# A Decentralized Framework for Resource-Constrained Task Redistribution*

**Doron Reid**[1], **Anaiya Reliford**[1], **Anietie Andy**[1], **Sonya T. Smith**[1], **Marcus Alfred**[1], **Sean Phillips**[2]
[1]Research Institute for Tactical Autonomy
Washington, DC 20009
[2]Air Force Research Laboratory
Space Vehicles Directorate
Albuquerque, NM 87117

## Abstract

Earth-observing satellite constellations are increasingly expected to operate autonomously in dynamic, resource-constrained, and failure-prone environments. As constellation size and heterogeneity grow, centralized and static task scheduling paradigms struggle to provide the adaptability required to maintain mission continuity under disruptions, intermittent connectivity, and limited onboard storage. This paper presents a decentralized task rescheduling framework inspired by a service-industry scheduling analogy that models satellites as resource constrained agents operating under memory restrictions, spatial access limitations, and intermittent offloading opportunities. A multi-criteria redistribution algorithm prioritizes memory availability, proximity, capability, and workload to reassign tasks following agent loss. Through large-scale simulation, we demonstrate that the proposed framework preserves high-priority task throughput and fair workload distribution despite irreversible capacity loss. The resulting system provides a physically motivated abstraction for studying resilient, decentralized scheduling and establishes a bridge between intuitive heuristic policies and future learning-based autonomy for heterogeneous low Earth orbit satellite constellations.

## Problem Statement

Large-scale multi-agent systems operating under resource, memory, and communication constraints must continuously allocate and reallocate tasks in dynamic and failure-prone environments. Existing decentralized scheduling approaches typically assume static agent capacity, unlimited internal state, or continuous access to coordination infrastructure, limiting their applicability to realistic systems where agents experience storage saturation, intermittent offloading opportunities, and permanent loss. The problem addressed in this work is how to design a decentralized task rescheduling framework that enables heterogeneous agents with finite memory, spatially constrained processing windows, and intermittent relief to maintain high-priority task throughput and fair workload distribution following agent failures.

*Approved for public release; distribution is unlimited. Public Affairs approval AFRL-2026-0782


## Introduction

Large-scale multi-agent systems increasingly operate in dynamic, resource-constrained, and failure-prone environments, where agents must continuously allocate and reallocate tasks while maintaining system-level performance. Satellite constellations provide a concrete motivating domain for this problem. Earth-observing spacecraft in Low Earth Orbit (LEO) must execute time-sensitive sensing tasks while operating with limited onboard computation, finite data storage, constrained power budgets, and only intermittent access to ground stations for data offload. These constraints directly couple task execution decisions with internal resource state, making scheduling a tightly constrained, real-time decision problem rather than a purely combinatorial one.

In contested and congested space environments, satellite loss due to hardware failure, environmental hazards, or adversarial action is no longer a rare edge case. When a satellite becomes non-operational, its in-progress and queued observation tasks become orphaned and must be redistributed among remaining spacecraft whose own resources are already bounded. If redistribution is poorly managed, high-priority observations may be delayed or dropped, onboard buffers may saturate, and system performance may degrade irreversibly. Designing decentralized task rescheduling mechanisms that remain effective under irreversible capacity loss, heterogeneous agent capabilities, and intermittent offloading opportunities is therefore a fundamental autonomy challenge.

This challenge is further compounded in heterogeneous constellations, where satellites differ in sensor modalities, sensing quality, computational resources, and preferred task types. Optimal execution requires aligning task objectives with satellite-specific capabilities while simultaneously managing energy reserves, reaction wheel momentum, and finite data storage. Coordinating these interdependent decisions during periods of disruption is operationally intensive without onboard autonomy. Consequently, the transition from traditional ground-centric operations to autonomous, onboard decision-making represents a critical shift in the architecture of next-generation Earth observation missions.

In this work, we investigate an autonomous framework

that enables heterogeneous satellites to dynamically schedule, reassign, and execute Earth-observation tasks in real-time, utilized a foundational framework of our rule-based baseline, the Waiter-to-Restaurant Analogy. This paper extends the framework to capture additional operational realities of satellite constellations. Each service agent operates within spatially constrained regions using priority-aware, capability-informed queues. Task complexity consumes variable memory, forcing agents to reason about overload rather than simply task count. A dual-task processing model enables parallel progress on primary and lightweight tasks, reflecting mixed-priority workloads (Reliford et al. 2026).

We introduce a manager-assisted relief mechanism in which agents occasionally receive partial memory relief when passing through specific regions. This mechanism is directly motivated by satellite downlink operations as spacecraft can only offload data when in view of ground stations rather than continuously. Together, these extensions enable systematic study of how decentralized, memory-bounded agents redistribute work after permanent capacity loss, and how scheduling structure influences high-priority task preservation, overload avoidance, and system-level resilience.

