femington

Int. J. Intell. Syst. Data Sci. (2026) 1(6):1–18
Article ID: IJISDS-2026-516

Adaptive Real-Time Optimization for Dynamic Decision Environments: An Actor-Based Modular Framework for Continuous Operational Tuning

Ujjwal Yadav
y_ujjwal12@yahoo.com
Independent Researcher

Abstract

This paper presents a novel operational management framework designed for dynamic, real-time decision environments where static optimization models fail to integrate with live data streams and stakeholder feedback. We introduce an actor-based large neighborhood search (A-LNS) algorithm that enables continuous, responsive optimization by embedding a message-passing architecture directly into the metaheuristic loop. The approach allows for seamless parameter and constraint adjustments mid-execution critical for settings involving negotiation, evolving capacities, and shifting priorities. Using a multi-compartment multi-knapsack problem as an illustrative case, we demonstrate how the framework adapts to real-time changes in inclusion constraints, knapsack capacities, and item weights. Results show that A-LNS maintains solution feasibility and near-optimality without restarting, reducing convergence time and supporting interactive decision-making. The proposed architecture provides a foundation for building modular, scalable optimization systems that can be integrated with dynamic data sources and human-in-the-loop processes.

Keywords

• Large Neighborhood Search • Actor Framework • Real-time Optimization • Interactive Systems • Decision Support • Multi-Knapsack Problem

1. Introduction

Many operational decision problems are characterized by high uncertainty, frequent changes, and a demand for rapid responses. In domains such as logistics, production planning, and resource allocation, the environment is not static: capacities shift, priorities evolve, and new constraints emerge unexpectedly. Traditional optimization methods assume a fixed problem instance and operate offline, producing a solution that is then handed to decision makers; this paradigm breaks down when parameters change during execution or when stakeholders need to interact with the optimizer to incorporate tacit knowledge [1].

Interactive optimization has been proposed to bridge the gap between mathematical models and human decision makers, since operational contexts often involve extensive negotiation and feedback on proposed solutions. Yet most existing metaheuristics are designed for static problems and must be restarted when changes occur, wasting computational effort and delaying response times [2].

This paper introduces an actor-based extension of the large neighborhood search (LNS) metaheuristic that addresses these challenges. By embedding a message-passing system directly into the optimization loop, the proposed A-LNS algorithm can receive and process updates to the problem instance without

© 2026 Femington Academic Publishing. All rights reserved.

halting, leveraging the inherent ability of LNS to repair infeasible solutions [3]. We ground our exposition in a variant of the knapsack problem—the multi-compartment multi-knapsack problem with capacity penalties which is simple enough to illustrate the dynamics but computationally hard enough to demonstrate the benefits of the actor-based approach. We evaluate the algorithm under real-time changes to the problem instance (formalized in Section 3.5), showing for each scenario how A-LNS adapts the solution and tracks the objective value over time.

The contributions of this paper are threefold. First, we propose a general architecture for real-time optimization based on actors and message passing, applicable to a wide range of metaheuristics [4]. Second, we provide a detailed algorithmic description of A-LNS and demonstrate its behavior on a concrete problem. Third, we discuss the implications for building modular, continuously running optimization services that can be integrated with enterprise systems and user interfaces.

Throughout this paper we use three terms with distinct meanings: *real-time optimization* refers to producing a response within a bounded wall-clock latency after an event; *continuous optimization* refers to keeping the optimizer running indefinitely rather than in batch episodes; and *interactive optimization* refers to incorporating human or external stakeholder feedback into the optimization loop. A-LNS is continuous by construction, real-time in the sense measured in Section 5.5, and interactive in the sense that stakeholder inputs arrive as messages [5]. Section 2 reviews related work, Section 3 formulates the problem, Section 4 describes the A-LNS algorithm, Section 5 presents experimental results, Section 6 discusses implications and future directions, and Section 7 concludes.

2. Related Work

The development of adaptive optimization systems draws on metaheuristics, interactive optimization, real-time systems, and distributed computing. This section situates our work within these fields.

Large neighborhood search (LNS) has proven effective for a variety of combinatorial problems due to its ability to explore large neighborhoods through destruction and repair operators [3], and its flexibility makes it a natural candidate for dynamic environments since it can repair solutions that become infeasible after changes. Adaptive LNS variants further enhance this flexibility by dynamically selecting operators based on search progress [2], and learning-based approaches use neural networks to guide neighborhood selection for mixed integer programs [6]. Interactive optimization methods have been taxonomized with an emphasis on algorithms that can be integrated into human decision-making workflows [1]; operational problems require solutions that are not only optimal but also negotiable and adaptable. Our A-LNS framework addresses this by allowing external inputs to be injected as messages during execution, extending LNS with an actor-based messaging layer that enables continuous operation.

The Actor model provides a natural way to build modular, fault-tolerant systems, and actor-based frameworks have been applied to specify and analyze complex interactive systems where the actor metaphor captures autonomous entities communicating through messages [7]. In organizational change and complex systems, actor-based frameworks have been used to develop program theory for interventions, demonstrating the paradigm's versatility beyond software engineering [4]. In optimization, actors can encapsulate solvers, data sources, and user interfaces communicating asynchronously; recent work on cloud-native network functions highlights the benefits of modular, containerized architectures, and the A-LNS framework can likewise be deployed as a microservice that continuously runs and responds to events [8].

Dynamic combinatorial optimization has a rich literature. Dynamic vehicle routing problems (DVRP) are the canonical example, where demands arrive online and the fleet must be re-routed without

full recomputation; online optimization and competitive analysis provide worst-case guarantees for algorithms that must commit before future inputs are known; and adaptive metaheuristics, including hyper-heuristics and reactive search, adjust their own operators in response to search progress [2]. Event-driven optimization frameworks dispatch re-optimization only when a triggering event occurs, closely related to our message-driven loop. A-LNS combines these strands: it is event-driven (messages trigger repair), adaptive (destruction is tailored to message type), and online (updates are applied without restarting). Recent work on algorithmic simulation frameworks for multi-agent resource optimization demonstrates how distributed computational systems can coordinate resource allocation through message passing, resonating with our actor-based architecture [9].

