Politecnico di Torino

Master's degree programme in
Mathematical Engineering



Greedy Policies versus Rolling-Horizon Optimization for Task Design in Robotic Mobile Fulfillment Systems

Supervisor
Paolo Brandimarte

Candidate
Martina Chiodo

Academic Year 2025/2026

Graduation Session September 2026

Summary

The Robotic Mobile Fulfillment System (RMFS) is a parts-to-picker material handling system increasingly adopted in e-commerce warehouses, where autonomous mobile robots (AMRs) bring storage pods to stationary pickers. Making such a system perform well requires coordinating a wide range of decisions across different planning levels: from the real-time routing and traffic management of the AMRs on the warehouse floor, to the replenishment of pod inventory, to the tactical coordination of orders, pods and workstations over an operating period. This thesis focuses on the last of these, and specifically on the picking-side decisions that shape the workload at the stations, leaving routing and replenishment outside its scope.

Within this scope, the decisions to be made are of two kinds. The first are assignment decisions: which orders are handled at which picking workstation and which pods are selected to serve them. The second is the scheduling of the pod movements, the *tasks*, that these assignments define.

This work studies two alternative strategies for making these decisions. The first applies greedy policies online: each decision is triggered by an event, such as the arrival of an order, and is made from the current state of the system alone without knowledge of what is yet to come. The second re-optimizes periodically over a batch of orders through an optimizer based on a two-phase decomposition of the problem. A first phase fixes the assignments: each order is assigned to a single workstation and to one or more pods that together hold the required SKUs. These assignments are chosen to minimize the number of pods to be moved, which is equivalent to maximizing the *pile-on*, the average number of units picked from a pod each time it visits a workstation [1], since the more orders each pod serves per visit, the fewer pod trips are needed overall. Once the assignments are fixed, it is known which pods must serve which workstations, and the second phase then schedules the resulting picking operations and sequences the opening of the orders at the workstations. This scheduling relies on a time-space network that accounts for pod travel times, ensuring the schedule respects pod travel times. Since these times are deterministic, obtained as distance over speed, they act as a proxy for the stochastic real system: the resulting picking schedule is best understood not as a plan to be executed exactly, but as a trace. Both phases are handled by a Large

Neighborhood Search heuristic, made necessary in particular by the time-space network, which makes the second phase computationally too expensive to solve exactly vusing a comercial sover. At each iteration a subset of the decision variables is freed and re-optimized with a MIP solver over the resulting smaller model, while the rest are kept fixed.

The two strategies are compared within a discrete-event simulation framework, designed and implemented from scratch as part of this work, that reproduces the warehouse under a continuous stream of orders and stochastic travel and picking times, and collects performance indicators such as order flow times, throughput and resource utilization. The comparison highlights the trade-off between the two strategies. The greedy policies are simple and reactive, whereas the optimizer requires a heavier computational and implementation effort. This effort, however, pays off: thanks to its global view and its ability to coordinate decisions, the optimizer achieves a higher throughput than the reactive policies across all the configurations considered. This advantage nonetheless rests on the local search being given enough time to reach a near-optimal solution within each decision epoch.

Acknowledgements

I would like to thank my supervisor, Professor Paolo Brandimarte, for his guidance through this thesis and for his valuable feedback, which helped me improve this work.

A special thanks goes to my mother Mara, my father Antonio, and my brother Federico, for always supporting my choices and for their unwavering belief in me and in my abilities.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

2 Literature Review 3
2.1 Decision Sub-Problems in RMFS Operations . . . 3
2.2 Solution Approaches . . . 5
2.2.1 Sequential Decomposition Approaches . . . 5
2.2.2 Integrated Approaches . . . 6

3 Problem Formulation 9
3.1 Problem Overview . . . 9
3.2 Time-Space Network Representation . . . 12
3.3 Mathematical Modelling . . . 13
3.3.1 Stage 1: Assignment Problem . . . 15
3.3.2 Stage 2: Task-Scheduling Problem . . . 16
3.4 Size and Complexity . . . 19

4 Heuristic Solution Method 21
4.1 Large Neighborhood Search framework . . . 21
4.2 Stage 1 Solution Approach . . . 23
4.3 Stage 2 Solution Approach . . . 24
4.4 Validation on Small Test Cases . . . 26

5 Evaluation Framework 29
5.1 Discrete-Event Simulation . . . 30
5.2 Simulator Architecture . . . 30
5.2.1 System Entities and State Variables . . . 31
5.2.2 Simulation Engine . . . 32

5.2.3 Initialization . . . 32
5.2.4 Events . . . 32
5.2.5 KPI Collector . . . 35
5.3 Decision-Making . . . 36
5.3.1 Local Policies . . . 37
5.3.2 Optimizer Mapping . . . 38
5.4 Input Parameters . . . 40

6 Computational Experiments 45
6.1 Design of Experiments . . . 45
6.2 Throughput Analysis . . . 48
6.2.1 Factors affecting the optimizer's advantage . . . 50
6.3 Mean Flow Time Analysis . . . 51
6.4 Pod Movement Analysis . . . 53

7 Conclusion 57

A Source Code 61

B Detailed Computational Results 65

Bibliography 75

List of Tables

2.1 Decision sub-problems addressed in the reviewed literature 4

3.1 Notation used in the mathematical formulation 14
3.2 Number of decision variables and constraints for different problem instances. 20

4.1 Heuristic validation on small test cases. 28

5.1 Numerical parameters of the simulator, grouped by functionality. . . 41

6.1 Parameters held constant across all configurations. 46
6.2 Test configurations for the computational experiments. 47
6.3 Throughput comparison across test instances 49
6.4 Analysis of variance on the per-seed difference $\mathcal{D}$ (optimizer minus greedy policies), with the corresponding effect sizes (η^2). 51
6.5 Mean flow time comparison across test instances 53
6.6 Pod movement comparison across test instances 55

B.1 Greedy policies: per-replication throughput 66
B.2 Optimizer: per-replication throughput 66
B.3 Greedy policies: replication sizing. 67
B.4 Optimizer: replication sizing. 68
B.5 Greedy policies: per-replication average flow time 70
B.6 Optimizer: per-replication average flow time 70
B.7 Greedy policies: mean number of closed orders by order size 71
B.8 Optimizer: mean number of closed orders by order size 71
B.9 Greedy policies: per-replication average number of pods moving simultaneously 72
B.10 Optimizer: per-replication average number of pods moving simultaneously 72

List of Figures

3.1 Toy example of the warehouse layout 10
3.2 Example of a time-space network 13

5.1 Event flow of the simulation model 34
5.2 Empirical distribution of the number of order lines 42
5.3 Empirical SKU request-frequency distribution 43
5.4 SKU distribution across pods 44

6.1 Throughput comparison across test instances 48
6.2 Mean flow time comparison across test instances 52
6.3 Pod movement comparison across test instances 54

B.1 Diagnostic plots for the analysis of variance on the per-seed difference $\mathcal{D}$. 69

Chapter 1

Introduction

E-commerce has grown rapidly in recent years, increasing the pressure on warehouses to process a larger number of orders while keeping fulfilment times short. Traditional systems, where workers travel to storage locations to retrieve each item, can struggle to meet these requirements. This has contributed to the adoption of automated systems that bring inventory to the worker, among which Robotic Mobile Fulfillment Systems (RMFSs) are one of the most established solutions. The concept traces back to Kiva Systems, later acquired by Amazon and renamed Amazon Robotics in 2012, and similar systems have since been developed by other companies [2].

In an RMFS, a fleet of autonomous mobile robots (AMRs) carries movable shelves, usually called pods, back and forth between the storage area and a set of fixed picking stations, where human operators pick the items a shipment needs. What makes such a system interesting from an optimisation point of view is everything that has to be decided to make it operate efficiently. At one end there is the design of the warehouse: where to place the picking stations, how many AMRs to deploy, and how to lay out the storage area so that the system performs efficiently once it is built. Once the warehouse exists, orders keep arriving and have to be served efficiently, which means deciding which pod should fulfill which order, which station should handle it, and in what sequence, all while the system remains responsive to new orders as they arrive. Then, at the lowest level, there are the robots themselves, and the theme of routing them without collisions and managing when each one should recharge without leaving the system short of capacity.

This thesis is concerned with the middle layer, also called the *tactical level*, the one that decides how orders are served once the warehouse is operational. An RMFS is particularly well suited to decisions taken online: a new order can, in principle, be inserted into the plan the moment it arrives, and which pods to retrieve or how to group orders together can be reconsidered just as often. This flexibility raises a natural question: is it better to make the most of it through reactive rules that respond to the current state of the system, or to wait and make

decisions based on a more global view of the system?

To address this question, this thesis focuses on a simplified version of the tactical layer, in which the focus is on the design of picking tasks required to fulfill customer orders. Two strategies, each offering a different degree of lookahead, are compared. The first relies on reactive policies that make decisions online based solely on local information. The second uses a mathematical formulation to plan over a batch of orders at a time, allowing decisions to be made from a broader view of the system. The two strategies are evaluated through a discrete-event simulator, designed and implemented from scratch as part of this work, which reproduces the warehouse under a continuous stream of orders and measures the resulting system performance.

This thesis is organized as follows. Chapter 2 reviews the existing literature on tactical decision-making in RMFSs, with particular attention to the problem formulation and solution methodologies relevant to this work. Chapters 3 and 4 formalize the problem addressed in this thesis, first through a two-stage mathematical decomposition and then through a heuristic method designed to solve it. Chapter 5 introduces the evaluation framework developed as part of this work, describing both the simulation engine and the greedy policies used for comparison. Finally, Chapter 6 presents and discusses the computational results obtained through the simulator.

Chapter 2

Literature Review

This thesis is framed within a parts-to-picker robotic mobile fulfillment system (RMFS). Such systems are distinguished from picker-to-parts ones by which party moves to fulfill an order: in a picker-to-parts system the picker travels to the required storage locations, whereas in a parts-to-picker system the storage locations themselves, in the form of movable shelves, are brought to a stationary picker, a distinction drawn by Boysen and de Koster [3] in their review of the field. Autonomous mobile robots (AMRs) carry out this transport, moving the movable shelves, hereafter *pods*, between the storage area and the picking stations where human operators assemble customer orders.

As already mentioned, decisions in such a system are usually organised across three planning levels:

- **Strategic level**: warehouse layout design and fleet sizing.
- **Tactical level**: assignment and coordination of orders, pods, and workstations over an operating period.
- **Operational level**: AMR routing and traffic management, decided in real time as AMRs move on the warehouse floor.

The present work is situated at the tactical level. Section 2.1 examines which decisions belong to this level, and how consistently the literature defines them; Section 2.2 then surveys how those decisions are solved.

2.1 Decision Sub-Problems in RMFS Operations

Core Decisions Three decisions recur in almost every tactical-level study reviewed here:

1. **Order-to-workstation assignment** determines which orders are handled at which picking station.

2. **Order-to-pod assignment** determines which pods are required to fulfill those orders.
3. **Pod sequencing** determines the order in which the pods assigned to a workstation are presented to the picker.

The set of decisions is largely shared across the literature; the terminology is not. Merschformann et al. [1] and Jiao et al. [4], for instance, refer to order-to-workstation assignment as *pick order assignment* (or *replenishment order assignment*, when replenishment is involved), and to order-to-pod assignment as *pick pod selection*. To avoid misrepresenting individual contributions, this chapter adopts each author's own terminology when discussing their work and reserves the generic terms above for sections that compare across studies.

Extended Decisions A smaller group of studies extends this core set. Some add **robot-to-pod assignment**, the dispatching of specific robots to specific pod-moving tasks, as in the dynamic stochastic framework of Rimélé et al. [2], where every decision, whether a storage, retrieval, or opportunistic task, is explicitly tied to the robot that becomes available; others incorporate **replenishment missions** alongside the picking ones.[1]

Table 2.1: Decision sub-problems addressed in the reviewed literature

Citation	OW	OP	RP	Seq.	Repl.	Objective
Allgor et al. [5]	✓	✓				Pile-on maximisation
Valle and Beasley [6]	✓	✓		✓		Rack usage minimisation
Zhen et al. [7]	✓	✓	✓	✓		Makespan minimisation
Barnhart et al. [8]	✓	✓		✓		Throughput maximisation
Merschformann et al. [1]	✓	✓			✓	Rule-based control
Wu et al. [9]	✓	✓	✓	✓	✓	Makespan minimisation
Zhuang et al. [10]	✓	✓		✓		Pod movement minimisation
Jiao et al. [4]	✓	✓				Pile-on maximisation

OW = order-to-workstation assignment; OP = order-to-pod assignment; RP = robot-to-pod assignment; Seq. = pod sequencing; Repl. = replenishment-related decisions.

Summary of Coverage Table 2.1 summarises which decisions each paper models and which objective it pursues. The three core decisions are almost always present; the studies diverge in the additional decisions they include and, more significantly,

[1] Replenishment missions restock pods with new inventory and are the counterpart of picking missions, which retrieve items to fulfill customer orders. This thesis is concerned exclusively with the latter.

in the objective they optimise. Some formulations maximise throughput or the pile-on ratio, the number of items picked per pod visit, while others minimise makespan or total travel distance. This choice typically reflects the practical priority behind the study, whether system-level efficiency or preventing pickers from idling while awaiting the next pod.

2.2 Solution Approaches

Agreement on which sub-problems constitute the tactical level does not translate into agreement on how to solve them, and here the literature divides into two families. The first treats the sub-problems as *separable*: each is solved on information handed down from the previous decision, and no later decision can revise an earlier one. The second keeps them *integrated*, through a metaheuristic or, in one paper, an exact algorithm applied to a deliberately narrowed version of the problem.

2.2.1 Sequential Decomposition Approaches

The sequential family spans a spectrum from tightly coupled phase decompositions to sets of fully independent rules.