## Literature Review

Task scheduling and load balancing is widely studied across many domains such as cloud computing, manufacturing systems, satellites, and distributed environments. Early methods like First-Come-First-Serve (FCFS), Shortest Job First (SJF), Min–Min, and Max–Min aimed to improve job completion time but struggled in heterogeneous systems due to their poor processing speeds (Devi et al. 2024). Later approaches would incorporate heterogeneity-aware heuristics such as Heterogeneous Earliest Finish Time (HEFT) and Dynamic Heterogeneous Shortest Job First (DHSJF), as well as metaheuristic techniques like Genetic Algorithms to optimize performance metrics (Devi et al. 2024; Rekha and Dakshayini 2019). These methods improved workload distribution and overall system utilization, but they generally assume static task sets and do not explicitly address persistent capacity loss caused by agent removal.

Many tasking problems are Nondeterministic Polynomial (NP) time hard, and researchers have developed exact optimization methods to address this. Mixed Integer Linear Programming (MILP) and Integer Programming (IP) formulations were noted to be utilized in mobile robot job shop scheduling (JSSP) (Rema et al. 2025) and the health care sector (van Kessel, Freeman, and Santos 2023). These approaches decompose large problems into smaller subproblems to reduce computational complexity. These methods can produce high-quality solutions, but rely on centralized solvers and can become computationally expensive in dynamic, large-scale systems.

The majority of research has been focused on scheduling models, with little attention to schedule adjustments (Nof and Grant 1991). Rescheduling under disruptions has received particular attention in airline maintenance, healthcare operations, satellite observations, and construction planning. In the sector of airline maintenance, task schedules must continuously adapt to stochastic task arrivals and changing resource availability (van Kessel, Freeman, and Santos 2023). A study done by (Lagos, Delgado, and Klapp 2020), developed a maintenance task scheduling algorithm designed to prevent tasks from becoming overdue. Scheduling is performed by formulating the problem as a Markov Decision Process and applies Integer Programming and Approximate Dynamic Programming techniques to minimize the expected costs of expired maintenance tasks (Lagos, Delgado, and Klapp 2020). Their approach distinguishes between critical and non-critical tasks that may obtain an extension. However, the model does incorporate factors of resource availability or the revising of schedules in response to disruptions (van Kessel, Freeman, and Santos 2023). Most disruption management frameworks focus on restoring feasibility or minimizing schedule changes under temporary disturbances. Few studies examine the long term removal of an agent or analyze how redistribution affects long term system degradation in decentralized settings.

Collectively, these prior works provide the algorithmic and theoretical foundations for scheduling under constraints, heterogeneity, and disruption. (Li et al. 2024; van Kessel, Freeman, and Santos 2023) establishes the importance of rescheduling in dynamic environments but largely assumes centralized control and temporary disturbances. In contrast, our work extends this literature by modeling permanent agent removal and decentralized task redistribution under memory and spatial constraints. We will analyze specifically degradation behavior, backlog formation, and fairness under sustained capacity loss. In this way, our framework will build upon existing scheduling and rescheduling research while addressing a gap in resilient, decentralized multi-agent systems.

## Mapping the Waiter Analogy to Satellite Constellations

### Conceptual Mapping

The waiter-based scheduling framework is designed as an abstraction of decentralized satellite operations. Each element of the restaurant environment corresponds to a concrete component of a space system:

- **Waiters → Satellites**: Each waiter represents an individual spacecraft capable of executing sensing tasks subject to onboard resource constraints.
- **Tasks → Earth Observation Requests**: Tasks correspond to observation opportunities (e.g., imaging, atmospheric retrievals, environmental measurements) with assigned priority and complexity.
- **Sections → Orbital Regions**: Restaurant sections represent discrete spatial regions along an agent's trajectory. Section 6 corresponds to the primary area of interest where tasks originate, analogous to geographic regions such as CONUS where observation opportunities are concentrated.
- **Waiter Capability Vector → Satellite Sensor Capability Vector**: Speed, efficiency, accuracy, and multitasking

map to relative proficiency of a satellite's sensor suite across multiple task types.

- **Waiter Memory Capacity → Onboard Data Buffer / Computational Capacity**: The finite memory space assigned to each waiter represents a satellite's limited onboard storage and processing capacity.

### Task Memory and Onboard Storage

In the analogy, each task consumes memory proportional to its task complexity. A waiter may only accept new tasks if sufficient memory is available. This directly maps to satellites accumulating data products in onboard buffers. High-complexity tasks represent data-intensive observations, while simple tasks represent lightweight products. When memory is exhausted, new tasks cannot be accepted, mimicking buffer saturation that prevents additional data collection.