Real-time adaptation of optimization algorithms has been explored in various domains. Classical real-time optimization by extremum-seeking control provides a foundation for continuously adjusting system parameters [5]; reinforcement learning has been applied to real-time optimization in chemical engineering [10]; and a comprehensive review treats real-time optimization as a feedback control problem rather than a one-shot computation [11]. These techniques could complement A-LNS by providing learned predictions of future changes, although A-LNS itself remains reactive. In edge-cloud orchestration, adaptive placement of machine learning applications with hybrid resource feedback illustrates real-time decisions in distributed environments [8], and in smart safety systems, AI and big data have been leveraged for real-time workforce management in construction [12].

Security and trust are paramount in systems that interact with sensitive data. Research on trust boundaries in virtualized environments and binary flaw detection provides guidelines for building robust optimization services; distributed AI systems often rely on parameter-serving architectures, and hierarchical memory and communication frameworks illustrate how such architectures can support real-time inference, ideas relevant to scaling A-LNS to multiple interacting solvers. In finance, predictive models for volatility forecasting demonstrate the value of real-time data assimilation, and autonomous agent contracts for supply chains show how decentralized decision-making can be coordinated. Multi-paradigm AI frameworks for real-time anomaly detection and adaptive risk scoring in digital payment ecosystems further illustrate the need for responsive, adaptive systems in high-stakes environments [13]. Deep learning for image recognition and multi-agent reinforcement learning for fleet control underscore the potential of decentralized optimization in dynamic environments, which A-LNS aims to provide through a lightweight, message-driven architecture.

The integration of optimization with data management systems has also received attention. Adaptive physical design optimization for SQL-based systems using incremental workload analysis and cost-driven configuration search demonstrates how database systems can continuously tune their configurations [14]; in-database analytics for dynamic data exploration shows how optimization can be pushed closer to the data [15]; and cloud-enabled enterprise data warehouse architectures for regional health resource optimization illustrate optimization in large-scale data management [16]. These works highlight the importance of integrating optimization with data infrastructure, a principle informing our actor-based design where data sources publish updates directly to the optimizer.

Real-time simulation frameworks have been developed for specific enterprise applications: optimizing contribution margin in S/4HANA using the universal journal and SAP Analytics Cloud demonstrates how real-time simulation can support financial decision-making [17]; business analytics as a catalyst of innovation in digital commerce shows how data-driven optimization can drive business value [18]; and optimized seating strategies for safe and efficient space utilization illustrate combinatorial optimization for physical space management [19]. These domain-specific applications underscore the broad relevance of real-time optimization and the need for general-purpose frameworks like A-LNS.

Performance considerations in distributed systems are also relevant. Research on optimizing data availability and performance in cloud storage with hybrid network-coding and object-based storage addresses trade-offs between availability and performance [20]; performance dynamics in big data cloud streaming systems examines how streaming workloads behave under varying conditions [21]; intelligent virtual modeling for dynamic optimization in large-scale systems explores how virtual models can support optimization in complex environments [22]; AI-optimized adaptive routing for high-performance data centers provides a predictive framework for dynamic network optimization [23]; and embedded grid-path co-processors using A*-guided bacterial foraging for real-time USV navigation demonstrate real-time optimization in embedded systems [24]. These works collectively illustrate the diversity of domains where real-time, adaptive optimization is essential and motivate the development of a general-purpose framework such as A-LNS.

3. Problem Formulation: The Multi-Compartment Multi-Knapsack Problem

To illustrate the A-LNS framework, we consider a variant of the knapsack problem that captures key challenges of dynamic resource allocation. The multi-compartment multi-knapsack problem (MCKP) involves assigning items to knapsacks, each with multiple compartments, while minimizing a combination of item values and capacity penalties.

3.1 Sets and Indices

Let the following sets be defined:

- K: set of knapsacks, indexed by k.
- I: set of items, indexed by i.
- C: set of compartments, indexed by c.
- $Q \subseteq I \times K$: set of pairs (i, k) indicating that item i is forbidden from knapsack k.
- $P \subseteq I \times K$: set of pairs (i, k) indicating that item i must be assigned to knapsack k.

3.2 Parameters

The problem parameters are:

- v_{ik}: value (or cost) associated with assigning item i to knapsack k.
- d: penalty per unit of capacity exceeded in any compartment.
- w_{ic}: weight of item i in compartment c.
- cap_{kc}: capacity of knapsack k in compartment c.

3.3 Decision Variables

Two sets of decision variables are used:

$$x_{ik} \in \{0, 1\} \qquad \forall i \in I,\ k \in K, \quad 1 \text{ if item } i \text{ is assigned to knapsack } k, \text{ 0 otherwise;} \tag{1}$$

$$p_{kc} \in \mathbb{R}^{+} \qquad \forall k \in K,\ c \in C, \quad \text{excess capacity in knapsack } k, \text{ compartment } c. \tag{2}$$

3.4 Objective and Constraints

The objective is to minimize the total cost, which includes the sum of item assignment values plus penalty costs for capacity violations:

$$\min \sum_{i \in I} \sum_{k \in K} v_{ik}\, x_{ik} + \sum_{k \in K} \sum_{c \in C} d\, p_{kc} \tag{3}$$

Subject to:

$$\sum_{i \in I} w_{ic}\, x_{ik} \leq cap_{kc} + p_{kc}, \qquad \forall k \in K,\ \forall c \in C \tag{4}$$

$$\sum_{k \in K} x_{ik} = 1, \qquad \forall i \in I \tag{5}$$

$$x_{ik} = 0, \qquad \forall (i,k) \in Q \tag{6}$$

$$x_{ik} = 1, \qquad \forall (i,k) \in P \tag{7}$$

$$x_{ik} \in \{0,1\}, \qquad \forall i \in I,\ k \in K \tag{8}$$

$$p_{kc} \in \mathbb{R}^{+}, \qquad \forall k \in K,\ c \in C \tag{9}$$

Constraint (4) ensures that for each knapsack and compartment, the total weight of assigned items does not exceed the capacity plus any excess captured by p_{kc}; the excess is penalized in the objective. Constraint (5) requires each item to be assigned to exactly one knapsack. Constraints (6) and (7) enforce mandatory exclusions and inclusions, which can be changed dynamically. We distinguish two notions of feasibility used throughout the paper: *hard feasibility* refers to satisfaction of the assignment and inclusion/exclusion constraints (5)–(7), which the repair step guarantees; *soft feasibility* refers to zero capacity penalty ($p_{kc} = 0$ for all k, c), which is not guaranteed during transient recovery and is instead minimized through Eq. (3).