Allgor et al. [5] study a continuous-arrival setting in which order shipments accumulate into a *pick backlog* over time. They separate the problem into two modules solved in sequence: a *picking module* decides which backlog shipments enter the *pick window* and which pods they require, maximising *pile-on* (the number of units picked per pod trip), after which an *assignment module* sends those already-selected pods to stations so as to minimise the AMR travel distance per pod trip, with no mechanism for revising the picking module's decisions. The picking module alone is a mixed-integer program too computationally demanding to solve directly, and it is therefore decomposed further into four sequential phases: feasibility checking, pod selection, provisional scheduling, and final scheduling. Each of them narrows the search space passed to the next.

Similarly, Valle and Beasley [6] couple two of the decisions and treat the third separately. Order-to-workstation and order-to-pod assignment are formulated jointly as a single *order and rack allocation* program, solved by a matheuristic built directly on that formulation, while rack sequencing is handled independently for each picker once its orders and racks are fixed. The authors justify the split by observing that, once an allocation is set, every assigned rack must eventually reach the picker regardless of order, so picking time does not depend on the sequence chosen. Sequencing thus reduces to a *feasibility problem*: finding a presentation order that completes every assigned order within a limited number of bin positions, falling back on *rack revisits* when no single-visit sequence is feasible.

At the opposite, Merschformann et al. [1] forgo optimisation across decisions altogether. They fix five decisions: pick order assignment, replenishment order assignment, pick pod selection, replenishment pod selection, and pod storage assignment.[2] For each they define a *rule-based control policy*, and the policies are evaluated independently through *discrete-event simulation*. The simulation environment, *RAWSim-O*, is an open-source RMFS framework introduced in earlier work by the same lead author [11] and subsequently adopted by others, including Jiao et al. [4].

2.2.2 Integrated Approaches

The integrated family can revisit decisions, but this comes at the cost of tractability. The studies below differ mainly in how they address this trade-off.

The most direct route keeps a single integrated model and makes it tractable by solving it in pieces. Zhen et al. [7] cast the tactical problem as one mixed-integer program and solve it with a *Two-Layer Revolving Algorithm* (TRA): blocks of variables are fixed in a revolving manner, the rest re-solved, and the frozen block then rotated, so that a choice made while one block was fixed can be reconsidered once another is released. On the same instances, they show this revolving procedure outperforming a purely sequential three-stage baseline, providing the clearest direct evidence that integration is worthwhile. Wu et al. [9] push this idea to its limit. Having shown several of their five sub-problems to be NP-hard, they abandon alternation altogether and let a *Variable Neighbourhood Search* (VNS) explore moves across all of them simultaneously, collapsing the tactical problem into a single search space rather than a rotation of smaller ones.

Jiao et al. [4] keep the integrated problem tractable in a different way, replacing a general-purpose search with a fast, ad-hoc rule-based heuristic. It decides pick order assignment and pick pod selection jointly, in an alternate, order-driven framework, proceeding station by station and alternating between covering backlog orders with the pods already present and bringing in a new pod only when none can be fully covered. Because each pass is cheap, the program can be re-solved whenever a workstation's spare capacity crosses a predefined threshold, so the allocation adapts continuously to the state of the system rather than being computed once for a static backlog. It is this rolling, state-triggered logic that the present thesis adopts, here through a periodic rather than event-triggered re-optimization.

The remaining studies retain a conventional metaheuristic but refine its search. Zhuang et al. [10] address the *multi-workstation* variant of the problem, combining an *Adaptive Large Neighbourhood Search* (ALNS) with a data-driven heuristic and

[2] Pod storage assignment decides where a pod is returned in the storage area once it leaves a workstation. It plays no further role in this thesis and is mentioned here only for completeness.

simulated annealing to build the matching order and pod sequences, all directed at minimising total pod movement. Distinctively, they extend the setting beyond a single picker by explicitly modelling *workload balancing* across workstations and *rack conflicts*, the contention that arises when two workstations require the same pod simultaneously. Barnhart et al. [8] instead introduce a learning component: an offline model predicts which neighbourhoods are most promising, and the search is steered accordingly. Their formulation also relies on a time-space network to approximate warehouse congestion. This thesis builds on the same device, though here it enforces the temporal feasibility of pod movements rather than serving as a congestion proxy.

Exactness is achieved only by narrowing the scope. Justkowiak et al. [12] solve single-workstation *order scheduling and rack sequencing* to optimality by dynamic programming, minimising the number of *rack visits* and made tractable by a *set-covering*-based pruning rule. Yet it is precisely this restriction to a single workstation that prevents the method from scaling to the multi-workstation, multi-pod setting that Zhen et al. [7], Wu et al. [9], Jiao et al. [4], and Zhuang et al. [10] are designed to handle.

Two points stand out from this review. The first is agreement: order-to-workstation assignment, order-to-pod assignment, and pod sequencing recur throughout as the tactical core, regardless of terminology. The second is a trade-off. Integrating these decisions tends to improve solution quality, as shown directly by Zhen et al. [7], but not without cost: the price is a harder and more expensive problem to solve. This trade-off motivates the sequential decomposition adopted in this thesis, which accepts some loss in solution quality in exchange for a problem that remains tractable at the scale considered here.

Chapter 3

Problem Formulation

This chapter formally defines the decision problem addressed in this thesis. Building on the tactical sub-problems discussed in Chapter 2, the tactical planning problem is decomposed into two sequential sub-problems: an order-to-workstation and order-to-pod assignment problem, and the subsequent task-scheduling problem, which defines the picking schedule.

The remainder of this chapter is organized as follows. Section 3.1 provides an overview of the system under study. Section 3.2 introduces the time-space network representation used to capture the temporal dimension of pod movements. Section 3.3 then formally states the mathematical model, separately addressing the order-to-workstation and order-to-pod assignment problem (Section 3.3.1) and the scheduling problem (Section 3.3.2). Section 3.4 closes with an analysis of the dimensionality of the resulting formulations.

3.1 Problem Overview

This work addresses decisions at the tactical level, so the system layout is taken as given, with the following information known in advance:

- **Warehouse layout**: the number of pods and workstations and their positioning, and the size of the fleet operating in the system.
- **Pod information**: for each pod, its storage location and the set of stock-keeping units (SKUs) it contains.

Warehouse Layout The warehouse layout is defined on a discrete grid, as shown in Figure 3.1. The grid is generated from two input parameters, the number of pod rows and the number of pod columns, whose product determines the total number of pods in the system. Pods are arranged in columns, with an aisle left free

between consecutive columns to allow AMR passage, and workstations are placed along the perimeter of the resulting grid.

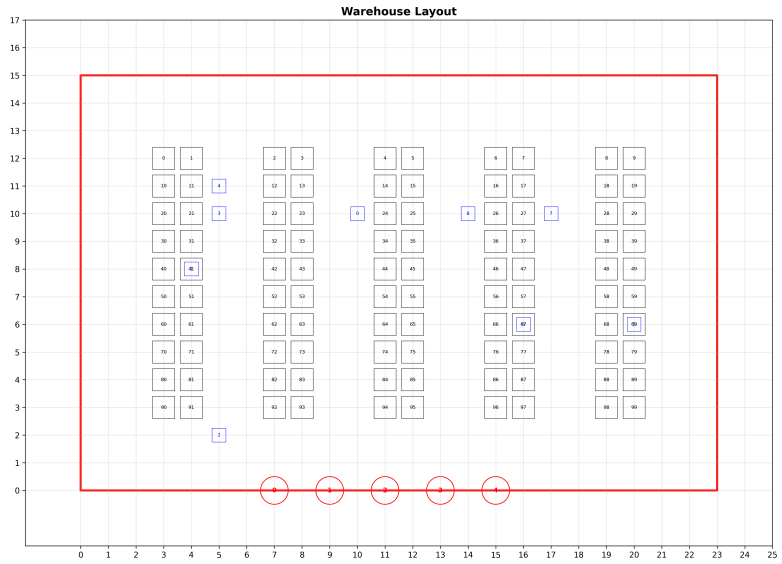


Figure 3.1: Example of the layout of a small warehouse consisting of 100 pods (black squares), 15 AMRs (blue squares), and 9 workstations (red circles).

Each cell is associated with a unique identifier defined by a bijective mapping of its spatial coordinates:

$$cell_id = x + X \cdot y$$

where X is the horizontal dimension of the grid. This indexing scheme identifies every position in the warehouse uniquely, and is later used to define the nodes of the time-space network.

Pod and SKU Distribution Since this work considers only picking operations and not replenishment, the units of every SKU in a pod are treated as effectively unlimited. Otherwise, a pod could run out of stock and would need a replenishment tasks to be restocked, which lies outside the scope of this thesis. As is standard in the literature, each SKU can be stored in more than one pod for efficiency reasons, which makes order-to-pod assignment a genuine decision: a requested SKU must be retrieved from one of the candidate pods.

Since some SKUs are requested more often than others, it is natural for them to also be stored in more pods. Pod content is therefore generated so that the more popular SKUs appear in a larger number of pods. The specific algorithm and parameters are reported in Chapter 5, together with the other input parameters used in the simulation experiments.

Order Generation In reality orders arrive continuously; here the stream is approximated with constant inter-arrival times. As they arrive, orders accumulate in a backlog and wait there until they are processed. Each order is defined by the set of SKUs it requires and is generated in two steps. First, the number of distinct SKUs it contains, or equivalently its *order lines*, is drawn at random: most orders are small, and larger ones become progressively less likely, as is typical of e-commerce demand. This number fixed, the SKUs are then drawn according to the same popularity pattern that governs their distribution across the pods.

Because the backlog may exceed the fulfillment capacity of a single decision epoch, the optimization does not act on it all at once. At each epoch, a fixed number of orders is drawn from the head of the backlog, according to a first-in-first-out (FIFO) rule.

Decision Structure Once this subset is selected, three decisions must be made. Two are assignments and one is a scheduling decision, and together they define and organise the tasks to be carried out.

Each order needs a station at which its items will be picked. This is the role of the order-to-workstation assignment, which distributes the orders across the workstations. Once an order is opened at a workstation it cannot be reassigned, which becomes a hard constraint for orders opened in earlier decision epochs and not yet closed. An order must also be paired with pods that can fulfill it: this is the role of the order-to-pod assignment, which links each order to a set of pods jointly holding all the SKUs it requires. Once both assignments are made the tasks are fixed, since it is now known which pods must move and to which workstation.

What remains is to arrange these tasks over time. Scheduling, the third decision, sets for each workstation the sequence in which the assigned pods visit it and the assigned orders are opened. The temporal dimension enters at this stage, bringing with it an operational limit: each workstation can serve only a bounded number of orders at any one time. Orders carried over from earlier epochs are treated as already open at the start of the horizon.

At this stage, the hard constraints also concern the pod movements, as the plan has to stay consistent in both space and time. A pod occupies a single location at any instant, its movements are constrained by travel times, and it cannot serve two workstations at once; moreover, the number of pods moving simultaneously is bounded by the number of available AMRs. To capture these requirements, the scheduling problem is built on a time-space formulation, inspired by Barnhart et al. [8] and detailed in the next section, which encodes pod positions and movements over a discretised horizon and writes the consistency requirements directly into the model.

These decisions are not solved jointly but sequentially. The two assignments are handled together as a single problem, whose output feeds the task-scheduling

problem detailed in Section 3.3. The whole process runs within a rolling-horizon framework, recomputed every 15 minutes from the current system state and the newly selected orders, carrying over those left open at the workstations by the previous epoch.

3.2 Time-Space Network Representation

A time-space network is defined over a planning horizon discretized into a set $\mathcal{T}$ of time periods, each of duration Δ. Each node is identified by a spatial component $l \in \mathcal{W} \cup \mathcal{L}$, where $\mathcal{W}$ denotes the set of workstation locations and $\mathcal{L}$ the set of pod storage locations, together with a temporal component $t \in \mathcal{T}$.

The set of arcs is partitioned into two subsets:

- **Travel arcs**: arcs of the form $\Big((l_1, t_1), (l_2, t_2)\Big)$, where $l_1 \neq l_2$ and $t_2 - t_1$ equals the travel time between locations l_1 and l_2, rounded up to the nearest time period. These arcs represent the movement of pods between different locations.
- **Idle arcs**: arcs of the form $\Big((l, t_1), (l, t_2)\Big)$, where $t_2 = t_1 + 1$. These arcs model a pod remaining at the same location for one time period.

More formally, the time-space network is represented as a directed graph $G^{TSN} = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ and $\mathcal{A}$ denote the sets of nodes and arcs, respectively. Nodes represent the possible locations of a pod at a specific time period, while arcs describe the feasible transitions between these states.

A pod trajectory is obtained by selecting a sequence of temporally consecutive arcs that connects its initial position to the locations required by the picking schedule. Each pod starts from its storage location at the beginning of the planning horizon and follows a path that includes the necessary workstation visits. The path reaches the final time period $|\mathcal{T}|$ at a node $(l, |\mathcal{T}|)$, where l denotes the pod location at the end of the horizon. Although not explicitly imposed by the mathematical formulation, the implementation attempts to return each pod to its storage location at the end of the planning horizon whenever feasible. This is an implementation choice aimed at keeping the resulting trajectories consistent with the natural resting state of the system.

The time-space network representation enables the coordination of pod movements and their integration with picking operations at the workstations. Figure 3.2 illustrates a simplified example, adapted from Barnhart et al. [8], in which the system consists of five pods, each associated with a specific storage location and a set of SKUs, and one workstation where two customer orders must be fulfilled. The network tracks the evolution of pod positions over the planning horizon, identifying the time periods in which pods arrive at the workstation and become available for picking operations.

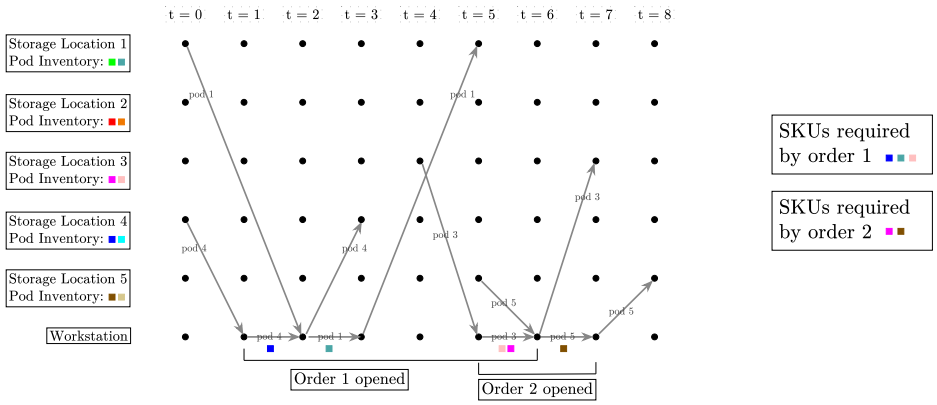


Figure 3.2: Example of a time-space network representing pod movements and order processing over the planning horizon. Each node corresponds to a location-time pair, while travel and idle arcs describe the pod movements. Colored squares represent the SKUs required by each order.
Figure adapted from Barnhart et al. [8].