### Managers as Ground Stations

Managers provide intermittent memory relief to waiters when they enter specific sections. This mechanism emulates ground-station downlink:

- Managers → Ground Stations
- Entering Manager Section → Satellite in Ground-Station Visibility Window
- Memory Relief → Partial Data Downlink

Just as satellites cannot continuously transmit data to Earth, waiters do not receive constant assistance. Relief is probabilistic and limited in magnitude, reflecting variable contact durations, link quality, and downlink rates. The constraint that at most two managers may assist a waiter per minute mirrors practical limitations on simultaneous ground contacts or downlink throughput. This abstraction ensures that storage pressure accumulates over time and is only alleviated during discrete access windows, capturing a key coupling between task execution and communication geometry.

### Firing Event as Satellite Loss

Firing a waiter permanently represents catastrophic satellite failure. All tasks held by that waiter are pooled and redistributed to remaining agents, analogous to orphaned tasks released when a satellite becomes non-operational. This firing event also represents permanent loss of onboard storage and sensing capabilities.

### Multi-Criteria Redistribution

The ranking tuple used during redistribution corresponds to satellite retasking heuristics:

- Distance to Section 6 → Orbital Proximity to Task Location
- Capability Score → Sensor–Task Compatibility
- Memory Pressure → Buffer Fill Ratio
- Queue Length → Current Workload

This hierarchy prioritizes avoiding overload before optimizing efficiency, ensuring that tasks are not assigned to satellites already near saturation.

## System Architecture

Building upon the foundational restaurant model from our previous work, this paper extends the system with significant operational constraints that more accurately reflect cognitive demands on service staff. This system now models a 24-hour restaurant shift (1440 minutes) with five waiters operating across seven spatial sections. A waiter's distance is defined on a circular topology. Distance decreases as a waiter moves closer to Section 6, and the topology wraps around Section 7 through Section 1. There are 100 tasks which all originate exclusively in Section 6, which imposes both spatial and temporal constraints on the workers. Waiters are characterized by a capability vector consisting of speed, efficiency, accuracy, and multitasking ability. These attributes are taken and a scalar capability score is computed as:

$$C = \text{speed} + \text{efficiency} + 0.5\,\text{accuracy} + 0.5\,\text{multitasking} \tag{1}$$

Speed and efficiency are weighted most strongly because they more strongly influence throughput under time constraints, and accuracy and multitasking contribute at half weight. This weighted model reflects the operational reality that rapid coverage and throughput are more critical to performance than precision in routine routing tasks. However, the critical innovation introduced now is cognitive load management through memory constraints, task complexity variation, and managed section engagement.

### Memory Constraints

We introduced a memory capacity model that reflects the weight of managing simultaneous tasks rather than approaching task assignment as purely capacity based. Each waiter has a fixed memory space of M = 500 memory units, which represents their ability to account for and manage future tasks. Each task consumes memory according to its complexity tier:

$$\text{Task Memory Cost} = \begin{cases} \mathcal{U}(5,9) & \text{if simple} \\ \mathcal{U}(10,19) & \text{if standard} \\ \mathcal{U}(20,34) & \text{if complex} \\ \mathcal{U}(35,50) & \text{if critical} \end{cases} \tag{2}$$

Where U (a,b) denotes uniform random distribution between a and b. A waiter can only accept a task if the memory a waiter used ($M_{used}$) plus the given task cost is less than or equal to the fixed memory space of 500 ($M_{capacity}$).

### Section Constraints

Waiters can only perform tasks within section 6 under strict temporal constraints:

- **Visit Duration:** Maximum 15 minutes per visit
- **Cooldown Period:** 90-minute mandatory break before re-entering Section
- **Circular Movement:** Waiters move clockwise through sections $1 \to 7 \to 1$

- **Distance Function:**

$$d = \begin{cases} 6 - s, & s \leq 6 \\ (7 - s) + 6, & \text{otherwise} \end{cases} \tag{3}$$

These constraints create realistic scheduling pressures meaning a waiter far from Section six may not reach it before their cooldown resets, effectively reducing their productivity window.

## Manager System

To avoid cognitive overload, a manager system is introduced to provide relief for waiters. Three managers are stationed in sections two, four, and six. When a waiter enters a manager's section, the manager may provide cognitive assistance with probability $p_{\text{help}} = 0.02$, freeing between 2%–10% of the waiter's used memory, provided the waiter has previously completed at least one task. To prevent unrealistic dependency, a maximum of two managers may assist any single waiter within the same minute.

This design is motivated by satellite ground station operations, where satellites do not continuously downlink data but instead transmit information only when passing over designated ground stations. Similarly, waiters receive cognitive assistance only when they are within range of a manager, ensuring intermittent support rather than constant dependency across the system.