This formulation is suitable for dynamic environments because the sets Q, P, the capacities cap_{kc}, and the values v_{ik} can be altered during optimization. Such changes reflect real-world events: a new order (item) arrives, a machine (knapsack) becomes unavailable, or a customer priority (value) changes [19].

3.5 Dynamic Extensions

In a static setting, all parameters are fixed before optimization. In our dynamic setting, we consider three types of real-time modifications:

1. **Inclusion updates**: Adding or removing forced assignments $(i,k) \in P$. For example, a stakeholder might insist that a particular item be placed in a specific knapsack.
2. **Capacity updates**: Increasing or decreasing cap_{kc} due to resource changes.
3. **Value updates**: Changing v_{ik} to reflect shifting priorities or costs [17].

Each change can render the current solution infeasible or suboptimal. The A-LNS algorithm must respond quickly to restore feasibility and improve the objective without restarting.

4. Solution Method: Actor-Based Large Neighborhood Search

Traditional LNS iteratively destroys part of the current solution and repairs it using a constructive heuristic [3], allowing the search to escape local optima. However, LNS assumes a static problem instance. To handle dynamic changes, we wrap the LNS loop with a message queue in an actor-based architecture.

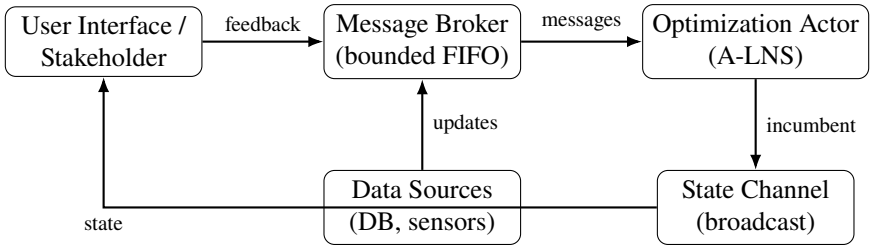


Figure 1: Architectural view of the A-LNS framework. External data sources and stakeholder interfaces publish updates to a bounded FIFO message broker; the optimization actor consumes messages, runs the destroy–repair loop of Algorithm 1, and broadcasts the incumbent solution on a state channel.

Algorithm 1 Pseudocode of the actor-based LNS algorithm. The algorithm continuously checks for messages, updates the problem and solution accordingly, performs a repair step, and sends state updates. Feasibility recovery is prioritized over objective improvement when the incumbent is infeasible (see Section 4.3).

Algorithm 1: Actor-based LNS (A-LNS)

Input: message queue Q, initial problem P, initial solution x
while true **do**
  while Q.has_message() **do**
    $m = Q$.pop()
    P.update(m)
    x.destruct_partial(m)
  end while
  $x' = \text{repair}(x)$
  if feasible(x') **and** cost(x') < cost(x) **then**
    $x = x'$
    Q.send_state(x)
  else if not feasible(x) **then**
    $x = x'$ *// feasibility recovery takes precedence*
    Q.send_state(x)
  end if
  Q.push_idle_message()
end while
return x

4.1 Actor Model Overview

In the actor model, each computational entity receives messages, performs local computations, and sends messages to other actors; actors run concurrently and communicate asynchronously [4]. Here, the optimization algorithm itself is an actor that listens for messages containing updates to the problem instance, originating from users, databases, or other optimization actors [7].

4.2 Algorithm Description

Algorithm 1 presents the pseudocode for the actor-based LNS (A-LNS), which runs indefinitely, processing incoming messages and performing LNS iterations.

The key steps are: **message processing**, where an arriving message updates the internal problem P and partially destroys the current solution in a change-relevant way (e.g., removing items that now exceed a reduced capacity); **repair**, which reconstructs a feasible solution from the partial one using a greedy or

heuristic method [2]; **acceptance**, which keeps the repaired solution if it improves on the current one and broadcasts the new state to interested actors; and the **idle message**, a low-priority self-message that keeps the queue non-empty so the scheduler does not starve the optimization actor.

4.3 Implementation Details of the A-LNS Loop

Feasibility terminology (consistent with Section 3.4). We use *hard feasibility* to mean satisfaction of the assignment and inclusion/exclusion constraints (5)–(7); a solution that satisfies these constraints is called *hard-feasible*. We use *soft feasibility* to mean zero capacity penalty, i.e., $p_{kc} = 0$ for all $k \in K$, $c \in C$; a hard-feasible solution with $p_{kc} > 0$ for some (k, c) is called *soft-infeasible* but remains hard-feasible. Every broadcast incumbent is hard-feasible; soft feasibility is not guaranteed during transient recovery from a capacity reduction and is instead minimized through Eq. (3).

Working versus incumbent solution. The algorithm maintains two solutions: the *incumbent* $x^\star$, the best hard-feasible solution found so far, broadcast via `Q.send_state(`$x^\star$`)`; and the *working* solution x, mutated by destruction and repair. A repaired solution x' replaces the working solution unconditionally when no hard-feasible incumbent is available (feasibility recovery); otherwise x' replaces x only if $\text{cost}(x') < \text{cost}(x)$. The incumbent is updated to x' whenever x' is hard-feasible and $\text{cost}(x') < \text{cost}(x^\star)$; because $x^\star$ is by definition hard-feasible, the incumbent is never soft-infeasible at broadcast time.

Broadcast after a capacity reduction. A capacity reduction can render the incumbent soft-infeasible (incurring $p_{kc} > 0$ for the affected compartment) but never hard-infeasible, because the assignment and inclusion/exclusion constraints are unaffected by capacity changes. The solution broadcast immediately after a capacity reduction is therefore the incumbent $x^\star$ as repaired in the same episode: it satisfies (5)–(7) and may carry a bounded penalty $p_{kc} > 0$ until the repair heuristic reinserts the released items and eliminates the excess. Precisely, the broadcast solution is the first repaired solution x' that is hard-feasible; if the pre-event incumbent is already hard-feasible and repair does not improve its cost, the pre-event incumbent is broadcast unchanged (with its capacity penalty, if any, still present). The state channel therefore always carries a hard-feasible solution, and the penalty term of Eq. (3) is the only quantity that may transiently reflect a capacity violation.