The information provided by the network is exploited in the mathematical model to represent the temporal and spatial feasibility of pod movements. In particular, its structure allows the model to account for pod availability at workstations, transportation times, and the limited number of pods that can be simultaneously involved in transportation activities, given the available AMR fleet size. The corresponding constraints are introduced in the following sections.

3.3 Mathematical Modelling

The mathematical model consists of two optimization problems solved in sequence: once the first is solved, its decision variables are fixed and passed as input to the second.

The first problem sets the order-to-workstation and order-to-pod assignments, minimizing the number of pods to be moved, or equivalently maximizing the pile-on, while penalizing travel distance. Its output is a set of tasks, one per pod to be moved. The second problem schedules these tasks, fixing for each pod the picking operations it must perform and their sequence, so as to maximize throughput, i.e. the number of picking operations completed within the horizon, while penalizing order flow times to avoid leaving orders in the backlog indefinitely.

Table 3.1 introduces the notation common to the two problems.

Table 3.1: Notation used in the mathematical formulation

Symbol	Definition
Sets	
$\mathcal{O}$	Set of orders in the current decision epoch
$\mathcal{O}_{fix}$	Set of orders already open at a workstation ($\mathcal{O}_{fix} \subset \mathcal{O}$)
$\mathcal{O}(w)$	Set of orders in the current decision epoch assigned to workstation w
$\mathcal{I}$	Set of SKUs
$\mathcal{I}_o$	Set of SKUs required by order $o \in \mathcal{O}$
$\mathcal{P}$	Set of pods
$\mathcal{P}_i$	Set of pods containing SKU $i \in \mathcal{I}$
$\mathcal{W}$	Set of workstations
$\mathcal{L}$	Set of pods' storage locations
$\mathcal{T}$	Set of time steps considered in the current decision epoch
Parameters	
C_w	Capacity of each workstation
R	Number of AMRs available in the system
δ^{SKU}	Processing time of picking a SKU
δ^{pod}	Processing time of moving a pod to a workstation
Time-space network	
$\mathcal{N}$	Set of nodes of the time-space network
$\mathcal{N}_{end}$	Set of terminal nodes, i.e. in the form $(l, \lvert\mathcal{T}\rvert)$ where $l \in \mathcal{L} \cup \mathcal{W}$
$\mathcal{N}_p^0$	Initial node for pod $p \in \mathcal{P}$, i.e., its storage location at time 0
$\mathcal{A}$	Set of arcs of the time-space network
$\mathcal{A}^+_{N(w,t)}$	Set of outgoing arcs of the node with spatial component w and temporal component t
$\mathcal{A}^-_{N(w,t)}$	Set of incoming arcs of the node with spatial component w and temporal component t
$\mathcal{A}_p^{\text{away}}(t)$	Set of arcs used by pod p that are away from its storage location s_p at time t: $\mathcal{A}_p^{\text{away}}(t) = \{a = (src, dst) \in \mathcal{A} : \text{loc}(src) \neq s_p,\ t(src) \leq t < t(dst)\}$

3.3.1 Stage 1: Assignment Problem

The decision variables, encoding the assignments already outlined, are defined as follows:

$$z_{ow} = \begin{cases} 1 & \text{if order } o \in \mathcal{O} \text{ is assigned to workstation } w \in \mathcal{W}, \\ 0 & \text{otherwise,} \end{cases}$$

$$x_{iop} = \begin{cases} 1 & \text{if SKU } i \in \mathcal{I}_o \text{ of order } o \in \mathcal{O} \text{ is retrieved from pod } p \in \mathcal{P}_i, \\ 0 & \text{otherwise,} \end{cases}$$

$$y_{wp} = \begin{cases} 1 & \text{if pod } p \in \mathcal{P} \text{ visits workstation } w \in \mathcal{W}, \\ 0 & \text{otherwise.} \end{cases}$$

Here z_{ow} and x_{iop} are the genuine assignment decisions, while y_{wp} is an auxiliary variable, since its value is implicitly determined by the assignment of orders to workstations and by the selection of pods for SKU retrieval.

The objective of this stage, to be minimized, combines two terms concerning the pods to be moved: the first counts the pod-to-workstation visits, whose minimization drives up the pile-on, while the second penalizes the distance those pods travel. The optimization problem is formulated as

$$\min \quad \sum_{w \in \mathcal{W}} \sum_{p \in \mathcal{P}} y_{wp} + \alpha \sum_{w \in \mathcal{W}} \sum_{p \in \mathcal{P}} y_{wp} \cdot dist(w, p) \tag{3.1}$$

s.t. **Assignment constraints**

$$\sum_{w \in \mathcal{W}} z_{ow} = 1 \quad \forall o \in \mathcal{O}, \tag{3.2}$$

$$\sum_{p \in \mathcal{P}_i} x_{iop} = 1 \quad \forall o \in \mathcal{O},\ i \in \mathcal{I}_o, \tag{3.3}$$

$$z_{o,\bar{w}_o} = 1 \quad \forall o \in \mathcal{O}_{fix}, \tag{3.4}$$

Consistency and workload constraints

$$y_{wp} \geq x_{iop} + z_{ow} - 1 \quad \forall o \in \mathcal{O},\ i \in \mathcal{I}_o,\ w \in \mathcal{W},\ p \in \mathcal{P}_i, \tag{3.5}$$

$$L^{\min} \leq \sum_{o \in \mathcal{O}} d_o z_{ow} \leq L^{\max} \quad \forall w \in \mathcal{W}, \tag{3.6}$$

Domain of definition$x,\ y,\ z$ **binary**.

Constraints (3.2) and (3.3) assign each order to exactly one workstation and each required SKU to exactly one pod, respectively. Constraint (3.4) enforces the initial

conditions for orders already in progress, keeping them assigned to their current workstation $\bar{w}_o$. Constraint (3.5) links the three decision variables, forcing $y_{wp} = 1$ whenever an order assigned to w retrieves one of its SKUs from p. Constraint (3.6) balances the workload across workstations, with lower and upper bounds

$$L^{\min} = \left\lfloor 0.8 \, \frac{\sum_{o \in \mathcal{O}} d_o}{|\mathcal{W}|} \right\rfloor, \qquad L^{\max} = \left\lceil 1.2 \, \frac{\sum_{o \in \mathcal{O}} d_o}{|\mathcal{W}|} \right\rceil,$$

where $d_o = |\mathcal{I}_o|$ is the number of SKUs required by order o.

Constraint (3.6) is not an operational requirement of the system, but is introduced to encourage a balanced workload across workstations without adding a further term to the objective. It is also the only constraint that can make the problem infeasible, since too tight a band may leave no way to distribute the orders. The tolerances 0.8 and 1.2 around the average load $\left(\sum_{o \in \mathcal{O}} d_o\right)/|\mathcal{W}|$ were tuned empirically to keep the band tight enough to enforce a balanced workload, yet wide enough for a feasible assignment to exist across the instances considered.

The weight α in the objective (3.1) tunes the trade-off between the two terms, the number of visits and the travel distance. Reducing visits is the primary goal, and the distance term only discourages retrieving a pod from far away when closer ones would serve the same SKUs. With a per-visit cost $1 + \alpha \, dist$, α is calibrated on the distance span $D = \max_{w,p} dist(w,p) - \min_{w,p} dist(w,p)$ as

$$\alpha = \frac{k^\star - 1.5}{D}, \qquad k^\star = 3,$$

so that sharing a pod at the largest possible extra distance still pays off once it saves at least two visits, while at smaller extra distances it is preferred earlier. Normalising by D keeps this balance independent of the layout scale.

3.3.2 Stage 2: Task-Scheduling Problem

The second-stage optimization problem determines the picking schedule to be followed by the operational system. In particular, this stage defines the sequence in which pods should visit the previously assigned workstations and, consequently, the order in which orders should be processed at each workstation. The formulation does not explicitly account for the stochasticity affecting pod routing, as it is used to determine a reference schedule for the operational system.

The decision variables introduced at this stage are:

$$x_{oit} = \begin{cases} 1 & \text{if item } i \in \mathcal{I}_o \text{ of order } o \in \mathcal{O} \\ & \text{has been picked by time } t \in \mathcal{T} \text{ at the assigned workstation,} \\ 0 & \text{otherwise,} \end{cases}$$

$$f_{ot} = \begin{cases} 1 & \text{if at least one SKU of order } o \in \mathcal{O} \\ & \text{has been delivered by time } t \in \mathcal{T} \text{ at the assigned workstation,} \\ 0 & \text{otherwise,} \end{cases}$$

$$g_{ot} = \begin{cases} 1 & \text{if all SKUs of order } o \in \mathcal{O} \\ & \text{have been delivered by time } t-1 \in \mathcal{T} \text{ at the assigned workstation,} \\ 0 & \text{otherwise,} \end{cases}$$

$$v_{ot} = \begin{cases} 1 & \text{if order } o \in \mathcal{O} \text{ is open at the assigned workstation at time } t \in \mathcal{T}, \\ 0 & \text{otherwise,} \end{cases}$$

$$y_{pa} = \begin{cases} 1 & \text{if pod } p \in \mathcal{P} \text{ is assigned to time-space network arc } a \in \mathcal{A}, \\ 0 & \text{otherwise.} \end{cases}$$

The decision variable x_{oit} defines the picking schedule in cumulative form. Based on it, the auxiliary variables f_{ot} and g_{ot} identify the beginning and the end of the processing interval of each order at the workstation. As cumulative variables, x_{oit}, f_{ot}, and g_{ot} follow an increasing binary pattern of the form $(0,\dots,0,1,\dots,1)$. The difference between f_{ot} and g_{ot} allows the model to identify the time interval during which an order is active at the workstation, represented by the non-cumulative variable v_{ot}. A temporal shift is introduced in the definition of g_{ot} to ensure that v_{ot} contains at least one value equal to 1 for orders composed of a single SKU. Without this shift, f_{ot} and g_{ot} would coincide for such orders, resulting in an empty processing interval and preventing the model from accounting for the workstation occupation of the order. Lastly, the variable y_{pa} determines the path followed by each pod within the time-space network.

The objective function maximizes the system throughput, i.e., the number of SKUs retrieved by the end of the planning horizon, while penalizing longer completion times. In the application, the set of orders $\mathcal{O}$ considered by the model is only a subset of the full backlog, which may be too large to be processed in a single decision epoch. The penalty therefore discourages the continuous postponement of less convenient orders, preventing them from remaining unprocessed in the backlog indefinitely.

$$\max \quad \sum_{o\in\mathcal{O}} \sum_{i\in\mathcal{I}_o} x_{oi,T} - \frac{0.1}{|\mathcal{T}|} \sum_{o\in\mathcal{O}} \sum_{t\in\mathcal{T}} \beta_o(t)(1-g_{ot}) \tag{3.7}$$

s.t. **Workstation constraints**

$$\sum_{o\in\mathcal{O}(w)} v_{ot} \le C_w \quad \forall w \in \mathcal{W},\ t \in \mathcal{T}, \tag{3.8}$$

$$\sum_{o\in\mathcal{O}(w)}\sum_{i\in\mathcal{I}_o}\delta^{SKU}(x_{oit}-x_{oi,t-1})+\sum_{p\in\mathcal{P}}\sum_{a\in\mathcal{A}^-_{N(w,t)}}\delta^{pod}y_{pa}\leq\Delta\quad\forall w\in\mathcal{W},\ t>0, \tag{3.9}$$

Consistency constraints

$$x_{oit}\geq x_{oi,t-1}\quad\forall o\in\mathcal{O},\ i\in\mathcal{I}_o,\ t>0, \tag{3.10}$$

$$x_{oit}-x_{oi,t-1}\leq v_{ot}\quad\forall o\in\mathcal{O},\ i\in\mathcal{I}_o,\ t>0, \tag{3.11}$$

$$v_{ot}=f_{ot}-g_{ot}\quad\forall o\in\mathcal{O},\ t\in\mathcal{T}, \tag{3.12}$$

$$f_{ot}\geq x_{oit}\quad\forall o\in\mathcal{O},\ i\in\mathcal{I}_o,\ t\in\mathcal{T}, \tag{3.13}$$

$$g_{ot}\leq x_{oi,t-1}\quad\forall o\in\mathcal{O},\ i\in\mathcal{I}_o,\ t>0, \tag{3.14}$$

$$f_{ot}\geq f_{o,t-1}\quad\forall o\in\mathcal{O},\ t>0, \tag{3.15}$$

$$g_{ot}\geq g_{o,t-1}\quad\forall o\in\mathcal{O},\ t>0, \tag{3.16}$$

$$f_{ot}\leq\sum_{i\in\mathcal{I}_o}x_{oit}\quad\forall o\in\mathcal{O}\setminus\mathcal{O}_{fix},\ t\in\mathcal{T}, \tag{3.17}$$

$$g_{ot}\geq\sum_{i\in\mathcal{I}_o}x_{oi,t-1}-(|\mathcal{I}_o|-1)\quad\forall o\in\mathcal{O},\ t>0, \tag{3.18}$$

$$v_{o,0}=1\quad\forall o\in\mathcal{O}_{fix}, \tag{3.19}$$

$$v_{ot}\geq v_{o,t-1}-g_{ot}\quad\forall o\in\mathcal{O},\ t>0, \tag{3.20}$$

Pod operations and fleet sizing

$$\sum_{a\in\mathcal{A}^+_{\mathcal{N}^0_p}}y_{pa}=1\quad\forall p\in\mathcal{P}, \tag{3.21}$$

$$\sum_{a\in\mathcal{A}^-_n}y_{pa}-\sum_{a\in\mathcal{A}^+_n}y_{pa}=0\quad\forall p\in\mathcal{P},\ n\in\mathcal{N}\setminus(\mathcal{N}^0_p\cup\mathcal{N}_{end}), \tag{3.22}$$

$$x_{oit}-x_{oi,t-1}\leq\sum_{a\in\mathcal{A}^-_{N(w(o),t)}}y_{p(i)a}\quad\forall o\in\mathcal{O},\ i\in\mathcal{I}_o,\ t>0, \tag{3.23}$$

$$\sum_{p\in\mathcal{P}}\sum_{a\in\mathcal{A}^{\text{away}}_p(t)}y_{pa}\leq R\quad\forall t\in\mathcal{T}, \tag{3.24}$$

Domain of definition$x,\ f,\ g,\ v,\ y$ **binary**.

