this part and add more contents, I then delete some] Federated Learning (FL) enables collaborative model training while preserving privacy by sharing gradients rather than raw sensor data. Recently, FL techniques have been applied to robotic systems to enable distributed learning across networked agents. For instance, Gutierrez et al. [32] developed a ROS 2-based FL framework allowing multiple robots to jointly train models in both simulation and real-world deployments. Yu et al. [33] proposed a lifelong federated learning approach with continuous sim-to-real transfer for autonomous mobile robots, enabling fleets to continually improve a vision-based obstacle-avoidance model across simulated and real environments while keeping data local to each robot. Seyedmohammadi et al. [34] proposed *MoFLeuR* to handle non-IID motion data in HRI without centralizing datasets, while Chen et al. [35] introduced *FedPoD* to optimize lightweight prompt learning on edge devices.

However, node failures in dynamic edge networks can disrupt convergence; Delliquadri et al. [36] and Barroso et al. [37] emphasize the need for rapid failure recovery and live container migration. While Peña Queralta et al. [38] review blockchain and other distributed ledger technologies for decentralized multi-robot systems and highlight scalability and latency challenges, these platforms still face limitations that make their direct use for strict real-time control nontrivial. SwiftBot addresses this by replacing heavy consensus protocols with a lightweight Distributed Hash Table (DHT) overlay to achieve the millisecond-level responsiveness required for industrial robotics.

VI. Discussion

VII. Conclusion

In this paper, we presented SwiftBot, a distributed robotic task execution system that integrates LLM-based task decomposition with intelligent container orchestration over a DHT overlay network. SwiftBot addresses critical limitations of existing approaches—brittle hand-coded planners, centralized coordination bottlenecks, and inefficient resource allocation—through three innovations: semantic task decomposition achieving 94.3% accuracy without domain training, distributed warm container management reducing startup latency to 120ms median (5.4× improvement), and DHT-based decentralized scheduling maintaining 1.6-2.1× better performance under high load. Evaluation on 1,200 multimedia tasks validates that co-designing semantic understanding with resource management enables both flexibility and efficiency for real-time robotic control. Future work includes supporting heterogeneous robot teams, incorporating reinforcement learning for container pre-warming optimization, and enabling privacy-preserving edge-based LLM deployment.

VIII. evaluation part 2

References

[1] H. Guo, F. Wu, Y. Qin, R. Li, K. Li, and K. Li, "Recent trends in task and motion planning for robotics: A survey," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–36, 2023.

[2] R. Bernardo, J. M. Sousa, and P. J. Gonçalves, "Planning robotic agent actions using semantic knowledge for a home environment," *Intelligence & Robotics*, vol. 1, no. 2, pp. 116–130, 2021.

[3] W. Mao, W. Zhong, Z. Jiang, D. Fang, Z. Zhang, Z. Lan, H. Li, F. Jia, T. Wang, H. Fan *et al.*, "Robomatrix: A skill-centric hierarchical framework for scalable robot task planning and execution in open-world," *arXiv preprint arXiv:2412.00171*, 2024.

[4] Z. Luan, Y. Lai, R. Huang, S. Bai, Y. Zhang, H. Zhang, and Q. Wang, "Enhancing robot task planning and execution through multi-layer large language models," *Sensors*, vol. 24, no. 5, p. 1687, 2024.

[5] Z. Yan, N. Jouandeau, and A. A. Cherif, "A survey and analysis of multi-robot coordination," *International Journal of Advanced Robotic Systems*, vol. 10, no. 12, 2013.

[6] S. Liu, Y. Niu, and B. Wu, "Introduction to multi-robot coordination algorithms," in *2022 IEEE 4th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*. IEEE, 2022, pp. 832–837.

[7] D. P. Mtowe, L. Long, and D. M. Kim, "Low-latency edge-enabled digital twin system for multi-robot collision avoidance and remote control," *Sensors*, vol. 25, no. 15, p. 4666, 2025.

[8] A. E. Celik, I. Rodriguez, R. G. Ayestaran, and S. C. Yavuz, "Decentralized system synchronization among collaborative robots via 5g technology," *Sensors (Basel, Switzerland)*, vol. 24, no. 16, p. 5382, 2024.

[9] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6g: A comprehensive survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 334–366, 2021.

[10] P. A. M. Devan, F. A. Hussin, R. Ibrahim, K. Bingi, and F. A. Khanday, "A survey on the application of wirelesshart for industrial process monitoring and control," *Sensors*, vol. 21, no. 15, p. 4951, 2021.

[11] S. B. Kamtam, Q. Lu, F. Bouali, O. C. L. Haas, and S. Birrell, "Network latency in teleoperation of connected and autonomous vehicles: A review of trends, challenges, and mitigation strategies," *Sensors*, vol. 24, no. 12, p. 3957, 2024.

[12] A. Prorok, M. Malencia, L. Carlone, G. S. Sukhatme, B. M. Sadler, and V. Kumar, "Beyond robustness: A taxonomy of approaches towards resilient multi-robot systems," *arXiv preprint arXiv:2109.12343*, 2021.

[13] S. Choudhury, J. K. Gupta, M. J. Kochenderfer, D. Sadigh, and J. Bohg, "Dynamic multi-robot task allocation under uncertainty and temporal constraints," *Autonomous Robots*, vol. 46, no. 1, pp. 231–247, 2022.

[14] M. Groshev, G. Baldoni, L. Cominardi, A. de la Oliva, and R. Gazda, "Edge robotics: Are we ready? an experimental evaluation of current vision and future directions," *Digital Communications and Networks*, vol. 9, no. 1, pp. 166–174, 2023.