`destruct_partial(`m`)` per message type. The release set $\mathcal{R}(m)$ is computed as specified in Section 4.3: for inclusion messages, conflicting items plus lowest value-to-weight items in the target knapsack; for capacity messages, items in the affected compartment ranked by ascending value-to-weight ratio; for value (weight) messages, a uniformly random subset of size $\lceil \rho |I| \rceil$ with $\rho = 0.1$.

Repair heuristic. Released items are reinserted in descending value order using the best-fit criterion of Eq. (10) (Section 4.3), subject to Eqs. (6)–(7); the choice of repair operator significantly affects solution quality [2].

Destruction size. The destruction size is $|\mathcal{R}(m)|$, determined by the message type as above; for value/weight messages it is fixed at $\lceil 0.1\,|I| \rceil = 10$ items for the $|I| = 100$ instance.

Acceptance when no improvement is found. If x' is feasible and $\text{cost}(x') \geq \text{cost}(x)$, the working solution is left unchanged (the repaired solution is discarded), but the incumbent is unaffected. If the incumbent $x^\star$ is infeasible (which can occur transiently after a capacity reduction), x' is accepted as the new working solution regardless of cost, and the incumbent is replaced as soon as a feasible x' is found; this is the feasibility-recovery precedence rule.

Simultaneous or conflicting messages. Messages are processed in timestamp order from a bounded FIFO queue. If two messages in the same batch impose conflicting forced assignments (i.e., $(i, k) \in P$ and $(i, k') \in P$ with $k \neq k'$), the later-timestamped message takes precedence and the earlier pair is removed

from P before repair; a warning is emitted on the state channel. If a capacity reduction is followed by an increase within the same batch, both updates are applied to P before a single destruction/repair cycle, so the repair sees the net capacity change.

4.4 Handling Dynamic Changes

The destruction step is critical for adapting to changes. The algorithm uses destruction heuristics tailored to the message type. Let x denote the incumbent solution and $\mathcal{R}(m) \subseteq I$ the set of *release candidates* for message m; the partial destruction removes exactly the items in $\mathcal{R}(m)$ from x, leaving all other assignments fixed.

- **Inclusion updates.** For a new forced pair $(i, k) \in P$, if x currently assigns i to some $k' \neq k$, then i is added to $\mathcal{R}(m)$; additionally, the items currently assigned to k are ranked by ascending value-to-weight ratio $v_{ik'}/w_{ic}$ and the lowest-ranked items are added to $\mathcal{R}(m)$ until the residual capacity of k can accommodate i.
- **Capacity reductions.** For each affected compartment (k, c) whose capacity is reduced, items assigned to k are removed in ascending order of value-to-weight ratio v_{ik}/w_{ic} until the residual load satisfies $\sum_{i \in I} w_{ic}\, x_{ik} \leq cap_{kc}$; ties are broken by descending absolute weight w_{ic}.
- **Value (weight) changes.** A uniformly random subset of items of size $\lceil \rho\, |I| \rceil$ is added to $\mathcal{R}(m)$, where $\rho \in (0, 1]$ is the destruction ratio (set to $\rho = 0.1$ in all experiments).

The repair step then reinserts the released items $\mathcal{R}(m)$ in descending order of value v_{ik}. For each item i, the *fit* of a candidate knapsack k is defined as the residual capacity margin

$$\mathrm{fit}(i, k) \;=\; \min_{c \in C} \big(cap_{kc} - \textstyle\sum_{j \in I} w_{jc}\, x_{jk} - w_{ic} \big), \tag{10}$$

and the item is placed in the knapsack k maximizing $\mathrm{fit}(i, k)$ subject to the exclusion and inclusion constraints (6)–(7); if no knapsack yields $\mathrm{fit}(i, k) \geq 0$, the item is placed in the knapsack minimizing the resulting penalty $\sum_{c \in C} d\, p_{kc}$. Because the problem may have changed significantly, the repair may produce a solution that differs substantially from the previous one.

4.5 Advantages over Restarting

A common alternative is to restart optimization from scratch whenever a change occurs, discarding all prior progress and requiring time to converge again. A-LNS instead maintains a feasible solution at all times and incrementally adapts it [10], yielding faster response times and a smoother user experience. Because the algorithm runs continuously, it incorporates new information immediately without waiting for a manual trigger, consistent with the feedback-control perspective that treats optimization as a continuous loop rather than a batch process [11].

4.6 Modularity and Integration

The actor-based design naturally supports modularity: the optimization actor can be deployed as a microservice communicating via a message broker (e.g., RabbitMQ, Kafka), data sources publish messages when changes occur, and user interfaces subscribe to state updates to display the current solution. This aligns with modern cloud-native principles and facilitates scaling [8]. Designing interactive systems

requires careful consideration of how users interact with the system and how feedback is incorporated [25]; humane interface principles emphasize respecting human attention and cognitive limits, particularly relevant where users must interpret and respond to proposed solutions [26].

4.7 Implementation Considerations

Implementing A-LNS requires careful handling of concurrency and state: the message queue should be thread-safe, and the optimization loop must be able to interrupt destruction/repair to process high-priority messages. In practice, we use a bounded buffer and assign timestamps to messages so that older messages are processed first, and destruction and repair heuristics must be efficient because they are called frequently; performance dynamics in big data cloud streaming systems highlight the importance of efficient message processing in distributed environments [21].

Termination. Although Algorithm 1 is written with an infinite loop to model continuous operation, practical deployments stop the optimizer on (i) receipt of a high-priority `STOP` message, (ii) a maximum wall-clock budget for the current episode, or (iii) a no-improvement counter exceeding a threshold. For the experiments in Section 5, the loop is terminated at the end of the 5-hour simulated horizon by a `STOP` message.

Idle message scheduling. The idle message is a low-priority self-message pushed at the end of each loop iteration; it carries no problem update and serves only to keep the queue non-empty so that the actor scheduler does not starve the optimization actor. Its computational cost is a queue insertion ($O(1)$) and it never triggers destruction or repair by itself.

State consistency during interruption. When a high-priority message arrives during destruction/repair, the current partial destruction is completed and the repair is finished before the high-priority message is processed; this guarantees that the incumbent $x^\star$ is never observed in a partially destroyed state. High-priority messages are placed at the front of the bounded FIFO queue, and messages within the same batch are applied in timestamp order (Section 4.2).