Constraints 3.8 and 3.9 regulate the workstation capacity. Constraint 3.8 limits the maximum number of simultaneously open orders at each workstation, while Constraint 3.9 enforces a temporal capacity constraint, requiring the total processing

time of the picking operations scheduled in each time slot not to exceed the available processing time.

Constraints 3.10, 3.15, and 3.16 enforce the cumulative structure of the decision variables. Based on these variables, Constraints 3.11–3.14 define the order processing logic. In particular, Constraint 3.11 allows SKU picking only when the corresponding order has been opened at its assigned workstation, Constraint 3.12 defines the activity variable v_{ot}, Constraint 3.13 forces f_{ot} to switch to 1 after the first SKU of an order is picked, whereas Constraint 3.14 keeps g_{ot} equal to 0 until all the required SKUs have been picked. Constraint 3.18 completes the definition of g_{ot} by forcing it to switch to 1 only after the last required SKU has been retrieved.

Constraints 3.20, 3.19, and 3.17 handle the initial conditions. Constraint 3.17 defines the standard behaviour of f_{ot} for newly released orders, while Constraints 3.20 and 3.19 account for orders already under processing at the beginning of the planning horizon. For these orders, f_{ot} may still be equal to 0 until the first SKU is picked within the considered horizon, and Constraint 3.20 therefore ensures that v_{ot} remains equal to 1 until g_{ot} switches to 1, correctly representing the workstation occupation.

Constraints 3.21–3.23 define the pod movements within the time-space network. Constraint 3.21 requires each pod to start from its storage location, Constraint 3.22 enforces flow conservation in the network, and Constraint 3.23 links the picking decisions to the pod trajectories, allowing an SKU to be picked only if the assigned pod has reached the workstation at the beginning of the corresponding time slot. Finally, Constraint 3.24 imposes the fleet-size limitation.

The objective function (3.7) combines throughput maximization with a penalty on order completion time. The order-dependent penalty term $\beta_o(t)$ is defined as

$$\beta_o(t) = \frac{\tau + t\Delta - a_o}{\Delta} \quad \forall o \in \mathcal{O},\ t \in \mathcal{T},$$

where τ denotes the external time at the beginning of the planning horizon, a_o is the arrival time of order o in the external time scale, and $t\Delta$ converts the discrete time index into the corresponding external time unit. The term $(1 - g_{ot})$ in (3.7) activates the penalty only for the orders that have not yet been completed at time t, so that an order accumulates penalty at a rate growing with the time it has already spent in the system.

3.4 Size and Complexity

The optimization problems formulated in the previous sections may become computationally challenging due to their combinatorial nature, especially for larger instances and, in particular, for the second-stage problem.

Two reduction strategies are therefore adopted. First, in the second stage the set of pods considered is restricted to the subset selected in the first stage, and a correspondingly reduced time-space network is built, so that no variable or constraint is generated for pods that have no role in the task design problem. Second, the variables involving both orders and SKUs are reformulated by collapsing the two indices into a single order-item index, through the bijective mapping

$$(o, i) \longmapsto im \qquad \forall o \in \mathcal{O},\ i \in \mathcal{I}_o.$$

The transformation reduces the number of indices in the formulation without changing the correspondence between an order-item pair and its decision variables.

Table 3.2 summarized the dimensionality of the two optimization stages by showing the number of decisional variables and constraints. Only the implemented size is reported, since computing the theoretical size over all possible combinations of the formulation sets would substantially overestimate the actual model dimension. The reported values are those returned by the code and are therefore instance-dependent: the number of pods considered in the second stage follows from the first-stage solution, which is itself affected by the stochasticity of the order-generation algorithm. The table is thus intended to characterize the order of magnitude of the resulting optimization problems.

Table 3.2: Number of decision variables and constraints for different problem instances.

Instance parameters				**Stage 1**		**Stage 2**	
$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{W}\rvert$	$\lvert\mathcal{I}\rvert$	Variables	Constraints	Variables	Constraints
100	300	4	3000	48'700	188'665	8'710'220	368'020
200	300	4	3000	101'000	396'538	15'395'320	883'842
100	900	9	10'000	169'200	1'442'096	102'120'485	1'032'657
200	900	9	10'000	315'000	2'746'457	164'157'757	2'569'478

Note: The number of SKUs per order is equal to 1 with probability 50%; otherwise, it is sampled from a shifted geometric distribution with parameter 0.7.
The time-space network consists of 50 time slots of time unit $\Delta = 20$ sec.
The counting has been performed with no initial condition imposed; the number of such constraints is however negligible, and bounded by $\sum_{w \in \mathcal{W}} C_w$.

Chapter 4

Heuristic Solution Method

The optimization problems presented in Section 3.3 are characterized by a large combinatorial search space, whose size grows rapidly with the instance. Solving them to optimality with exact solvers therefore quickly becomes computationally impractical. This chapter presents the heuristic algorithms developed for each optimization stage; they build on the Large Neighborhood Search (LNS) paradigm introduced by Shaw [13].

The remainder of the chapter is organized as follows. Section 4.1 introduces the general local search framework shared by both optimization stages. Section 4.2 details its instantiation for Stage 1, including the initial solution construction and the local search parameters; Section 4.3 follows the same structure for Stage 2.

4.1 Large Neighborhood Search framework

Both optimization stages are solved with a local search heuristic built around the same core mechanism and adapted to the specifics of each problem. Rather than defining the neighborhood explicitly through a fixed set of moves, which for these problems is difficult to design effectively, the heuristic follows the Large Neighborhood Search (LNS) framework. In this framework the neighborhood is characterized implicitly through two operators, following Pisinger and Ropke [14]: a *destroy* operator, which removes part of the current solution, and a *repair* operator, which reconstructs a complete solution from the partial one. The neighborhood of a solution is thus the set of all solutions reachable by applying the destroy method and then the repair method.
Algorithm 4.1 summarizes this general procedure

The destroy operator frees a limited number of decision variables at each iteration. This number is not fixed but grows adaptively: the search starts from a small subproblem, fast to repair, and frees more variables only after some consecutive

Algorithm 4.1 Time-limited large neighbourhood search

Require: feasible solution x, time budget τ, max iterations without improvement $it_no_improv^{\max}$

$x_{\text{best}} \leftarrow x$
$k \leftarrow k_{\text{init}}$ ▷ initial neighbourhood size
$it_no_improv \leftarrow 0$ ▷ iterations since last improvement
$t_0 \leftarrow \text{TIME}$
repeat
  $x_{\text{current}} \leftarrow r\big(d(x_{\text{best}}, k)\big)$ ▷ destroy k elements, then repair
  if x_{current} improves the objective over x_{best} **then**
    $x_{\text{best}} \leftarrow x_{\text{current}}$
    $it_no_improv \leftarrow 0$ ▷ reset on improvement
  else
    $it_no_improv \leftarrow it_no_improv + 1$
  end if
  adjust k according to the search progress ▷ grow on stalls, shrink on overflow
until $\text{TIME} - t_0 \geq \tau$ **or** $it_no_improv \geq it_no_improv^{\max}$
return x_{best}

iterations fail to improve the solution, up to a preset cap, expressed as a fraction of the total variables, that keeps the subproblem tractable. This keeps the early iterations fast while still allowing the larger moves needed to escape plateaus later on. The variables to be freed are drawn at random, assigning a higher weight to those selected less frequently in previous iterations, to prevent variable to be left permanently unselected.

The repair operator rebuilds a complete solution by formulating a reduced optimization problem in which only the freed variables are treated as decision variables, while the fixed ones enter the model as constants wherever they appear in the objective or in the constraints. The size of this subproblem therefore scales with the number of freed variables rather than with the size of the full instance. The new solution x_{current} is then obtained by solving the reduced problem with the commercial solver Gurobi, subject to a time limit, so that the repair returns the best solution found within that budget.

As can be seen in Algorithm 4.1, the search relies on two stopping criteria. The first bounds the overall computational time, while the second bounds the number of iterations without improvement. For the latter, the maximum number of allowed iterations is fixed to 100 for both stages.

4.2 Stage 1 Solution Approach

The first local search algorithm addresses the optimization problem introduced in Section 3.3.1, following the Large Neighbourhood Search scheme of Section 4.1 with the destroy and repair operators specialized as follows. The search starts from the feasible solution built by the greedy construction described at the end of this section.

The destroy operator draws k orders from the batch considered and frees the decision variables associated with them: all rows z_{ow} of each selected order o, all the variables x_{iop} of its item-order pairs (i, o), and the columns y_{wp} of the touched pods, that is, the pods brought to a workstation to supply the SKUs required by the freed orders. All the remaining variables are kept fixed at their incumbent values.

The repair operator builds the reduced problem over the freed variables, with the fixed ones entering as constants. Two couplings with the fixed part require care: a fixed item picked from a touched pod still requires that pod to be brought to its workstation, which is imposed directly on the corresponding visit variable; and the workload-balance constraint receives a constant load from the fixed orders, to which only the freed orders add variable terms. The solution x_{current} is the optimum returned by Gurobi on this reduced problem, and it replaces the incumbent whenever it lowers the objective.

The behaviour of the heuristic is governed by a small set of parameters, whose values were fixed empirically. For the first stage, the neighbourhood starts at $k_{\text{init}} = 30$ freed orders and is allowed to grow up to $k_{\max} = 80$; each repair subproblem is solved by Gurobi with a time limit of 5 s and an optimality gap of 2%, while the overall search runs under a wall-clock budget of 30 s. The neighbourhood is enlarged after three consecutive stalls and, more mildly, after a merely marginal improvement, while it is shrunk whenever the freed part would exceed a cap set at half of the decision variables.

The initial solution is derived by the greedy construction of Algorithm 4.2, which seeks a feasible assignment while bringing as few pods as possible to the workstations. Assigning an order o to a workstation w opens a set $\mathcal{P}^{\text{new}}(o, w)$ of new pods, one for each SKU of o not yet covered at w, chosen as the closest pod stocking it. The insertion cost is the contribution this adds to objective (3.1),

$$c(o, w) \;=\; \sum_{p \in \mathcal{P}^{\text{new}}(o,w)} \Big(1 + \alpha\, dist(w, p)\Big), \tag{4.1}$$

with α as in Section 3.3.1. Since each pod stores several SKUs, a pod is counted only once even when it supplies more than one SKU of the order, and pods already brought to w are reused at no cost; hence $c(o, w)$ shrinks as the SKUs of o overlap with those already served at w, which makes the construction greedy with respect

to the objective.

The construction runs in three phases. Orders already open at a workstation are first fixed to it, activating their pods. Each workstation is then filled up to its minimum SKU-load bound, adding at each step the order of least cost per SKU and favouring those that share a SKU with its active pods. The orders still unassigned are finally placed on the cheapest workstation that respects the maximum SKU-load bound. Large candidate sets are subsampled at random to keep the construction fast.

Algorithm 4.2 Initial solution construction for Stage 1

Require: orders, workstations $\mathcal{W}$, SKU set $\mathcal{S}(o)$ of each order

$z, x, y \leftarrow 0$; $\ell_w \leftarrow 0$ for all w ▷ ℓ_w: current SKU load of workstation w

$L \leftarrow \lfloor 0.8\,\Sigma/|\mathcal{W}| \rfloor$, $U \leftarrow \lceil 1.2\,\Sigma/|\mathcal{W}| \rceil$ ▷ $\Sigma = \sum_o |\mathcal{S}(o)|$

Fix the orders already open at a workstation and activate their pods

$\mathcal{F} \leftarrow$ the remaining unassigned orders

for each workstation $w \in \mathcal{W}$ **do** ▷ fill up to the lower bound

  while $\ell_w < L$ **and** $\mathcal{F} \neq \emptyset$ **do**

    $\mathcal{C} \leftarrow \{\, o \in \mathcal{F} : \mathcal{S}(o)$ shares a SKU with the pods active at $w \,\}$

    $o \leftarrow \arg\min\ c(o, w)/|\mathcal{S}(o)|$ over $\mathcal{C}$, or over $\mathcal{F}$ if $\mathcal{C} = \emptyset$

    assign o to w, update the pods; $\ell_w \leftarrow \ell_w + |\mathcal{S}(o)|$; remove o from $\mathcal{F}$

  end while

end for

for each remaining order $o \in \mathcal{F}$ **do** ▷ leftovers under the upper bound

  $\mathcal{W}_{\text{adm}} \leftarrow \{\, w : \ell_w + |\mathcal{S}(o)| \leq U \,\}$

  $w \leftarrow \arg\min_{w \in \mathcal{W}_{\text{adm}}} c(o, w)$, or the least-loaded w if $\mathcal{W}_{\text{adm}} = \emptyset$

  assign o to w, update the pods; $\ell_w \leftarrow \ell_w + |\mathcal{S}(o)|$; remove o from $\mathcal{F}$

end for

return z, x, y

4.3 Stage 2 Solution Approach

The second local search algorithm addresses the optimization problem introduced in Section 3.3.2, following the Large Neighbourhood Search scheme of Section 4.1 with the destroy and repair operators specialized as follows. The search starts from the feasible solution built by the greedy construction described at the end of this section.