## Dual-Task Processing

Unlike our previous single-task-focused model approach, the algorithm has been extended to allow parallel task engagement in section 6, provided the waiter is capable of handling multiple tasks. The algorithm is priority driven, so while maintaining focus on the current high priority task at hand, waiters simultaneously attempt to complete quick medium/low-priority tasks. This feasible using two task progress models.

**Primary Task Progress:**

$$\Delta p_{\text{main}} = U(5, 15) \times \frac{C}{20} \quad \text{percent per minute} \tag{4}$$

**Quick Task Progress:**

$$\Delta p_{\text{quick}} = U(10, 25) \times \frac{C}{20} \quad \text{percent per minute} \tag{5}$$

While working on a primary task, a waiter makes progress over time and simultaneously attempts a lower-priority task. The progress increment is composed of two components:

- **Base progress:** A random draw from a uniform distribution, capturing natural variation in task difficulty and waiter focus.
- **Capability scaling:** The waiter's capability score $C$, normalized by the maximum value of 20, which rewards more capable waiters with faster progress.

Medium and low priority tasks progress faster due to their simplicity, low memory cost, and requirement of less cognitive load. On average, quick tasks progress 1.5-2.5x faster which don't interfere with the waiter's main focus in realistic service scenarios.

Among all medium and low priority tasks within a waiters queue, the lowest memory cost is selected:

$$\begin{gathered} t_{\text{quick}} = \arg \min_{t \in \text{task\_list}} \text{memory\_cost}(t) \\ \text{s.t. } t_{\text{priority}} \in \{\text{medium}, \text{low}\} \end{gathered} \tag{6}$$

The lower the memory cost, the less of a burden it is for that waiter to work on that task, maximizing the chance of completion within the 15-minute Section six visibility window. Attacking the cheapest task first, optimizes memory space for new assignments and clears easy items constantly.

# Dynamic Rescheduling Framework

At initialization, all 100 tasks are distributed to avoid bias toward any individual waiter. Tasks are categorized by priority (40% high, 30% medium, 30%, low) and assigned using a priority-aware round-robin procedure, ensuring early handling of highest-priority tasks. The assurance of high priority tasks handling, prevents the situation where lower priority tasks are blocking critical assignments.

During execution, the environment becomes dynamic as a random waiter gets fired from their shift. Firing is permitted only after 25% (300 minutes) of the simulation has elapsed, with a 5% probability per minute. The system is limited to one waiter firing per simulation. The firing event constraints are as follows:

The firing mechanism is governed by the following constraints:

- **Timing protection:** No firing event may occur before minute 360, ensuring sufficient system stabilization.
- **Permanent status:** Once fired, a waiter cannot be rehired and remains removed for the remainder of the simulation.
- **One-fire limit:** After a waiter is fired, no additional firings are permitted, preventing cascading disruptions.

## Firing Event

The permanent removal model of a waiter creates a partial-system failure scenario. The firing event forces the remaining four waiters to acquire work that was originally distributed across five. This evaluates whether the algorithm can maintain high-priority task completion, fairness under load, and memory-aware scheduling. Unlike our previous algorithm where rehiring restored memory capacity, this permanent removal creates an irreversible capacity loss, impacting key metrics:

$$C_{\text{avail}} = 4 \times 500 - \sum_{t \in \mathcal{T}_{\text{reassigned}}} M_{\text{cost}}(t) \tag{7}$$

At initialization, each waiter's memory utilization reflects a forecasted allocation derived from the priority-aware round-robin task assignment. Because tasks are distributed at the start of the simulation and each task consumes memory proportional to its complexity, waiters begin the mission with a pre-committed cognitive load. Thus, memory saturation following a firing event is not the result of arbitrary

accumulation, but of previously forecasted resource commitments. The redistribution event therefore forces the remaining waiters to absorb orphaned tasks on top of already allocated memory budgets, making capacity loss structurally constrained rather than opportunistic.

If a fired waiter had tasks totaling more than 250 memory units, the remaining waiters may not have enough capacity to take all pending tasks. This can create a backlog where work cannot proceed even though time is available. High-priority tasks must compete with the overhead of task redistribution. The four-criterion ranking helps redistribute tasks quickly ($O(n \log n)$ in the number of tasks), but the loss of capacity from the fired waiter still limits performance. The memory-pressure criterion helps direct tasks to less-burdened waiters, but the system cannot guarantee perfectly balanced workload increases.

### Task Redistribution Upon Firing

Due to this event, the rescheduling logic is implemented through a task redistribution mechanism. When a waiter is removed, all of their tasks (queued tasks, the current in-progress task, and any unassigned pending tasks) are pooled together and immediately redistributed among all active waiters. This unified pooling ensures that no work is lost and that global backlogging is addressed.