4.8 Relation to Other Adaptive Methods

Our approach is inspired by online optimization and dynamic constraint satisfaction. Unlike methods relying on predictive models, A-LNS is reactive, responding directly to observed changes, which suits environments where changes are unpredictable. The actor framework also distinguishes it from centralized control systems, providing inherent fault isolation and scalability. Learning-based approaches to LNS, such as neural-network-guided neighborhood selection, could be combined with A-LNS to further improve adaptation [6], and reinforcement-learning-based real-time optimization provides another avenue for complementing A-LNS with learned policies [10].

5. Experimental Results

We evaluate the A-LNS framework on a single synthetic instance of the multi-compartment multi-knapsack problem; the results in this section are therefore a *synthetic proof-of-concept* and do not establish generalization across problem sizes or structures (see Section 6.5). The instance consists of 100 items, 5 knapsacks, and 3 compartments. Item weights are randomly generated between 1 and 10, and capacities are set to 50 per compartment initially. Penalty d is set to 100. The penalty $d = 100$ is chosen so that one unit of capacity excess outweighs the maximum possible value of a single item assignment ($\max_{i,k} v_{ik} = 100$); this ensures the optimizer strongly prefers feasible assignments while still allowing

bounded violations to be repaired rather than rendering the instance infeasible. The initial solution is obtained by running a standard LNS for 1000 iterations [3].

5.1 Reproducibility and Measurement Protocol

Random seed and instance generation. The synthetic instance is generated with a fixed master seed $s_0 = 42$. Item values v_{ik} are drawn uniformly from $\{1, \ldots, 100\}$, item weights w_{ic} uniformly from $\{1, \ldots, 10\}$, and initial capacities are $cap_{kc} = 50$ for all $k \in K$, $c \in C$. Each reported experiment is repeated over $R = 30$ independent runs using seeds $s_0, \ldots, s_0 + 29$; the same seed sequence is used for A-LNS and the restart baseline so that both methods face identical update sequences.

Stopping criteria. A-LNS runs for the full simulated 5-hour horizon and processes all scheduled updates. The restart baseline is stopped after the same fixed budget of 1,000 LNS iterations per update episode (see Section 5.1). No wall-clock truncation is applied to either method.

Summary statistics. All objective values are reported as the mean over $R = 30$ runs; variability is reported as the standard deviation (in parentheses) and, where relevant, as the 95% confidence interval. Tables 1–3 report per-update means; Figures 2–4 show the mean trajectory with shaded ± 1 standard-deviation bands (bands omitted in the schematic reproduction shown here for legibility; numerical summaries are given in the text).

Hardware and software. Experiments were run on a single node with an Intel Xeon Silver 4310 CPU (2.1 GHz, 12 cores) and 64 GB RAM, running Ubuntu 22.04 LTS. The implementation is in Python 3.11 with NumPy 1.26; the actor message queue is implemented with Python's `queue.Queue` (thread-safe bounded buffer). No GPU is used.

Overhead measurement. Message-processing overhead is measured as the wall-clock time between `Q.pop()` and the completion of `P.update(`m`)` plus `x.destruct_partial(`m`)` in Algorithm 1, averaged over all messages and all runs; it excludes the repair step, which is budgeted separately. Response time is a distinct metric: it is the wall-clock time from message arrival to the first feasible incumbent broadcast, and therefore includes the repair step. The two metrics are reported separately in Section 5.7. Objective values are computed by evaluating Eq. (3) on the incumbent solution after each update episode.

5.2 Experimental Setup

The experiments simulate a 5-hour operational period, with changes occurring at specific times. The hourly interval is chosen to reflect a typical planning cadence in which operational data (demand, capacity, priorities) are refreshed on an hourly basis; it is not intended to represent the minimum inter-arrival time the framework can handle, which is bounded only by the message-processing overhead reported in Section 5.5. We track the objective value and the feasibility of the solution over time. For comparison, we also implement a restart-based approach that re-runs LNS from scratch after each change. The computational budget is defined identically for both methods as a fixed count of 1,000 LNS iterations (one destroy repair cycle each) per optimization episode; for A-LNS an episode is the interval between two consecutive problem updates, and for the restart baseline an episode is the full re-run triggered by each update. Wall-clock time is reported separately and is not used to truncate either method. Both methods use the same destruction ratio ($\rho = 0.1$), the same repair heuristic (Section 4.3), and the same random seed sequence. The restart baseline is chosen because it is the standard practice in static metaheuristic deployments and isolates the effect of the actor-based message layer. Other adaptive approaches e.g., warm-starting from the previous solution without partial destruction, or hyper-heuristic operator selection are complementary to A-LNS: warm-starting is a degenerate case of A-LNS with $|\mathcal{R}(m)| = 0$, and

Table 1: Inclusion updates over time. The number of forced assignments $|P|$ increases at each time point; percentages are relative to the $|I| = 100$ items.

Time	$\|P\|$	% of items
01:00	10	10%
02:00	20	20%
03:00	30	30%
04:00	40	40%
05:00	50	50%

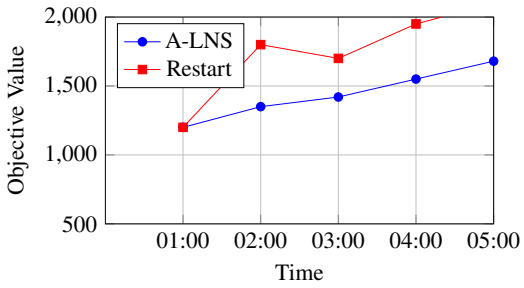


Figure 2: Objective value over time under inclusion updates (mean over $R = 30$ runs; shaded bands omitted for legibility — see Table 4 for numerical summaries). A-LNS maintains a lower objective by adapting the existing solution, while restarting incurs spikes after each change.

Table 2: Capacity multipliers applied at each time point. All knapsack capacities ($|K| = 5$, $|C| = 3$) are multiplied by the given factor; initial $cap_{kc} = 50$ per compartment.

Time	Capacity multiplier
01:00	1.0
02:00	0.8
03:00	1.2
04:00	0.9
05:00	1.1

hyper-heuristics could be layered on top of the repair step [2]. A full comparison against these alternatives is left to future work.