The destroy operator draws k pods among those selected by the first stage

and frees the decision variables associated with them: the variables x_{oit} of each item-order pair (o, i) whose SKU is retrieved from a freed pod, the rows f_{ot}, g_{ot} and v_{ot} of the affected orders, and the rows y_{pa} describing the routes of the freed pods.

The repair operator builds a reduced network, since the routes of the fixed pods do not change, and the fixed routes enter as constants when needed. Here the fixed variables appear as constant terms in three constraints: the bound on the number of orders open simultaneously at a workstation, the workload-balance constraint at each constraint, and the bound on the number of pods out of storage at the same time. In each of them, the contribution of the fixed part is precomputed from the incumbent and added as a constant, so that only the freed variables remain as unknowns.

The parameters governing this stage were again fixed empirically. Because the repair subproblems are considerably heavier here, both the per-solve and the overall time limits are larger than in the first stage: the neighbourhood starts at $k_{\text{init}} = 15$ freed pods and may grow up to $k_{\max} = 50$; each repair subproblem is solved by Gurobi with a time limit of 30 s and an optimality gap of 2%, while the overall search runs under a wall-clock budget of 300 s. As before, the neighbourhood is enlarged after three consecutive stalls and, more mildly, after a merely marginal improvement, while it is shrunk whenever the freed part would exceed a cap set at 35% of the decision variables.

The initial solution is built by the greedy construction of Algorithm 4.3, one workstation at a time. Only the picking schedule x is constructed, as the other variables can be derived from it. As in the first stage, orders sharing pods are grouped into batches, so that one pod visit serves several orders, and consecutive batches are chained on their common pods, so that a pod already at the workstation is reused by the next batch.

Workstation capacity is enforced through a counter $a(t)$, the number of orders in progress at time t. An order runs from its first to its last pick and is committed only if $a(t)$ stays within the capacity C over that interval. Orders can therefore overlap, and a slot frees up as soon as an order completes.

The construction runs in four steps. Orders already open are scheduled first, since they occupy a slot over the whole horizon until completed. The remaining orders are grouped into pod-sharing batches, each seeded by an order sharing pods with many others and grown by the order adding the most sharing for the fewest new pods. The batches are then scheduled oldest first, committing an order only if all its pods find a slot and $a(t)$ has room over its window; a long pod visit is split across consecutive slots to respect the per-slot work budget. Orders that fail are retried individually, sliding their earliest time forward until a feasible region is found.

Algorithm 4.3 Initial solution construction for Stage 2

Require: orders per workstation, pod of each item, capacity C, horizon T

$x \leftarrow 0$
for each workstation w **do**
    $a(t) \leftarrow 0$ for all t ▷ $a(t)$: orders in progress at w at time t
    split the orders of w into $\mathcal{O}_{\text{open}}$ (already open) and $\mathcal{O}$ (the rest)
    $a(t) \leftarrow a(t) + |\mathcal{O}_{\text{open}}|$ for all t ▷ open orders hold a slot until completed

    Schedule $\mathcal{O}_{\text{open}}$; when one finishes at its last pick L, free $a(t)$ for $t > L$; retry any failures, oldest first

    Group $\mathcal{O}$ into pod-sharing batches and order them by chaining on shared pods

    Schedule the batches
    for each batch, and each order o in it, oldest first **do**
        reserve a pod visit for the items of o, split over consecutive slots to respect the per-slot work budget
        let $[f, \ell]$ be the picking window of o
        if all pods of o got a slot **and** $a(t) < C$ for all $t \in [f, \ell]$ **then**
            commit o: write its picks into x and add 1 to $a(t)$ on $[f, \ell]$
        else
            mark o as failed
        end if
    end for

    Retry the failed orders, oldest first, sliding the earliest time forward until a feasible region is found
end for
return x

4.4 Validation on Small Test Cases

This section presents a preliminary assessment of the proposed heuristic, comparing its solutions with those of the exact solver Gurobi when the two stages are solved in sequence on four small test cases. In the intended application, a fast solution is more valuable than one close to the optimum, so the heuristic is assessed primarily on its ability to return good solutions quickly. The results are not conclusive, as there is no guarantee that the observed behaviour carries over to the larger instances considered later in the thesis.

All test cases share the same dimensions: $|\mathcal{O}| = 50$, $|\mathcal{P}| = 50$, $|\mathcal{T}| = 50$, $|\mathcal{W}| = 3$, $|\mathcal{I}| = 500$. Gurobi is given a time limit of 20 minutes per stage, while the heuristic is limited to 30 seconds on the first stage and 200 seconds on the second.

The assessment targets two aspects. Generalization is addressed by considering four distinct test cases; stability, by solving each of them five times while varying the solver's internal seed.

Table 4.1 reports aggregate statistics over the five replications of each test case. The two stages behave differently and are therefore discussed separately.

In the first stage, Gurobi closes every test case to optimality, which provides an exact reference against which the heuristic can be measured. The heuristic does not reach this optimum, but it terminates in a small fraction of the time, a few seconds against the tens of seconds required by the exact solver.

In the second stage the situation is reversed. Gurobi does not close the problem to optimality within the time limit, so no certified optimum is available. The comparison is therefore made against the dual bound reported by the solver, which provides an upper limit on the true optimum. Across all test cases the objective attained by the heuristic remains close to this bound, while the runtime stays well below the limit imposed on the exact solver, although in the worst case the heuristic reaches its own time limit.

Finally, the heuristic appears stable across replications: for each test case, the worst-case objective value is generally close to the average one, suggesting that the internal randomization of the solver has a limited effect on the quality of the returned solution.

Table 4.1: Heuristic validation on small test cases.

		Test Case 1		**Test Case 2**		**Test Case 3**		**Test Case 4**	
		avg	*worst*	*avg*	*worst*	*avg*	*worst*	*avg*	*worst*
Stage 1	Exact Opt.	34.2	34.2	33	33	30.6	30.6	31.1	31.1
	LNS Obj.	37.18	38	37.28	38.30	34.06	36.5	35.86	37.4
	Exact Time [s]	38.95	47.99	16.07	17.40	29.51	33.95	29.51	33.95
	LNS Time [s]	7.06	10.8	3.84	5.1	3.94	4.8	4.74	7.4
Stage 2	Incumbent Sol.	69.65	61.52	48.80	27.41	77.95	73.68	66.55	60.33
	Best Dual Bound	90.88		90.95		82.19		82.25	
	LNS Obj.	89.02	88.75	88.84	88.43	80.61	80.33	79.94	79.46
	Exact Time [s]	900.25	900.35	900.25	900.32	900.37	900.56	900.3	900.41
	LNS Time [s]	175.64	200.3	141.80	194.5	120.06	200.4	174.78	200.2

Chapter 5

Evaluation Framework

This chapter presents the simulation environment used to evaluate the decisions produced by the optimization framework in a dynamic RMFS setting. The optimization model described in the previous chapters operates on a static representation of the system and therefore cannot capture the continuous arrival of orders and the resulting changes in the system state. The simulator overcomes this limitation: it reproduces the main operational dynamics of the RMFS in a simplified form and provides the performance measures used to compare the two decision-making strategies considered in this thesis.

The two strategies differ in how decisions are made. When optimization is disabled, decisions are taken by local greedy policies that react to events as they occur; when it is enabled, they are produced by the rolling-horizon optimization framework at regular intervals. This distinction shapes the sequence of events generated during a run and is therefore considered throughout the design of the simulator.

The chapter is organized as follows. Section 5.1 introduces the discrete-event simulation paradigm and the time-advance mechanism used in the simulator. Section 5.2 describes the simulator architecture: the main entities, the initial state, the events that drive the system and the collection of performance measures. Section 5.3 describes the two decision-making strategies and, in particular, how the solution of the optimization model is translated into objects executable by the simulator. Finally, Section 5.4 presents the input parameters of an instance and the procedures used to generate orders and distribute SKUs across pods. The full implementation is available in Appendix A.

5.1 Discrete-Event Simulation

According to Law [15, ch. 1], a discrete-event simulation represents a system whose state variables change only at a countable set of points in time. These points correspond to *events*, that is, occurrences that may change the state of the system.

This work adopts the *next-event time advance* mechanism. The simulation clock starts at zero and jumps directly to the time of the next event in the event calendar; the state is then updated and any resulting events are scheduled. The process repeats until the event calendar is empty or the simulation horizon is reached. Algorithm 5.1 summarizes the procedure.

Algorithm 5.1 Main discrete-event simulation loop

1: Initialize simulation clock $t \leftarrow 0$
2: Initialize event calendar $\mathcal{E}$ with initial events
3: Initialize system state S
4: **while** $\mathcal{E} \neq \emptyset$ and $t < T_{max}$ **do**
5: $e \leftarrow$ `pop_next_event`$(\mathcal{E})$
6: $t \leftarrow$ `time`(e)
7: Update state S according to the type of e
8: Schedule any new events generated by e into $\mathcal{E}$
9: **end while**
10: **return** collected performance statistics

This approach reproduces the dynamics of the RMFS without requiring a closed-form representation of the system, which is essential here because each decision alters the subsequent evolution of the system and, in turn, the information available to later decisions. A closed-form model would be unable to capture this feedback between decisions and system state.

Because the system is stochastic, a single run represents only one realization of its behaviour. Multiple replications are therefore performed to characterize performance through a distribution of outcomes rather than a single trajectory. Each replication uses a fixed random seed, which makes the experiments reproducible and allows the two decision-making strategies to be evaluated under the same conditions.

5.2 Simulator Architecture

The simulator is organized around three main components: a set of entities representing the physical elements of the warehouse, a state containing their current

status and the future events, and a collection of trackers used to measure performance during the simulation. These components are described below.

5.2.1 System Entities and State Variables

The physical and operational elements of the RMFS are represented by a set of core entities, each implemented as a Python dataclass.

- **Order**: a customer order, characterized by its arrival time, the SKUs it requires, the SKUs still to be picked, and a lifecycle status ranging from `BACKLOG` to `CLOSED`. Consistently with the pod representation, item quantities are not modelled: an order requires a set of distinct SKUs, whose number is the order's number of *order lines*.
- **Pod**: a storage pod containing a set of SKUs and having status `IDLE` or `BUSY`. Its position is explicitly tracked while the pod is idle. Detailed routing is not modelled.
- **Robot**: a single AMR with a current position and status `IDLE` or `BUSY`.
- **Workstation**: a picking station with a capacity for open orders and released tasks, processing times, and buffers for pods and orders waiting to be served.
- **Task and Visit**: a `Task` is defined by a pod, a priority determining its dispatching order, and an ordered sequence of `Visits`. Each visit corresponds to a workstation stop and specifies the orders and SKUs to be delivered there. A task can therefore move a pod through several workstations before returning it to storage. The AMR is not assigned when the task is designed; it is selected later, when the task is dispatched.
- **Warehouse**: the container of the global warehouse state, including the pods, workstations, AMRs and the layout in which they operate.

Fixed parameters and evolving quantities are kept apart through two dedicated classes, `SimulatorConfig` and `SimulatorState`.

- **`SimulatorConfig`** (immutable): contains the order-generation parameters, the simulation horizon, the warm-up period and the boolean flag `optimization_enabled`. The latter determines the decision-making mode: `False` selects the local policies, while `True` activates the rolling-horizon optimizer.
- **`SimulatorState`** (mutable): contains the simulation clock, the current warehouse state, the future-event queue, and the orders and tasks currently present in the system.

5.2.2 Simulation Engine

The discrete-event mechanism introduced in Section 5.1 is implemented by the `Simulator` class, which owns the immutable `SimulatorConfig` and a callable that returns a fresh `Warehouse` on demand. Each call to its `run()` method carries out one replication, rebuilding the `SimulatorState` and the statistics manager from scratch before advancing the event loop; as a result, consecutive replications are fully independent and retain no state from previous ones.

Randomness is handled through two separate pseudo-random generators seeded with the same value: one drives the generation of the incoming order stream, the other supplies all remaining stochastic draws. Since the optimizer and the local policies differ in the number and timing of the draws they perform, a single shared generator would desynchronize the order stream between the two strategies. Keeping the order generator isolated ensures that, for a fixed seed, both strategies face exactly the same sequence of incoming orders.

5.2.3 Initialization

At time zero, the warehouse is initialized by placing the pods in the storage grid and filling them according to Algorithm 5.6. Each AMR starts idle and is placed at a random position in the warehouse.

The system does not start empty. An initial backlog of orders is already present when the simulation clock reaches zero, so that the system starts from a loaded state rather than gradually building up its workload from an empty warehouse. This initial backlog is also intended to partially reproduce the effect of a warm-up period, while avoiding the additional computational effort required to simulate it explicitly.

Statistics are collected from time zero unless a warm-up period is specified in `SimulatorConfig`. In the experiments presented in Chapter 6, the warm-up period is set to zero.

5.2.4 Events

The simulation progresses entirely through events. An event changes the state of the system and can schedule one or more future events. The implemented event types are the following.

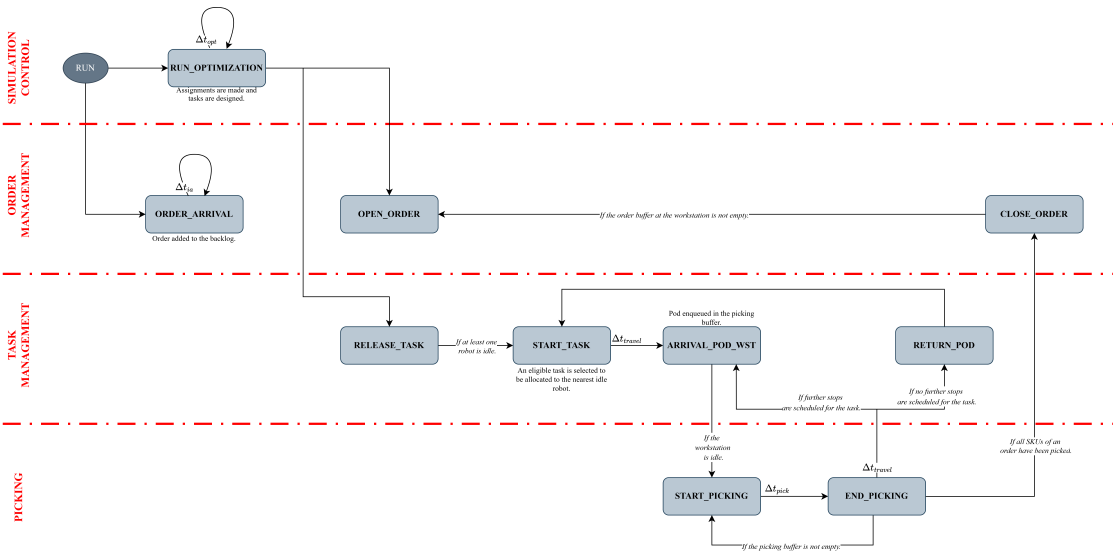
- `ORDER_ARRIVAL`: generates a new order, places it in the backlog and schedules the next arrival. When `optimization_enabled = False`, the order is immediately assigned to a workstation.
- `OPEN_ORDER`: is triggered when a workstation has spare order capacity. The order starts being processed at the workstation. Under the local policies,

opening an order also triggers a redesign of the tasks associated with that workstation. In both decision-making modes, the event triggers `RELEASE_TASK`.