[15] M. Saboia, L. Clark, V. Thangavelu, J. A. Edlund, K. Otsu, G. J. Correa, V. S. Varadharajan, A. Santamaria-Navarro, T. Touma, A. Bouman, H. Melikyan, T. Pailevanian, S.-K. Kim, A. Archanian, T. S. Vaquero, G. Beltrame, N. Napp, G. Pessin, and A.-a. Agha-mohammadi, "Achord: Communication-aware multi-robot coordination with intermittent connectivity," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 184–10 191, 2022.

[16] J. Gielis, A. Shankar, and A. Prorok, "A critical review of communications in multi-robot systems," *Current Robotics Reports*, vol. 3, no. 3, pp. 213–225, 2022.

[17] D. Jo and Y. Kwon, "Generation of critical information and sharing mechanism for autonomous multi-robot collaboration," *IEEE Access*, vol. 13, p. 146856, 2025.

[18] C. Xue, H. Zhang, F. Zhu, Y. Huang, and Y. Zhang, "Adaptive multi-robot cooperative localization based on distributed consensus learning of unknown process noise uncertainty," *IEEE Transactions on Automation Science and Engineering*, 2024.

[19] S. Mayya, D. S. D'Antonio, D. Saldana, and V. Kumar, "Resilient task allocation in heterogeneous multi-robot systems," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1327–1334, 2021.

[20] L. Robinson and D. De Martini, "Robot-relay: building-wide, calibration-less visual servoing with learned sensor handover networks," *Autonomous Robots*, vol. 50, no. 3, 2025.

[21] S. Zhang and Others, "Bilayer real time multi-robot communication maintenance deployment framework for robot swarms," in *Journal of Physics: Conference Series*, vol. 2850, 2024, p. 012011.

[22] S. Zhang, Z. Li, Y. Yin, S. Xu, V. Chaudhary, and H. Xu, "A survey on multi-robot collaboration systems: Architectures, performances, and applications," *TechRxiv preprint*, 2025, doi:10.36227/techrxiv.176045766.60277537.

[23] Z. He, Y. Cao, and M. Ciocarlie, "Uncertainty comes for free: Human-in-the-loop policies with diffusion models," *arXiv preprint arXiv:2503.01876*, 2025.

[24] Z. Mandi, S. Jain, and S. Song, "Roco: Dialectic multi-robot collaboration with large language models," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 286–299.

[25] A. Lykov and D. Tsetserukou, "LLM-BRAIn: Ai-driven fast generation of robot behaviour tree based on large language model," *arXiv preprint arXiv:2305.19352*, 2023.

[26] R. Wang, H.-L. Hsu, D. Hunt, S. Luo, J. Kim, and M. Pajic, "Llmcox: Large language model-based multi-robot coordinated exploration and search," *arXiv preprint arXiv:2509.26324*, 2025.

[27] A. A. Khan, M. Andrev, M. A. Murtaza, S. Aguilera, R. Zhang, J. Ding, S. Hutchinson, and A. Anwar, "Safety aware task planning via large language models in robotics," *arXiv preprint arXiv:2503.15707*, 2025.

[28] C. Zhang, J. Chen, J. Li, Y. Peng, and Z. Mao, "Large language models for human–robot interaction: A review," *Biomimetic Intelligence and Robotics*, vol. 3, no. 4, p. 100131, 2023.

[29] K. Liu, Z. Tang, D. Wang, Z. Wang, X. Li, and B. Zhao, "Coherent: Collaboration of heterogeneous multi-robot system with large language models," *arXiv preprint arXiv:2409.15146*, 2025, accepted to ICRA 2025.

[30] H. Singh, R. J. Das, M. Han, P. Nakov, and I. Laptev, "Malmm: Multi-agent large language models for zero-shot robotics manipulation," *arXiv preprint arXiv:2411.17636*, 2024.

[31] N. Ranasinghe, W. M. Mohammed, K. Stefanidis, and J. L. Martinez Lastra, "Large language models in human-robot collaboration with cognitive validation against context-induced hallucinations," *IEEE Access*, 2025.

[32] G. M. Gutierrez, J. A. Rincon, and V. Julian, "Federated learning for collaborative robotics: A ros 2-based approach," *Electronics*, vol. 14, no. 7, p. 1323, 2025.

[33] X. Yu, J. Peña Queralta, and T. Westerlund, "Towards lifelong federated learning in autonomous mobile robots with continuous sim-to-real transfer," *Procedia Computer Science*, vol. 210, pp. 86–93, 2022.

[34] S. J. Seyedmohammadi, S. M. Sheikholeslami, J. Abouei, A. Mohammadi, and K. N. Plataniotis, "Mofleur: Motion-based federated learning gesture recognition," in *2024 IEEE 4th International Conference on Human-Machine Systems (ICHMS)*. IEEE, 2024, pp. 1–6.

[35] S. Chen, G. Long, T. Shen, J. Jiang, and C. Zhang, "Federated prompt learning for weather foundation models on devices," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24)*, 2024, p. 5772.

[36] B. Delliquadri, C. Wang, S. Chen, Z. Li, H. Luo, and H. Xu, "Fast meta failure recovery for federated meta-learning," in *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 2023, pp. 3154–3158.

[37] J. P. R. Barroso, A. Manacero, R. S. Lobato, and R. Spolon, "A comprehensive performance evaluation of container migration strategies," *Computing*, vol. 107, no. 2, 2025.

[38] J. Peña Queralta, F. Keramat, S. Salimi, L. Fu, X. Yu, and T. Westerlund, "Blockchain and emerging distributed ledger technologies for decentralized multi-robot systems," *Current Robotics Reports*, vol. 4, pp. 43–54, 2023.