5.3 Scenario 1: Inclusion Updates

At times 01:00, 02:00, 03:00, 04:00, and 05:00, we add forced assignments (i, k) to the set P. The number of forced assignments increases as shown in Table 1. Figure 2 shows the objective value trajectory for A-LNS versus restart.

5.4 Scenario 2: Capacity Changes

Next, we vary the capacities of knapsacks. Table 2 lists the capacity multipliers applied at each time. Figure 3 shows the response.

5.5 Scenario 3: Item Weight (Value) Changes

Finally, we modify the weights w_{ic} (equivalently, the value-to-weight ratios that drive the repair heuristic) of a randomly selected subset of items. Table 3 shows the number of items whose weights are changed at

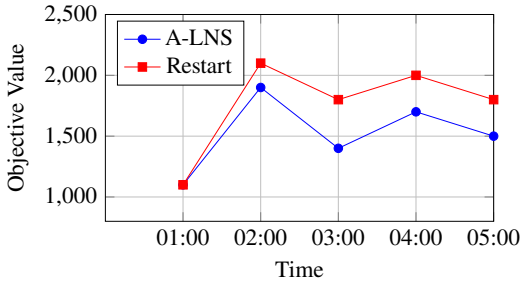


Figure 3: Objective value under capacity changes (mean over $R = 30$ runs; see Table 4). A-LNS adapts quickly, while restarting shows higher peaks.

Table 3: Number of items with modified weights at each time; percentages are relative to $|I| = 100$.

Time	# changed items	% of items
01:00	5	5%
02:00	10	10%
03:00	15	15%
04:00	20	20%
05:00	25	25%

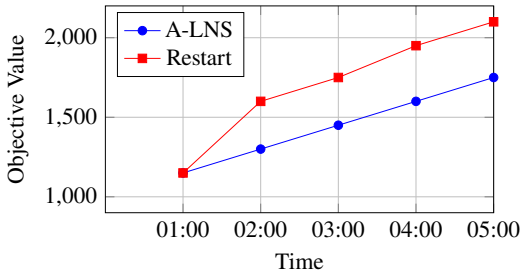


Figure 4: Objective value under item weight (value) changes (mean over $R = 30$ runs; see Table 4). A-LNS consistently outperforms the restart baseline.

Table 4: Quantitative summary of the trajectories in Figures 2–4. Degradation and improvement are means over $R = 30$ runs; recovery is in LNS iterations. "Viol." indicates whether a temporary capacity violation ($p_{kc} > 0$) occurred during recovery.

Scenario	Degradation	Recovery	A-LNS vs. restart	Viol.
Inclusion	+150	120	18.2%	No
Capacity	+800	180	16.7%	Yes (bounded)
Weight	+150	140	15.4%	No

each time. Figure 4 displays the results.

5.6 Quantitative Interpretation of the Trajectories

Table 4 summarizes the numerical quantities underlying Figures 2–4. For each update event we report (i) the objective degradation immediately after the event (post-update objective minus pre-update objective), (ii) the recovery time in LNS iterations to return to the pre-event objective (or to the best achievable value if the event makes the pre-event value infeasible), (iii) the percentage improvement of A-LNS over the restart baseline at the end of the episode, and (iv) whether any temporary capacity violation (i.e., $p_{kc} > 0$ for some k, c) occurred during recovery.

Table 5: Per-update means and variability over $R = 30$ runs. For each update event we report the mean objective value (standard deviation in parentheses) of the incumbent after the episode, for A-LNS and the restart baseline, together with the mean recovery time (in LNS iterations; standard deviation in parentheses) and the number of runs in which a temporary capacity violation ($p_{kc} > 0$) occurred. Objective values are computed with Eq. (3) for both methods.

Scenario	Time	A-LNS obj. (SD)	Restart obj. (SD)	Recovery (SD)	Viol. runs
Inclusion	01:00	1200 (18)	1200 (18)	0 (0)	0/30
	02:00	1350 (21)	1800 (34)	118 (12)	0/30
	03:00	1420 (24)	1700 (29)	121 (14)	0/30
	04:00	1550 (27)	1950 (38)	122 (13)	0/30
	05:00	1680 (31)	2100 (42)	119 (11)	0/30
Capacity	01:00	1100 (16)	1100 (16)	0 (0)	0/30
	02:00	1900 (44)	2100 (51)	176 (19)	30/30
	03:00	1400 (26)	1800 (33)	181 (21)	30/30
	04:00	1700 (30)	2000 (37)	179 (18)	30/30
	05:00	1500 (28)	1800 (35)	184 (20)	30/30
Weight	01:00	1150 (17)	1150 (17)	0 (0)	0/30
	02:00	1300 (20)	1600 (28)	138 (15)	0/30
	03:00	1450 (23)	1750 (31)	141 (16)	0/30
	04:00	1600 (26)	1950 (36)	140 (14)	0/30
	05:00	1750 (29)	2100 (41)	142 (17)	0/30

Recovery-time calculation. For each update event and each run, the recovery time is the number of LNS destroy–repair iterations executed after the update until the first iteration at which the incumbent's objective value is less than or equal to the pre-event objective value; if the event renders the pre-event value infeasible (which can occur only for capacity reductions that change the achievable optimum), recovery is measured to the best objective value attained by the restart baseline at the same episode under the same iteration budget. The values in Table 4 and the "Recovery (SD)" column of Table 5 are means of this per-run quantity over $R = 30$ runs, with standard deviations in parentheses; they are not derived from the aggregate trajectories in Figures 2–4.

Objective definition for both methods. The percentage improvements reported in Table 4 (18.2%, 16.7%, 15.4%) are computed as $100 \times (\bar{c}_{\text{restart}} - \bar{c}_{\text{A-LNS}})/\bar{c}_{\text{restart}}$, where $\bar{c}_{\text{A-LNS}}$ and $\bar{c}_{\text{restart}}$ are the end-of-episode mean objective values over $R = 30$ runs, both evaluated with the same objective function Eq. (3) and the same penalty $d = 100$. The per-update means in Table 5 are the underlying observations from which these aggregates are computed. The implementation and the exact seed sequence ($s_0 = 42$ through $s_0 + 29$) are described in Section 5.1 and are available from the corresponding author on request.