The pooled tasks are sorted by priority (high $\rightarrow$ medium $\rightarrow$ low), enforcing a high-priority-first policy so that critical work is assigned before lower-priority tasks and before highly capable waiters reach their memory limits. The algorithm then iterates through the sorted task list. For each task, it first checks whether a waiter can accept the task given their current memory usage. If not, the waiter is marked as ineligible; otherwise, the waiter is assigned a ranked tuple.

This ranking returns a four-tuple that defines a waiter's suitability for the redistributed task:

- **Memory pressure (highest priority):** $\frac{M_{\text{used}}}{M_{\text{capacity}}}$, ranging from 0 (empty) to 1 (full). Preferable waiters have sufficient available memory to avoid overload.
- **Distance to section 6:** Waiters closer to the task execution zone are preferred to reduce travel time.
- **Capability score:** Higher capability scores indicate greater suitability for handling complex tasks.
- **Queue length:** Among equally qualified candidates, waiters with fewer queued tasks are preferred.

This tuple-based ranking encodes a priority hierarchy that first avoids overload, then reduces spatial inefficiency, aligns task complexity with waiter capability, and finally balances workload across the system. After sorting waiters by this tuple, the top-ranked waiter is selected as the best candidate. This waiter is re-evaluated for available memory to handle edge cases, assigned the task, has their queue re-sorted to maintain priority ordering, and has their memory usage recomputed as $M_{\text{used}} = \sum$ task cost. If no waiter has memory capacity for a task, that task remains pending and must await memory relief (via manager assistance) or future completion of existing tasks.

This is a multi-criteria ranking that balances overload, efficiency, capability, and fairness. Prioritizing memory pressure initially prevents the common pitfall of overburdening the strongest waiter, while still respecting spatial and capability constraints.

## Simulation Overview

The Waiter to Restaurant task allocation policy was evaluated to assess the robustness of a Centralized Training, Decentralized Execution (CTDE) paradigm, where one waiter is fired permanently mid-mission.

At the beginning of the simulation, a queued list of tasks for the five waiters is displayed, including the memory state for each waiter and the amount of tasks they each have. The initial priority-aware task allocation and memory utilization are shown in Figure 1.

```
--- Initial Setup ---
Waiter 0 - Tasks: ['1(high)', '6(high)', '11(high)', '16(high)', '21(high)', '26(high)',
 '31(high)', '36(high)', '41(medium)', '46(medium)', '51(medium)', '56(medium)', '61(med
ium)', '66(medium)', '71(low)', '76(low)', '81(low)', '86(low)', '91(low)', '96(low)']
Waiter 1 - Tasks: ['2(high)', '7(high)', '12(high)', '17(high)', '22(high)', '27(high)',
 '32(high)', '37(high)', '42(medium)', '47(medium)', '52(medium)', '57(medium)', '62(med
ium)', '67(medium)', '72(low)', '77(low)', '82(low)', '87(low)', '92(low)', '97(low)']

Waiter 2 - Tasks: ['3(high)', '8(high)', '13(high)', '18(high)', '23(high)', '28(high)',
 '33(high)', '38(high)', '43(medium)', '48(medium)', '53(medium)', '58(medium)', '63(med
ium)', '68(medium)', '73(low)', '78(low)', '83(low)', '88(low)', '93(low)', '98(low)']
Waiter 3 - Tasks: ['4(high)', '9(high)', '14(high)', '19(high)', '24(high)', '29(high)',
 '34(high)', '39(high)', '44(medium)', '49(medium)', '54(medium)', '59(medium)', '64(med
ium)', '69(medium)', '74(low)', '79(low)', '84(low)', '89(low)', '94(low)', '99(low)']

Waiter 4 - Tasks: ['5(high)', '10(high)', '15(high)', '20(high)', '25(high)', '30(high)'
, '35(high)', '40(high)', '45(medium)', '50(medium)', '55(medium)', '60(medium)', '65(me
dium)', '70(medium)', '75(low)', '80(low)', '85(low)', '90(low)', '95(low)', '100(low)']
```

Figure 1: Initial priority-aware task allocation among the five waiters at the start of the simulation.

```
[MEMORY STATE AFTER INITIAL ASSIGNMENT]
Waiter 0: used=419/500, queued=20
Waiter 1: used=410/500, queued=20
Waiter 2: used=364/500, queued=20
Waiter 3: used=477/500, queued=20
Waiter 4: used=420/500, queued=20
```

Figure 2: Projected memory utilization for each waiter at simulation initialization.