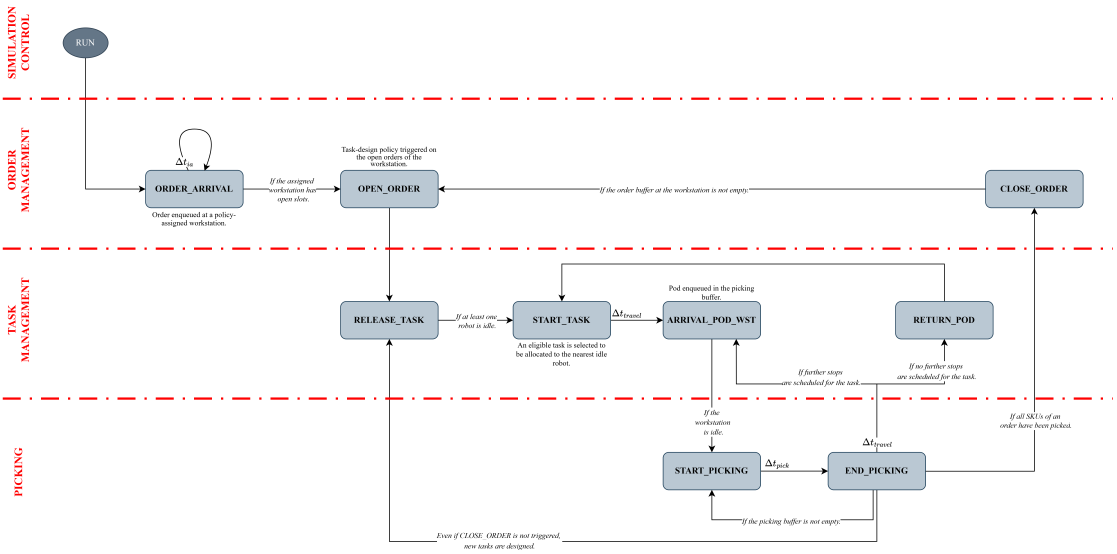
- **`CLOSE_ORDER`**: occurs when the last item of an order has been picked. The order is completed, the corresponding workstation slot is released and an `OPEN_ORDER` event is triggered.
- **`RELEASE_TASK`**: makes designed tasks available for execution. Under the optimizer, the tasks have already been designed and the event only realizes their release at the appropriate point in the simulation. Under the local policies, a release event is scheduled as soon as a task is designed.
- **`START_TASK`**: is triggered by `RELEASE_TASK`. Released tasks are considered in priority order. A task can start only if its pod is idle and the required orders are open. If an idle AMR is available, the task is assigned to the nearest one and an `ARRIVAL_POD_WST` event is scheduled after the corresponding stochastic travel time.
- **`ARRIVAL_POD_WST`**: occurs when a pod reaches a workstation. If the workstation is idle, the pod immediately starts being processed and `START_PICKING` is triggered. Otherwise, the pod waits in the workstation buffer.
- **`START_PICKING`**: starts the picking operation for the SKUs specified by the task and the orders associated with the visit.
- **`END_PICKING`**: updates the orders affected by the completed picks. If an order has received its last item, `CLOSE_ORDER` is triggered. Depending on the remaining visits in the task, the pod either moves to the next workstation or returns to storage.
- **`RETURN_POD`**: returns the pod to its storage location. Both the pod and the AMR become idle. The AMR remains at the pod location, which becomes its position for the next task.
- **`RUN_OPTIMIZATION`**: the event specific to `optimization_enabled = True`. It is triggered at fixed intervals and runs the optimization framework over the rolling horizon. The resulting plan is then mapped into tasks that can be executed by the simulator.

Figure 5.1 compares the event flows in the two decision-making modes. The general structure is shared by both modes, but the events that trigger order opening and task release depend on the value of `optimization_enabled`.

The travel times associated with the arrows labelled Δt_{travel} in Figure 5.1 are stochastic. A nominal travel time is first obtained by dividing the relevant distance by the AMR speed. A noise term is then sampled from a Beta(2,10) distribution rescaled on $[0,0.5\,\Delta t_{nom}]$. On average, this adds approximately 8% to the nominal travel time and accounts in a simplified way for congestion and for acceleration phases that are not explicitly modelled.



(a) Rolling-horizon optimizer (`optimization_enabled = True`).



(b) Dispatching policies (`optimization_enabled = False`).

Figure 5.1: Event flow of the simulation model under the two decisional frameworks. An arrow means that one event schedules the next; the Δt at its tail is the delay before the new event is triggered, and the label in italics is the condition for it to be scheduled at all.

The distance metric depends on whether the AMR is loaded. When carrying a pod, the AMR is constrained to the warehouse aisles and the distance is computed

using the ℓ_1 metric. When unloaded, it can move underneath the pods and the ℓ_2 metric is used.

As a consequence, the delay associated with `START_TASK` contains two legs: the unloaded movement from the AMR to the pod and the loaded movement from the pod to the first workstation. The delay associated with `END_PICKING` contains only a loaded movement from the current workstation to the next workstation or, when the task is complete, to the pod storage location.

Synchronization between simulator and optimizer

`RUN_OPTIMIZATION` is the point where the simulator state has to be transferred to the optimization model. The optimization is executed while the system is already operating, so the state provided to the optimizer cannot simply be a copy of the state at the current simulation time.

Tasks that are already being executed cannot be cancelled. Their future picks are therefore treated as already performed when the optimization problem is initialized. Orders that will be completed by these tasks are not considered by the optimizer, even though they are still represented as open by the simulator at the instant at which the optimization is called. On the other hand, released tasks that have not yet been dispatched are instead discarded. The demand associated with these tasks is consequently returned to the pool of items considered by the optimizer. The orders that remain open at the workstations are frozen and passed to the optimization model as part of its initial condition.

The optimizer therefore works on a projected state: tasks already in execution are accounted for through their expected future effects, while tasks that have not started are removed from the system and replanned. This keeps the optimization problem consistent with the state that the simulator is expected to reach when the new plan becomes relevant.

5.2.5 KPI Collector

The class `StatManager` collects the performance measures generated during a simulation run. Each measure is delegated to a tracker implementing the corresponding estimator. Unless otherwise specified, the simulation horizon is denoted by $[0, T]$.

The state of a discrete-event model is piecewise constant between two consecutive events. Time-weighted measures can therefore be computed exactly over each interval without numerical integration. When the monitored state changes at time t_j, the tracker accumulates the area $x(t_{j-1})(t_j - t_{j-1})$ and updates the reference value. At the end of the observation period, the accumulated area is divided by its duration.

Four types of trackers are used.

`ResourceTracker` Computes the utilization of a resource r, namely the fraction of the observation period during which the resource is busy:

$$\hat{\rho}_r = \frac{1}{T}\int_0^T \mathbb{1}_{\{r \text{ busy at } t\}}\,\mathrm{d}t = \frac{1}{T}\sum_j \mathbb{1}_{\{r \text{ busy on } [t_{j-1},t_j)\}}(t_j - t_{j-1}).$$

For a set R of homogeneous resources, the individual utilizations are averaged over $|R|$. This tracker is used for workstations and AMRs.

`TimeWeightedMeanTracker` Computes the time-weighted mean of a state variable $x(t)$:

$$\hat{x} = \frac{1}{T}\int_0^T x(t)\,\mathrm{d}t = \frac{1}{T}\sum_j x(t_{j-1})(t_j - t_{j-1}).$$

It is used to measure the average number of pods moving simultaneously.

`OrderFlowTracker` Computes the mean flow time of the orders completed during the observation period:

$$\hat{W} = \frac{1}{n}\sum_{i=1}^{n}(c_i - a_i),$$

where a_i and c_i are the arrival and completion times of order i, respectively, and n is the number of completed orders. The statistic is reported both globally and by number of order lines.

`Counter` Counts the occurrences of a specified event. In this work it is used to count the total number of SKUs picked, denoted by $N(T)$.

The distinction between these trackers is relevant to the interpretation of the results. Resource utilization and the average number of moving pods are time averages of the system trajectory. Mean flow time, instead, is a sample average over the orders completed during the run. These estimators describe different aspects of the system and should not be interpreted as equivalent measures.

5.3 Decision-Making

The `optimization_enabled` flag selects between the two decision-making strategies compared in this thesis. When it is disabled, decisions are made reactively by the local greedy policies, using only the information available when and where an event is triggered. When it is enabled, they come instead from the rolling-horizon optimizer of Chapter 3, invoked at regular intervals, whose solution is translated into simulator objects before execution.

The choice of the AMR executing a task is the same in both modes: once a task is ready and an idle AMR is available, the nearest one is assigned to it. This step is left outside the optimization model, so the travel times seen by the optimizer

cover the movement of the pod but not the initial trip of the AMR from its current position to the selected pod.

5.3.1 Local Policies

The local policies are event-driven. They react to the event that triggered the decision and have no access to a global representation of the warehouse state. Two greedy policies are implemented: one assigns orders to workstations, the other selects the pods that retrieve the required SKUs at a given workstation.

Policy 1: Order-to-Workstation Assignment This policy is triggered whenever a new order arrives, and assigns it to the least loaded workstation. The load of a workstation is the number of orders already waiting in its queue plus those currently open there.

Algorithm 5.2 Order-to-workstation assignment

1: assign the order to the workstation minimizing the sum of its queued and currently open orders

The orders assigned to a workstation are not sequenced explicitly: within each queue they are opened in arrival order, following a first-in-first-out (FIFO) rule.

Policy 2: Pod Selection This policy is triggered by the orders open at a workstation: it reads the SKUs they require and calls in the pods that can supply them. The pods are chosen greedily, one at a time, taking at each step the idle pod that covers the most still-uncovered SKUs, with distance to the workstation breaking ties. Each pod so chosen becomes a task with a single visit, grouping the open orders it contributes to. The loop stops once the workstation's task capacity is reached, all required SKUs are covered, or no remaining pod can add anything. At every invocation the policy runs afresh, rebuilding the tasks already released but not yet dispatched, so that newly opened orders are handled together with those already waiting.

Two limitations follow from acting locally. First, every task holds a single visit: a decision triggered by one workstation sees nothing of the others, and so cannot plan a pod trip that serves several of them. Second, the number of tasks is bounded by `WS_RELEASED_TASK_CAPACITY`, which limits how many tasks may be released and active at a workstation at once; without it, a fleet larger than the stations can process would pile up around a workstation without adding any throughput.

As in the order-assignment policy, the tasks are not sequenced explicitly. Their priority follows from the time at which they are built and released.

Algorithm 5.3 Pod selection

$U \leftarrow$ SKUs required by the orders open at the workstation, minus those already covered by its active tasks
if $U = \emptyset$ **then**
    return no tasks
end if
$c \leftarrow$ residual task capacity of the workstation ▷ capacity minus active tasks
while fewer than c tasks have been built **and** $U \neq \emptyset$ **do**
    among the idle pods not already selected, pick the pod p covering the most SKUs of U, breaking ties by distance to the workstation
    if no such pod covers any SKU of U **then**
        break
    end if
    build a visit on the orders that p contributes to, and create a task carrying it
    $U \leftarrow U \setminus \text{items}(p)$
end while
return the tasks

5.3.2 Optimizer Mapping

Unlike the greedy policies of the previous subsection, which act directly on the simulator's state, the optimizer reasons on the mathematical formulation of Chapter 3. This subsection describes how the binary variables of the second stage, which schedule the pod trips over the discrete time grid, are turned into objects the simulator can execute. These are `Task` objects, each tied to a pod and carrying an ordered list of visits and a priority. The order-to-workstation assignment is left out of this step, since the first stage has already fixed it and passes it on. Algorithm 5.4 covers what remains.

The optimizer's solution defines two sequences: the order in which the orders open at the workstations and the order in which their SKUs are picked. The picking sequence does not hold in the simulation, where travel times are subject to stochastic delays and the picks may no longer follow the planned order. The opening sequence does hold, since it is the order in which the workstations are told to serve orders. The mapping is therefore built on the opening sequence alone.

A pod's picks are split into separate tasks to avoid keeping it out longer than its work requires: while a pod is out it holds an AMR, and the AMRs are limited, so a pod left idle between two distant picks wastes a robot the rest of the fleet could use. Algorithm 5.4 groups a pod's picks by opening window, the step from which the order they serve becomes available at its workstation, and scans them in order,

Algorithm 5.4 Mapping of the optimizer solution onto simulator objects

Require: the second-stage binaries x, v; the order-to-workstation assignment
Ensure: the tasks to release; the order in which orders open at each workstation

Opening time.
for all orders o **do**
    $\tau_o \leftarrow \min\{t : v_{o,t} = 1\}$, or 0 if o is already open ▷ v: order-open variable
    drop o if $v_{o,t} = 0$ for all t
end for

Grouping.
for all picks (i, o) of a kept order **do**
    $t \leftarrow \min\{s : x_{(i,o),s} = 1\}$ ▷ x cumulative: first step the pick is done
    add i to the group (pod of i, τ_o, workstation of o), keeping its earliest pick
end for

Tasks.
for all pods p **do**
    sort the groups of p by opening time τ
    scan them in order; start a new trip whenever τ jumps by more than a round trip home
    for all trips **do**
        merge the groups on the same workstation into one visit
        release a task for p, its visits ordered by earliest pick
    end for
end for

Ordering.
sort the tasks by earliest pick and keep only their rank as priority
at each workstation, open the orders by τ, breaking ties by task rank

starting a new trip whenever the gap between two consecutive windows is wide enough for the pod to return to storage and come back out. This threshold is the pod's round trip, so it grows with distance, and a pod far from its workstation is kept out across larger gaps than a near one. The tasks are then ordered by their earliest pick, and this order, reduced to ranks, is what the simulator dispatches on.