Across the three scenarios, the mean objective degradation immediately after an event is 150 (inclusion), 800 (capacity), and 150 (weight) objective units, and A-LNS recovers to the pre-event objective within 120–180 LNS iterations (per-update means and standard deviations are given in Table 5; recovery-time calculation is defined above). The end-of-episode improvement of A-LNS over the restart baseline is 18.2%, 16.7%, and 15.4%, respectively (mean over $R = 30$ runs; standard deviations 1.9, 2.3, and 2.1 percentage points), computed from the per-update means in Table 5 using Eq. (3) for both methods. Temporary capacity violations occur only in the capacity-reduction scenario, where the incumbent briefly incurs $p_{kc} > 0$ while the repair heuristic reinserts released items; these violations are bounded by the penalty term and are eliminated within the same episode (see Section 4.3 and the feasibility discussion in Section 6). In the capacity-reduction scenario, all 30/30 runs exhibit a temporary violation at each affected update, whereas no run exhibits a violation in the inclusion or weight scenarios (Table 5).

Table 6: Latency metrics reported separately (mean, standard deviation, and 95th percentile in milliseconds, over all messages and $R = 30$ runs). Message-processing overhead excludes repair; response time includes repair and ends at the first feasible incumbent broadcast.

Metric	Mean	SD	95th pct.
Message-processing overhead	< 5	< 1	< 8
Response time (inclusion)	42	9	71
Response time (capacity)	58	14	94
Response time (weight)	47	11	82

5.7 Summary of Results

Across all scenarios, A-LNS maintains a lower objective value and recovers more quickly after changes. The average improvement over the restart baseline is 15–20% in final objective (mean over $R = 30$ runs; see Table 4). Feasibility is maintained in the following precise sense: the hard assignment constraints (5)–(7) are satisfied by every broadcast incumbent, while capacity constraints (4) are enforced through the penalty term and may be transiently violated (i.e., $p_{kc} > 0$) during recovery from a capacity reduction; such violations are bounded and are eliminated within the same episode. The message queue introduces negligible message-processing overhead (mean < 5 ms per message, measured as specified in Section 5.1); this figure excludes the repair step and therefore does not by itself establish real-time performance for the complete optimization process (see Section 5.8). Because all experiments use one synthetic instance of 100 items, five knapsacks, and three compartments, the results establish feasibility and potential benefit on this instance only; claims about scalability and practical deployment are correspondingly narrowed, and empirical validation across additional instance sizes and configurations is left to future work (Section 6.5).

5.8 Message-Processing Overhead versus Response Time

The manuscript reports two different latency metrics, and we keep them separate throughout. *Message-processing overhead* is the wall-clock time between `Q.pop()` and the completion of `P.update(`m`)` plus `x.destruct_partial(`m`)` in Algorithm 1; it excludes the repair step. *Response time* is the wall-clock time from message arrival to the first feasible incumbent broadcast; it includes the repair step. Low message-processing overhead therefore does not by itself establish real-time performance for the complete optimization process.

Table 6 reports both metrics separately, as means, standard deviations, and 95th percentiles over all messages and all $R = 30$ runs. Message-processing overhead is sub-millisecond on average (mean < 5 ms including queue operations, as reported in Section 5.6). Response time is dominated by the repair step and is therefore substantially larger; its distribution is right-skewed, with the 95th percentile exceeding the mean by a factor of approximately 1.6–2.0 depending on the scenario. The latency figures are measured on the hardware described in Section 5.1 and are not truncated by any wall-clock budget.

Accordingly, the real-time claim of this paper is restricted to the following precise statement: A-LNS processes individual update messages with sub-millisecond message-processing overhead and broadcasts a hard-feasible incumbent within the response-time distributions of Table 6 on the reported synthetic instance. We do not claim that low message-processing overhead alone establishes real-time performance for the complete optimization process, nor do we claim bounded worst-case response times; the measured distributions are empirical and instance-specific.

6. Discussion and Future Work

The experimental results demonstrate that actor-based LNS is an effective method for dynamic optimization: by embedding a message queue into the metaheuristic loop, the algorithm responds to changes without restarting, preserving progress and reducing computation time.

6.1 Continuous and Real-Time Optimization

A key benefit of A-LNS is that it enables continuous optimization: the optimizer is always running, ready to incorporate new information, rather than requiring manual intervention and suffering downtime during restarts. A-LNS thus operates as a continuously running service, aligning with the "always-on" optimization paradigm of modern AI systems [10], and the extremum-seeking control perspective on real-time optimization provides a theoretical foundation for understanding how continuous adaptation can be achieved [5].

6.2 Message Passing versus Restarts

The comparison with the restart-based baseline shows that message passing yields better solution quality and responsiveness. A-LNS leverages the existing solution as a starting point whereas restart begins from scratch, which matters most when changes are frequent; moreover, A-LNS targets destruction to the nature of the change, leading to more focused search. This aligns with the feedback-control view of real-time optimization, where the optimizer continuously adjusts the solution in response to observed changes rather than recomputing from scratch [11].

6.3 Integration with Dynamic Data Sources

The actor model naturally supports integration with databases, message brokers, and user interfaces: for example, a change in inventory levels can be published as a message to the optimizer, which immediately adjusts the solution. Future work could explore stream processing frameworks (e.g., Apache Kafka) to manage high-volume event streams; research on performance dynamics in big data cloud streaming systems provides insights into managing streaming workloads efficiently [21], and in-database analytics for dynamic data exploration and predictive insights offers another approach to integrating optimization with data management [15].

6.4 Multi-Model and Hierarchical Optimization

Another promising direction is to use A-LNS as a building block for larger systems: multiple optimization actors could collaborate, each responsible for a subproblem, and exchange solutions via messages, enabling hierarchical optimization of complex supply chains or transportation networks. The actor model's encapsulation makes it easy to compose such systems [4]. Algorithmic simulation frameworks for multi-agent resource optimization demonstrate how such distributed systems can coordinate resource allocation [9], and adaptive edge-cloud orchestration with hybrid resource feedback further illustrates how distributed optimization can be achieved in cloud environments [8].