Throughout the entire simulation, the printed event log illustrates how different waiters gradually complete tasks as time progresses. A representative segment of the execution log is shown in Figure 3. The event log first prints out waiters completing quick tasks, and receiving memory relief from managers in their respective sections.

```
--- Simulation Loop ---
[QUICK] Minute 30: Waiter 2 quickly completed Task 63 (medium)
[QUICK] Minute 43: Waiter 3 quickly completed Task 69 (medium)
[QUICK] Minute 49: Waiter 4 quickly completed Task 75 (low)
[COMPLETE] Minute 51: Waiter 2 finished Task 3 (high)
[QUICK] Minute 58: Waiter 2 quickly completed Task 58 (medium)
[QUICK] Minute 59: Waiter 1 quickly completed Task 57 (medium)
[COMPLETE] Minute 71: Waiter 3 finished Task 4 (high)
[COMPLETE] Minute 73: Waiter 1 finished Task 2 (high)
[COMPLETE] Minute 77: Waiter 4 finished Task 5 (high)
[QUICK] Minute 81: Waiter 0 quickly completed Task 71 (low)
[QUICK] Minute 93: Waiter 2 quickly completed Task 83 (low)
[QUICK] Minute 98: Waiter 4 quickly completed Task 90 (low)
[QUICK] Minute 99: Waiter 3 quickly completed Task 59 (medium)
```

Figure 3: Representative execution log showing waiters task completions and concurrent task progress.

```
[MANAGER] Manager 0 helped Waiter 1 in section 2. Freed 27 memory units (7.5%).
[MANAGER] Manager 1 helped Waiter 2 in section 4. Freed 27 memory units (8.0%).
[MANAGER] Manager 2 helped Waiter 1 in section 6. Freed 11 memory units (3.4%).
[MANAGER] Manager 0 helped Waiter 2 in section 2. Freed 11 memory units (3.7%).
[MANAGER] Manager 0 helped Waiter 2 in section 2. Freed 17 memory units (5.6%).
[COMPLETE] Minute 198: Waiter 2 finished Task 8 (high)
[MANAGER] Manager 2 helped Waiter 1 in section 6. Freed 15 memory units (4.2%).
[QUICK] Minute 206: Waiter 1 quickly completed Task 97 (low)
[MANAGER] Manager 2 helped Waiter 2 in section 6. Freed 24 memory units (7.2%).
```

Figure 4: Execution log illustrating intermittent manager-assisted memory relief.

In one simulation run, Waiter 4 is fired. This simulates a sudden failure or unavailability, triggering a redistribution event in which the remaining four waiters claim their tasks. Memory blocking events occur when a waiter cannot accept a task due to reaching their memory limit. An example of redistribution failure due to memory saturation is shown in Figure 3.

```
[EVENT] Fired Waiter 4.
[MEMORY BLOCK] Task 20 (high) cannot go to Waiter 3 (used 480/500)
[MEMORY BLOCK] Task 21 (high) cannot go to Waiter 2 (used 483/500)
[MEMORY BLOCK] Task 21 (high) cannot go to Waiter 3 (used 480/500)
[MEMORY BLOCK] Task 23 (high) cannot go to Waiter 0 (used 484/500)
[MEMORY BLOCK] Task 23 (high) cannot go to Waiter 2 (used 483/500)
[MEMORY BLOCK] Task 23 (high) cannot go to Waiter 3 (used 480/500)
[MEMORY BLOCK] Task 24 (high) cannot go to Waiter 0 (used 484/500)
[MEMORY BLOCK] Task 24 (high) cannot go to Waiter 1 (used 493/500)
[MEMORY BLOCK] Task 24 (high) cannot go to Waiter 2 (used 483/500)
[MEMORY BLOCK] Task 24 (high) cannot go to Waiter 3 (used 480/500)
[MEMORY BLOCK] Task 24 (high) cannot go to Waiter 0 (used 484/500)
[MEMORY BLOCK] Task 25 (high) cannot go to Waiter 0 (used 484/500)
```

Figure 5: Event log following the permanent removal of Waiter 4, showing tasks that could not be reassigned due to memory constraints.