Cutting on the opening windows was compared in simulation with cutting on the pick times. The latter gave a lower throughput, consistently with the fact that the picking sequence is not reproduced during the run, and the window rule was adopted.

5.4 Input Parameters

The numerical parameters defining an experimental configuration are read from an external CSV file. Table 5.1 lists the parameters used by the simulator and groups them according to their role. From these parameters, the simulator artificially generates all the data required for a run, such as the pod contents and the stream of incoming orders.

The following paragraphs describe two stochastic procedures that rely on these input parameters to generate specific parts of an instance. The first generates the orders, while the second determines how the available SKUs are distributed among the pods.

Order Generation The order-generation procedure follows, with minor adaptations, the code made publicly available by Barnhart et al. [8]. The number of order lines is generated through a split between single-line and multi-line orders. The parameters p_1 and p_g control the probability of the first case and the decay of the second, respectively.

The SKUs required by an order are then sampled from a normal distribution centred on the middle of the catalogue. This choice introduces a non-uniform demand profile: SKUs around the centre of the catalogue are more likely to be requested, while SKUs near the boundaries remain possible but less frequent. Algorithm 5.5 summarizes the procedure.

The empirical distributions produced by Algorithm 5.5 are shown in Figures 5.2 and 5.3.

In both configurations a single-line order is the most frequent outcome, with a frequency close to p_1 (about 0.5 and 0.4 respectively). The two cases differ mainly

[1]The geometric distribution is sampled via NumPy's `numpy.random.Generator.geometric` [16], whose support is $\{1,2,3,\dots\}$ (number of trials until the first success, inclusive), so that $G \geq 1$ always and the two branches of the order-size sampling remain mutually exclusive.

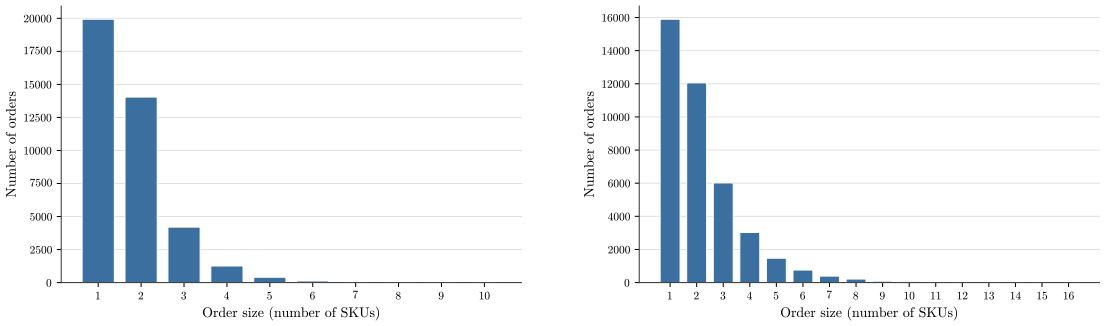
Table 5.1: Numerical parameters of the simulator, grouped by functionality.

Parameter	Description
Layout and capacity	
NUM_PODS	Total number of pods (GRID_ROWS × GRID_COLS)
NUM_SKUS	Number of distinct SKUs available
NUM_SKUS_PER_POD	Number of distinct SKUs stored in each pod
NUM_ROBOTS	Number of autonomous mobile robots
NUM_WORKSTATIONS	Number of picking stations
GRID_ROWS / GRID_COLS	Pod grid dimensions
WS_ORDER_CAPACITY	Maximum number of open orders per station
WS_RELEASED_TASK_CAPACITY	Maximum number of released and active tasks per station
Operational times	
ROBOT_SPEED	AMR speed
POD_PROCESS_TIME	Processing time of a pod at the workstation
ITEM_PROCESS_TIME	Picking time of a SKU at the workstation
Order generation	
INTERARRIVAL_TIME_ORDER	Order inter-arrival time
PROB_1_ITEM_ORDER	Probability that an order contains a single SKU
GEO_DIST_PARAM_ORDER	Geometric parameter for multi-SKU orders
Simulation control	
TIME_HORIZON	Total simulation duration
INITIAL_BACKLOG_SIZE	Number of orders waiting in the backlog at time zero
WARM_UP_TIME	Initial interval during which no statistic is collected
DELTA_T_OPT	Optimizer call interval
SEED	Seed of the random generator

Algorithm 5.5 Order generation

Require: catalogue $\mathcal{I}$ of distinct SKUs; p_1 = `PROB_1_ITEM_ORDER`; p_g = `GEO_DIST_PARAM_ORDER`; $s_{\max} = 30$

1: $U \sim \mathrm{Unif}(0,1)$ ▷ coin flip for single-line orders
2: $S \leftarrow 1$ if $U < p_1$, else $S \leftarrow G + 1$ with $G \sim \mathrm{Geom}(p_g) \mid G + 1 \leq s_{\max}$ ▷ number of order lines[1]
3: $(\mathrm{SKU}_i)_{i=1}^{S} \overset{\text{i.i.d.}}{\sim} \mathcal{N}\left(\frac{|\mathcal{I}|}{2}, \left(\frac{|\mathcal{I}|}{6}\right)^2\right)\Big|_{[1,|\mathcal{I}|]}$ ▷ the S SKUs, centred on the middle of the catalogue
4: **return** $[\mathrm{SKU}_1, \ldots, \mathrm{SKU}_S]$



(a) Instance with $p_1 = 0.5$ and $p_g = 0.7$.

(b) Instance with $p_1 = 0.4$ and $p_g = 0.5$.

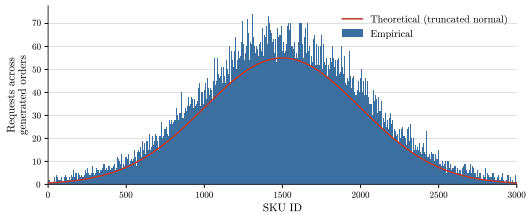
Figure 5.2: Empirical distribution of the number of order lines generated by Algorithm 5.5 under two parameter configurations, computed on a sample of 40 000 generated orders.

in the tail. With $p_1 = 0.5$ and $p_g = 0.7$, orders with more than 5 lines are already highly unlikely, whereas with $p_1 = 0.4$ and $p_g = 0.5$ the tail is heavier and orders remain non-negligible up to about 8 lines. The second configuration therefore produces a larger average number of order lines.

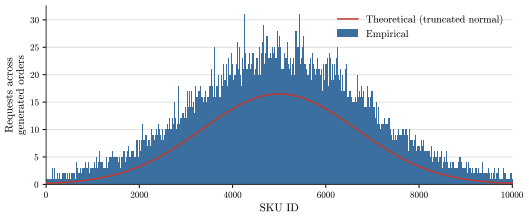
The two panels of Figure 5.3 have the same overall bell-shaped form. The difference is mainly in the scale of the distribution. Since the standard deviation is proportional to the catalogue size,

$$\sigma = \frac{|\mathcal{I}|}{6},$$

the same relative demand profile is spread over a wider range of SKU indices when $|\mathcal{I}|$ increases. The number of generated orders remains unchanged, so the peak frequency associated with an individual SKU decreases as the catalogue grows, while the overall shape of the distribution is preserved.



(a) Instance with $p_1 = 0.5$, $p_g = 0.7$, and $|\mathcal{I}| = 3000$.



(b) Instance with $p_1 = 0.5$, $p_g = 0.7$, and $|\mathcal{I}| = 10000$.

Figure 5.3: Empirical SKU request-frequency distribution across generated orders, under two catalog sizes, computed on a sample of 40 000 generated orders.

SKU Distribution across Pods The non-uniform demand profile is also reflected in the allocation of SKUs across pods. More frequently requested SKUs are assigned to more pods, creating multiple alternatives for the pod-selection decision.

The allocation is performed in two steps. First, the number of copies assigned to each SKU is determined. The resulting copies are then placed in the pods. This avoids generating an arbitrary allocation first and subsequently repairing it to guarantee that every SKU is available.

Every SKU receives at least one copy, which guarantees coverage. The remaining storage capacity is distributed according to the probabilities used to generate demand. When the proportional allocation does not produce an integer number of copies, the remaining slots are assigned using the largest-remainder rule. Ties are broken randomly to avoid introducing a relationship between SKU index and pod position.

Algorithm 5.6 summarizes the procedure. By construction, every pod is completely filled and no SKU occupies two slots of the same pod.

The replication budget

$$B = |\mathcal{P}|\, k - |\mathcal{I}|$$

determines how much storage capacity remains after assigning one copy to every SKU. If $k = |\mathcal{I}|/|\mathcal{P}|$, this budget is zero: every SKU is stored exactly once and the set of pods required by a collection of orders is uniquely determined. In that limiting case, pod selection is no longer a decision.

Increasing k creates additional copies that can be distributed according to SKU popularity. In the experiments, k is kept fixed at 30 across the warehouse sizes. This provides a consistent number of distinct SKUs per pod while leaving enough replication capacity for popular SKUs to appear in several pods. The resulting allocation is shown in Figure 5.4.

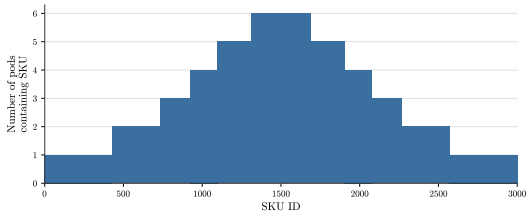
Figure 5.4 shows the number of pods containing each SKU for two representative

Algorithm 5.6 SKU-to-pod assignment

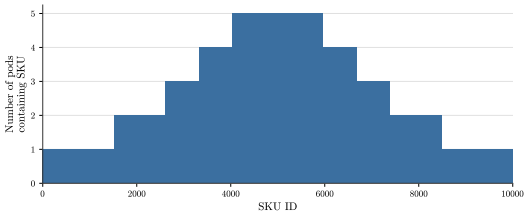
Require: catalogue $\mathcal{I}$; set of pods $\mathcal{P}$, each holding $k =$ `NUM_SKUS_PER_POD` distinct SKUs, with $|\mathcal{P}|\,k \geq |\mathcal{I}|$

Ensure: Items(p) for every pod, with $|\text{Items}(p)| = k$ and every SKU stored at least once

$B \leftarrow |\mathcal{P}|\,k - |\mathcal{I}|$ ▷ replication budget: the slots left once every SKU has one copy
$f_i \leftarrow$ probability that a single draw of Algorithm 5.5 requests SKU i
$n_i \leftarrow 1 + \lfloor B f_i \rfloor$ for every $i \in \mathcal{I}$ ▷ one copy for coverage, the rest by popularity
hand the $B - \sum_i \lfloor B f_i \rfloor$ leftover slots to the SKUs with the largest $B f_i - \lfloor B f_i \rfloor$
for each SKU i, in decreasing order of n_i **do**
    store a copy of i in each of the n_i pods with the most free slots, ties broken at random
end for



(a) Instance with $|\mathcal{P}| = 300$, $|\mathcal{I}| = 3000$ and $k = 30$.



(b) Instance with $|\mathcal{P}| = 900$, $|\mathcal{I}| = 10000$ and $k = 30$.

Figure 5.4: Distribution of the number of pods containing each SKU for two representative warehouse instances.

instances. In both cases, the allocation follows the bell-shaped demand weights, while SKUs in the tails mostly remain at one copy. The difference in the peak is explained by the replication capacity available relative to the size of the catalogue. In the first instance, 300 pods with 30 SKUs each provide 9000 slots for 3000 SKUs, leaving 6000 slots for additional copies. In the second instance, 900 pods provide 27000 slots for 10000 SKUs, leaving 17000 slots for replication. Thus, although the second instance has more replication slots in absolute terms, it has fewer additional slots per SKU: 2.0 versus 1.7. Since these additional slots are allocated according to SKU popularity, this results in a lower peak, with the most popular SKUs reaching 6 copies in the first instance and 5 in the second.

Chapter 6

Computational Experiments

This chapter compares the two decision-making strategies introduced using the evaluation framework presented in Chapter 5. Section 6.1 describes how the experimental configurations were constructed and why the results are reported as distributions rather than single values. Throughput is taken as the primary performance indicator and analyzed in Section 6.2; the effects on mean flow time and pod movement are then briefly examined in Sections 6.3 and 6.4.

6.1 Design of Experiments

Each configuration is defined by the values assigned to a subset of the simulator parameters listed in Table 5.1; the parameters held constant across all configurations are reported in Table 6.1.

Three factors define the experimental configurations: warehouse size, the average number of order lines (governed by the order-generation parameters p_1 and p_g), and the order arrival rate. The order stream is approximated by deterministic arrivals at equally spaced intervals, so the arrival rate fully defines the inter-arrival time as its reciprocal. Section 5.4 already discussed how p_1 and p_g shape the distribution of the number of lines per order; here they serve only to distinguish a lighter demand profile, with orders requiring fewer SKUs on average, from a heavier one, with orders requiring more. Crossing the three warehouse sizes with the two order-line profiles and the two arrival rates yields a full $3 \times 2 \times 2$ factorial design, that is, the twelve configurations of Table 6.2.

A configuration fixes only the deterministic part of the experiment. Each one describes a stochastic system, so a single run reflects only one realization of its performance and no conclusion can be drawn from it in isolation. Each configuration is therefore replicated over $N = 14$ independent seeds, and performance is summarized across these replications rather than from any single one. The same

Table 6.1: Parameters held constant across all configurations.