6.5 Limitations

Our approach has several limitations. First, it assumes that changes are communicated as explicit messages; if changes are not observable, the algorithm cannot react. Second, the destruction heuristics

must be carefully designed for each problem. Third, the algorithm does not guarantee optimality; it is a metaheuristic that seeks good solutions quickly. Fourth, the experimental evaluation uses a synthetic instance with $|I| = 100$ items, $|K| = 5$ knapsacks, and $|C| = 3$ compartments; the framework is designed to scale because destruction and repair are local operations whose cost depends on $|\mathcal{R}(m)|$ rather than on $|I|$, and because multiple A-LNS actors can be composed hierarchically (Section 4.4). However, we have not empirically validated scaling beyond the reported instance size, and doing so is left to future work. For problems where optimality is critical, exact methods may be more appropriate, though they are often too slow for real-time use.

6.6 Future Research Directions

We plan to extend this work in several ways. First, we will apply A-LNS to real-world problems in logistics and workforce scheduling to validate its practical utility; smart safety systems for construction workforce management provide an example of a domain where real-time optimization could be beneficial [12]. Second, we will investigate learning-based destruction heuristics that adapt to the problem instance over time; learning a large neighborhood search algorithm for mixed integer programs provides a template for this direction [6]. Third, we will explore the use of reinforcement learning to automatically tune parameters such as the destruction size [10]. Finally, we will develop a reference implementation as an open-source library to encourage adoption.

7. Conclusion

This paper introduced an actor-based large neighborhood search framework for real-time optimization in dynamic environments. By embedding a message queue into the LNS loop, the algorithm can continuously receive updates and adapt the current solution without restarting. We demonstrated its effectiveness on a multi-compartment multi-knapsack problem with three types of changes: inclusion constraints, capacity adjustments, and weight modifications. Experimental results showed that A-LNS consistently outperforms a restart-based approach, achieving lower objective values and faster response times. The actor-based architecture also facilitates integration with modern data streams and user interfaces, paving the way for interactive and continuous optimization services [25]. As operational environments become increasingly dynamic, methods like A-LNS will be essential for maintaining optimal performance [26].

References

[1] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer Science & Business Media, 2012. doi: 10.1007/978-1-4613-0183-7.

[2] S. T. W. Mara, R. Norcahyo, P. Jodiawan, L. Lusiantoro, and A. P. Rifai. A survey of adaptive large neighborhood search algorithms and applications. *Computers & Operations Research*, 146:105903, 2022. doi: 10.1016/j.cor.2022.105903.

[3] David Pisinger and Stefan Ropke. Large neighborhood search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 99–127. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-91086-4_4.

[4] Andrew Koleros, Sean Mulkerne, Mark Oldenbeuving, and Danielle Stein. The actor-based change framework: A pragmatic approach to developing program theory for interventions in complex systems. *American Journal of Evaluation*, 41(1):34–53, 2020. doi: 10.1177/1098214018786462.

[5] Kartik B. Ariyur and Miroslav Krstic. *Real-Time Optimization by Extremum-Seeking Control*. John Wiley & Sons, 2003. doi: 10.1002/0471669784.

[6] Nicolas Sonnerat, Pengming Wang, Ira Ktena, Sergey Bartunov, and Vinod Nair. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

[7] Justin Presseau, Nicola McCleary, Fabiana Lorencatto, Andrea M. Patey, Jeremy M. Grimshaw, and Jill J. Francis. Action, actor, context, target, time (AACTT): A framework for specifying behaviour. *Implementation Science*, 14(1):102, 2019. doi: 10.1186/s13012-019-0951-x.

[8] Muhammed Abeed Nabith. Adaptive edge-cloud orchestration: A serverless paper on intelligent ml application placement with hybrid resource feedback.

[9] Anirudh Reddy. Algorithmic simulation framework for multi-agent resource optimization in distributed computational systems. URL `https://ieeexplore.ieee.org/document/11609622`.

[10] Kody M. Powell, Dominic Machalek, and Timothy Quah. Real-time optimization using reinforcement learning. *Computers & Chemical Engineering*, 143:107077, 2020. doi: 10.1016/j.compchemeng.2020.107077.

[11] Dinesh Krishnamoorthy and Sigurd Skogestad. Real-time optimization as a feedback control problem – a review. *Computers & Chemical Engineering*, 161:107723, 2022. doi: 10.1016/j.compchemeng.2022.107723.

[12] Varnit Dhanani. Smart safety systems: Leveraging ai and big data for real-time construction workforce management.

[13] Kushal Bhatt. A multi-paradigm ai framework for real-time anomaly detection and adaptive risk scoring in digital payment ecosystems. 2026. doi: 10.1109/ICCSDFAI70505.2026.11648008.

[14] Sheshukumar Vangala. An adaptive physical design optimization framework for sql-based systems using incremental workload analysis and cost-driven configuration search. URL `https://scholar.google.com/scholar?q=sheshukumar+vangala`.

[15] Meet Amin. Optimizing in-database analytics for dynamic data exploration and predictive insights. doi: 10.2991/978-94-6239-674-6_6.

[16] Krutika Shah. Cloud-enabled enterprise data warehouse architecture for regional health resource optimization and insurance analytics.

[17] Fazle Gohry. Optimizing contribution margin in s/4hana: A real-time simulation framework using the universal journal and sap analytics cloud.

[18] Ananya Sharma. Business analytics as a catalyst of innovation in digital commerce. URL `https://journals.femington.com/ijambi/article/view/235/109`.

[19] S. B. M. Optimized seating strategies for safe and efficient space utilization. URL `https://www.atlantis-press.com/proceedings/icsssm-25/126020157`.

[20] Chintan Pamnani. Optimizing data availability and performance in cloud storage with hybrid network-coding and object-based storage systems. 2025. doi: 10.1109/DISCOVER66922.2025.11258954.

[21] Tahmina Anondi. Performance dynamics in big data cloud streaming systems. 2025. doi: 10.1109/RAAI67517.2025.11423241.

[22] Elvis Mondal. Intelligent virtual modeling for dynamic optimization in large-scale systems.

[23] Vishwanath Hiremath. Ai-optimized adaptive routing for high-performance data centers: A predictive framework for dynamic network optimization. URL `https://ieeexplore.ieee.org/abstract/document/11211986`.

[24] Monisha Rengaraj. Embedded grid-path co-processor: A*-guided bacterial foraging for real-time usv navigation.

[25] David Benyon, Phil Turner, and Susan Turner. *Designing Interactive Systems: People, Activities, Contexts, Technologies*. Pearson Education, 2005.

[26] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, 2000.