The reschedule log displays the claimed tasks they acquired from waiter four's queue according to the redistribution policy (memory-capacity/pressure, proximity to Section six, capability, and load). The resulting post-redistribution allocation and updated memory states are illustrated in Figure 4. Waiter 4 is now offline, has released its tasks, and is permanently removed from the simulation.

```
[RESCHEDULE] Waiter 4 was fired. Task redistribution:
Waiter 0: WORKING: 6(high) | QUEUED: ['9(high)', '11(high)', '11(high)', '16(high)', '16
(high)', '21(high)', '21(high)', '26(high)', '31(high)', '33(high)', '36(high)', '41(med
ium)', '46(medium)', '51(medium)', '56(medium)', '61(medium)', '66(medium)', '76(low)',
'81(low)', '86(low)', '91(low)']
Waiter 1: WORKING: 12(high) | QUEUED: ['2(high)', '8(high)', '12(high)', '15(high)', '15
(high)', '17(high)', '17(high)', '22(high)', '22(high)', '23(high)', '27(high)', '32(hig
h)', '37(high)', '41(medium)', '42(medium)', '47(medium)', '52(medium)', '67(medium)', '
72(low)', '77(low)', '82(low)', '87(low)', '92(low)']
Waiter 2: WORKING: 18(high) | QUEUED: ['1(high)', '3(high)', '4(high)', '5(high)', '6(hi
gh)', '7(high)', '10(high)', '14(high)', '19(high)', '20(high)', '23(high)', '28(high)',
 '31(high)', '33(high)', '38(high)', '43(medium)', '53(medium)', '73(low)', '78(low)', '
93(low)', '98(low)']
Waiter 3: WORKING: 14(high) | QUEUED: ['13(high)', '18(high)', '19(high)', '20(high)', '
24(high)', '28(high)', '29(high)', '34(high)', '36(high)', '39(high)', '44(medium)', '57
(medium)', '64(medium)', '74(low)', '79(low)', '84(low)', '89(low)', '94(low)', '99(low)
']
Waiter 4: WORKING: None | QUEUED: []
```

Figure 6: Task redistribution among the remaining waiters after Waiter 4 is removed.

```
[MEMORY STATE AFTER FIRING]
Waiter 0: used=499/500, queued=21
Waiter 1: used=500/500, queued=23
Waiter 2: used=496/500, queued=21
Waiter 3: used=500/500, queued=19
Waiter 4: used=0/500, queued=0
```

Figure 7: Updated memory utilization of the remaining waiters following task redistribution.

```
--- Simulation Complete ---

Task Status:
Task 1: Priority high, Status: done, Current Waiter: 2
Task 2: Priority high, Status: done, Current Waiter: 1
Task 3: Priority high, Status: done, Current Waiter: 2
Task 4: Priority high, Status: done, Current Waiter: 2
Task 5: Priority high, Status: done, Current Waiter: 2
Task 6: Priority high, Status: done, Current Waiter: 2
Task 7: Priority high, Status: done, Current Waiter: 2
Task 8: Priority high, Status: done, Current Waiter: 1
Task 9: Priority high, Status: done, Current Waiter: 0
Task 10: Priority high, Status: done, Current Waiter: 2
Task 11: Priority high, Status: done, Current Waiter: 0
Task 12: Priority high, Status: done, Current Waiter: 1
Task 13: Priority high, Status: done, Current Waiter: 3
Task 14: Priority high, Status: done, Current Waiter: 2
Task 15: Priority high, Status: done, Current Waiter: 1
Task 16: Priority high, Status: done, Current Waiter: 0
Task 17: Priority high, Status: done, Current Waiter: 1
Task 18: Priority high, Status: done, Current Waiter: 3
Task 19: Priority high, Status: done, Current Waiter: 2
Task 20: Priority high, Status: done, Current Waiter: 3
Task 21: Priority high, Status: done, Current Waiter: 0
Task 22: Priority high, Status: done, Current Waiter: 1
```

Figure 8: Final simulation summary showing completed tasks and the waiter responsible for each completion.

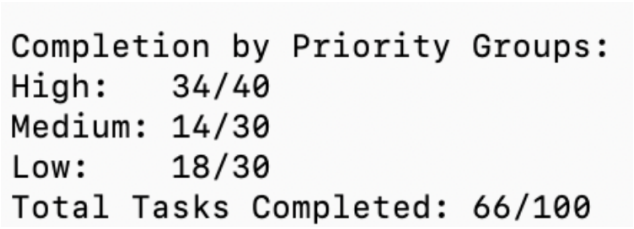


Figure 9: Task completion rates by priority level and overall completion rate across all tasks.

The final summary table provides a detailed view of each task's status, priority level, and the waiter responsible for its completion. The complete task-level summary is shown in Figure 5. 66/100 tasks were completed with High: 34/40, Medium: 14/30, Low: 18/30. Aggregate completion statistics by priority group are shown in Figure 6. High-priority completion rate is strong (85%), whereas medium and low priority task completion is lower. In prior simulations, medium-priority and low-priority tasks would remain pending or in progress while all high-priority tasks are completed. The dual-task processing policy preserves the baseline's strong high-priority completion performance while substantially increasing medium and low priority throughput. The summary is now a meaningful share of completions by the end of the simulation.

## Conclusion

This work extends a rescheduling benchmark for decentralized task allocation in heterogeneous LEO satellite constellations by shifting emphasis from policy optimization to the algorithmic structure governing task redistribution under permanent capacity loss. Using an interpretable service-inspired abstraction as a modeling lens, we formalize how memory-bounded agents with intermittent offloading opportunities and spatial constraints reschedule work following irreversible degradation. By introducing explicit cognitive (memory) capacity constraints, spatially constrained processing windows, intermittent manager-based relief, and dual-task processing, the benchmark captures degradation behaviors that are typically abstracted away in simplified scheduling studies.

The permanent removal of an agent exposed fundamental limits in capacity recovery, backlog absorption, and fairness. Results show that while high priority task completion can be largely preserved through memory aware and multi-criteria redistribution, medium and low priority throughput degrades predictably as cognitive and spatial constraints tighten. The inclusion of dual-task processing further demonstrated operational realities, significantly improving non-critical task throughput without compromising high priority performance.

Where prior work emphasized maximizing idealized completion rates under nominal conditions, this benchmark emphasizes realistic failure modes, recovery limits, and trade-offs that emerge under sustained disruption. Within a realistic simulation environment, the benchmark provides a foundation for comparing heuristic and learning-based approaches under stress conditions. This framework is positioned to be a critical step toward future learning-based approaches to resilient, autonomous constellation management.

## Future Work

While the current framework incorporates priority-aware redistribution and dynamic reassignment following permanent agent removal, several extensions will further increase operational realism and algorithmic rigor.

First, we will introduce explicit task deadlines to model time-sensitive sensing requirements to mimic real satellite operations. We will assign each task a deadline parameter based on its priority class and mission value. High-priority tasks will possess tighter temporal windows, while medium- and low-priority tasks will have progressively longer validity periods. This enables dynamic urgency escalation, allowing priority to evolve as deadlines approach rather than remaining a static label.

Second, although the current framework already includes high-priority-first redistribution and memory-aware ranking during failure events, proactive reassessment mechanisms can be formalized further. In the present implementation, orphaned and unassigned tasks are globally pooled, sorted by priority, and immediately redistributed using a multi-criteria suitability ranking. This ensures that high-priority tasks are never passively ignored and are always considered first during reassignment.

Finally, the waiter-based analogy will be transferred to the satellite domain to serve as the basis of comparison against reinforcement learning and auction-based policies. This analogy establishes a rule-based benchmark against which more advanced coordination strategies can be evaluated within the satellite simulation framework, enabling policy learning under realistic orbital, storage, and communication constraints.

## Acknowledgments

This material is based on work supported by the Department of the Air Force under the Air Force Contract No.FA9550-23-D-0001. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of the Air Force. Approved for public release; distribution is unlimited. Public Affairs approval AFRL-2026-0782.

## References

Devi, N.; Dalal, S.; Solanki, K.; et al. 2024. A Systematic Literature Review for Load Balancing and Task Scheduling Techniques in Cloud Computing. *Artificial Intelligence Review*, 57: 276.

Lagos, C.; Delgado, F.; and Klapp, M. A. 2020. Dynamic Optimization for Airline Maintenance Operations. *Transportation Science*, 54(4): 998–1015.

Li, H.; Li, Y.; Meng, Q. Q.; Li, X.; Shao, L.; and Zhao, S. 2024. An Onboard Periodic Rescheduling Algorithm for

Satellite Observation Scheduling Problem With Common Dynamic Tasks. *Advances in Space Research*, 73(10): 5242–5253.

Nof, S. Y.; and Grant, F. H. 1991. Adaptive/Predictive Scheduling: Review and a General Framework. *Production Planning & Control*, 2(4): 298–312.

Rekha, P. M.; and Dakshayini, M. 2019. Efficient Task Allocation Approach Using Genetic Algorithm for Cloud Environment. *Cluster Computing*, 22: 1241–1251.

Reliford, A.; Reid, D.; Smith, S. T.; Andy, A.; Alfred, M.; and Phillips, S. 2026. Autonomous Task Rescheduling in a Heterogeneous LEO Satellite Constellation Using Reinforcement Learning. In *AIAA SCITECH 2026 Forum*. Orlando, FL: American Institute of Aeronautics and Astronautics.

Rema, C.; Sousa, A.; Sobreira, H.; Costa, P.; and Silva, M. F. 2025. Exploring the Potential of LLM-based Chatbots for Task Scheduling in Robot Operations. In *Proceedings of the IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 45–51. Funchal, Portugal.

van Kessel, P. J.; Freeman, F. C.; and Santos, B. F. 2023. Airline Maintenance Task Rescheduling in a Disruptive Environment. *European Journal of Operational Research*, 308(2): 605–621.