Parameter	Value	Unit	Description
NUM_SKUS_PER_POD	30	–	Distinct SKUs stored in each pod
ROBOT_SPEED	0.34	cell/s	Robot speed
WS_ORDER_CAPACITY	4	–	Maximum open orders per workstation
WS_RELEASED_TASK_CAPACITY	10	–	Maximum released and active tasks per station
POD_PROCESS_TIME	5	s	Processing time of a pod
ITEM_PROCESS_TIME	5	s	Picking time of an item
TIME_HORIZON	3600	s	Total simulation duration
INITIAL_BACKLOG_SIZE	200	–	Number of orders in the backlog at the beginning of the simulation
DELTA_T_OPT	900	s	Optimizer call interval
WARM_UP_TIME	0	s	Warm-up time

fourteen seeds are reused across all twelve configurations.

On every configuration the two decision-making strategies, the rolling-horizon optimizer and the reactive greedy policies, are evaluated on this same set of seeds. Because a seed fully determines the stream of incoming orders (Section 5.2.2), the two strategies face an identical demand realization on each seed and differ only in how they design and assign tasks. Sharing the seeds makes the comparison between the two strategies paired within each configuration, which isolates the effect of the strategy from the variability of the demand and motivates the analysis of their per-seed difference in the following sections.

Determining the number of replications

The number of replications N is determined through the two-stage procedure of Law [15, ch. 9]. From an initial pilot of n_0 replications one computes the sample mean $\bar{X}(n_0)$ and the sample variance

$$S^2(n) = \frac{1}{n-1} \sum_{i=1}^{n} \Big(X_i - \bar{X}(n) \Big)^2, \tag{6.1}$$

evaluated at $n = n_0$. The required number of replications is then the smallest sample size, no smaller than n_0, at which the half-width of the confidence interval

Table 6.2: The twelve configurations, a full $3 \times 2 \times 2$ design crossing warehouse size, order lines and arrival rate. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

	Warehouse	**Size parameters**				**Order lines**		**Arrival Rate**
Config.	**size**	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{I}\rvert$	R	$\lvert\mathcal{W}\rvert$	p_1	p_g	λ [ord/h]
`S·fl·lr`	Small	300	3000	20	4	0.5	0.7	400
`S·ml·lr`						0.4	0.5	400
`S·fl·hr`						0.5	0.7	600
`S·ml·hr`						0.4	0.5	600
`M·fl·lr`	Medium	600	5000	30	6	0.5	0.7	400
`M·ml·lr`						0.4	0.5	400
`M·fl·hr`						0.5	0.7	600
`M·ml·hr`						0.4	0.5	600
`L·fl·lr`	Large	900	10000	40	9	0.5	0.7	400
`L·ml·lr`						0.4	0.5	400
`L·fl·hr`						0.5	0.7	600
`L·ml·hr`						0.4	0.5	600

on the mean meets the target relative precision:

$$N = \min\left\{ i \geq n_0 \;:\; t_{i-1,\,1-\alpha/2}\,\frac{S(n_0)}{\sqrt{i}} \;\leq\; \frac{\gamma}{1+\gamma}\left|\bar{X}(n_0)\right| \right\}, \tag{6.2}$$

where $t_{i-1,\,1-\alpha/2}$ is the Student-t quantile, α the significance level and γ the target relative precision. Because the unknown variance is replaced by its first-stage estimate $S^2(n_0)$, the procedure is an approximation.

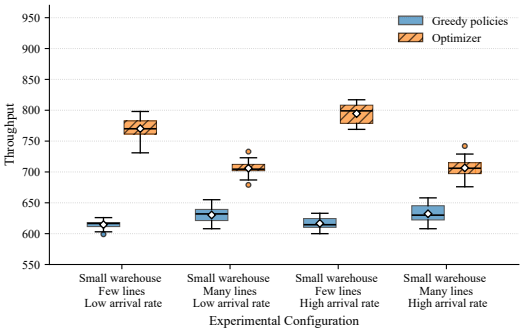
The procedure is applied with throughput as the primary response variable, $\alpha = 0.05$ and $\gamma = 0.02$. Starting from a pilot of $n_0 = 10$ replications, the criterion is already met in most configurations, while a few require a larger sample. Taking the largest requirement across configurations gives a common $N = 14$, which is used for every configuration; the resulting half-widths are reported in Appendix B. Results are consequently reported as distributions over seeds rather than as single point estimates, through boxplots in the figures and standard deviations in the tables.

In the following sections, each performance metric is reported per configuration through aggregate statistics over the replications, i.e. mean and standard deviation

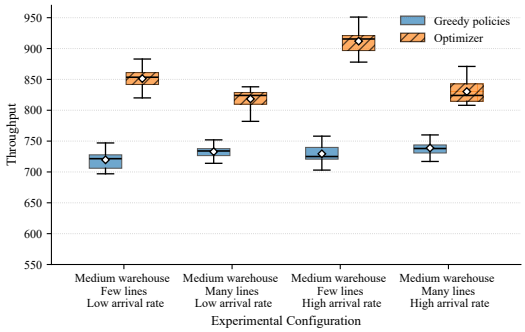
in the tables and the seed distribution in the boxplots. The complete per-replication tables are provided in Appendix B.

6.2 Throughput Analysis

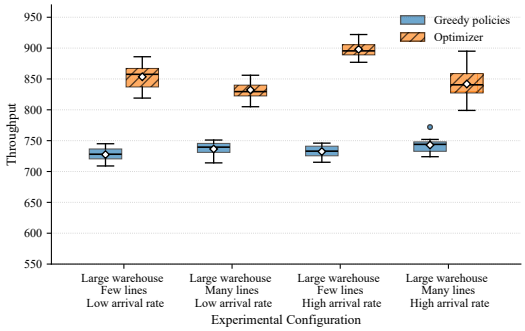
Throughput is defined as the number of SKUs picked over the simulation horizon. It provides the most direct measure of system productivity and is also the quantity that the second optimization stage aims to maximize.



(a) Small warehouse



(b) Medium warehouse



(c) Large warehouse

Figure 6.1: Comparison of the **throughput** achieved by the two decision-making strategies across the considered test instances. Boxplots report the variability across simulation seeds.

Figure 6.1 and Table 6.3 show that the optimizer outperforms the greedy policies in every configuration. A more formal check of this advantage can be obtained through a t-test, a statistic test used to assess whether the mean of one or two populations differs significantly from a reference value or from one another. Applied to the per-seed differences against a reference value of zero, the test confirms the advantage: the optimizer picks on average 121.8 more items than the greedy

Table 6.3: Comparison of the **throughput** achieved by the two decision-making strategies across the considered test instances. Values are reported as mean and standard deviation over the simulation seeds.

	Greedy policies		**Optimizer**	
Configuration	**Mean**	**Std**	**Mean**	**Std**
S · fl · lr	614.64	7.33	**770.21**	17.25
S · ml · lr	630.29	13.86	**706.00**	14.14
S · fl · hr	616.57	10.09	**794.57**	16.54
S · ml · hr	632.21	14.91	**706.57**	17.61
M · fl · lr	719.64	15.55	**851.29**	17.61
M · ml · lr	732.86	10.54	**818.36**	16.25
M · fl · hr	729.36	15.90	**912.00**	18.38
M · ml · hr	738.79	12.15	**830.21**	19.98
L · fl · lr	727.43	10.97	**853.50**	20.54
L · ml · lr	736.50	10.88	**832.21**	14.66
L · fl · hr	732.36	10.06	**898.00**	14.12
L · ml · hr	742.93	11.89	**841.79**	27.02

policies (95% CI $[115.03, 128.49]$), and the null hypothesis of zero difference cannot be accepted ($t_{167} = 35.73$, p-value $p < 0.001$).

What changes most is the size of this advantage in the different configurations. The plots and the table already reveal a clear pattern as the average order lines move from few to many: the two strategies react in opposite ways. Comparing `S·fl·lr` with `S·ml·lr`, for instance, the greedy policies reach a higher throughput as orders get longer, while the optimizer's throughput drops, so the gap between them narrows. A possible reason is that the optimizer produces a plan, but during execution the pod movements are affected by random delays, so the execution drifts away from the plan. Longer orders keep the workstations busy longer and rely more on coordinating decisions across the system, which is exactly what the optimizer provides. When the execution diverges from the plan, this coordination is lost and the advantage shrinks. The greedy policies react to the current state instead of following a plan, so they have no coordination to lose. With shorter orders the workstations turn over faster, the plan stays closer to the execution, and the optimizer's advantage is largest. This remains a hypothesis, consistent with the observed pattern, not a tested claim.

The pattern suggests that the advantage of the optimizer is not constant but depends on the experimental factors, order lines in particular. To assess this more

rigorously, the advantage is studied through an analysis of variance (ANOVA), which decomposes the total variability of a response into the contributions of the factors and their interactions and tests whether the differences between factor levels exceed what random variability alone would produce.

6.2.1 Factors affecting the optimizer's advantage

Because the two strategies are evaluated on the same seeds, at each configuration they face an identical stream of orders, so the two sets of replications are not independent. To account for this, the analysis is carried out on the per-seed difference

$$\mathcal{D}_{c,s} = \mathrm{thr}^{\mathrm{opt}}_{c,s} - \mathrm{thr}^{\mathrm{pol}}_{c,s}, \tag{6.3}$$

where c denotes the configuration and s the seed. Taking the difference on each seed cancels the demand realization common to both strategies and isolates the effect of the strategy itself.

The difference is modelled as a function of the three experimental factors and their interactions,

$$\mathcal{D} \sim \text{warehouse size} \times \text{order lines} \times \text{arrival rate}, \tag{6.4}$$

with warehouse size on three levels (small, medium, large) and order lines and arrival rate on two levels each (few/many and low/high). One level of each factor is taken as reference, small for warehouse, few for order lines and low for arrival; their combination, configuration `S·fl·lr`, is the baseline against which all effects are measured.

Table 6.4 reports the analysis of variance together with the effect size of each term, measured by

$$\eta^2 = \frac{\mathrm{SS}_{\mathrm{effect}}}{\mathrm{SS}_{\mathrm{total}}}, \tag{6.5}$$

i.e. the share of the total variability of $\mathcal{D}$ explained by that term. The direction of each effect is not reported here, as it is already visible in the boxplots of Figure 6.1.

Before interpreting the results, the normality of the residuals was assessed through the Shapiro–Wilk test, which does not reject the null hypothesis of normality ($W = 0.99$, $p = 0.39$; see the Q–Q and residual plots in Figure B.1). The assumptions of the analysis of variance can therefore be considered satisfied.

This analysis confirms the pattern already observed in the plots. Order lines is by far the dominant factor: it is highly significant ($p < 0.001$) and alone explains about 63% of the variability in the advantage. The arrival rate has a significant but much smaller effect ($\eta^2 \approx 0.05$); a higher arrival rate widens the optimizer's advantage, as a larger backlog gives the optimizer more orders to choose from and thus more room to coordinate resources effectively. The warehouse size, in

Table 6.4: Analysis of variance on the per-seed difference $\mathcal{D}$ (optimizer minus greedy policies), with the corresponding effect sizes (η^2).

```
Analysis of Variance Table

Response: D
                                          Df Sum Sq Mean Sq  F value    Pr(>F)
warehouse_size                             2    103      52   0.1065   0.89906
order_lines                                1 203845  203845 419.9343 < 2.2e-16 ***
arrival_rate                               1  17001   17001  35.0224 1.998e-08 ***
warehouse_size:order_lines                 2  13071    6536  13.4638 4.036e-06 ***
warehouse_size:arrival_rate                2   2282    1141   2.3508   0.09866 .
order_lines:arrival_rate                   1  12933   12933  26.6421 7.357e-07 ***
warehouse_size:order_lines:arrival_rate    2    802     401   0.8263   0.43956
Residuals                                156  75726     485
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Effect Size for ANOVA (Type I)

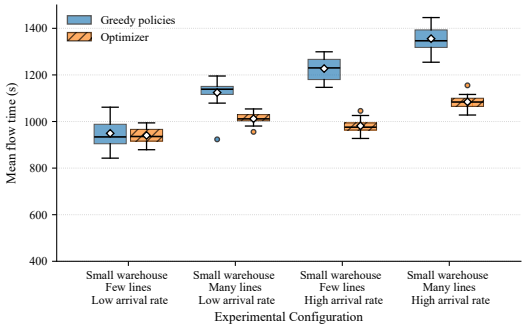
Parameter                                 |     Eta2 |       95% CI
-------------------------------------------------------------------
warehouse_size                            | 3.17e-04 | [0.00, 1.00]
order_lines                               |     0.63 | [0.55, 1.00]
arrival_rate                              |     0.05 | [0.01, 1.00]
warehouse_size:order_lines                |     0.04 | [0.00, 1.00]
warehouse_size:arrival_rate               | 7.01e-03 | [0.00, 1.00]
order_lines:arrival_rate                  |     0.04 | [0.00, 1.00]
warehouse_size:order_lines:arrival_rate   | 2.46e-03 | [0.00, 1.00]
```

contrast, has no significant main effect ($p = 0.90$), meaning that the size of the warehouse, on its own, does not change how much the optimizer gains over the greedy policies. Two interactions are significant, warehouse size × order lines and order lines × arrival rate, but their effect sizes are small ($\eta^2 \approx 0.04$ each), so they only slightly modulate the dominant effect of the order lines.

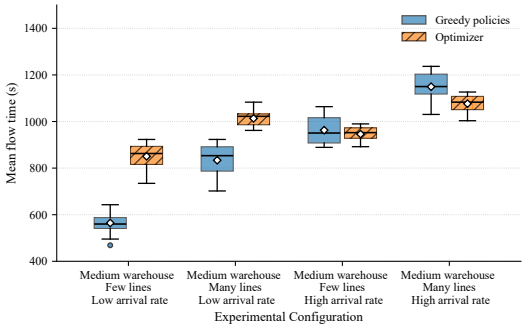
6.3 Mean Flow Time Analysis

Order flow time is the time an order spends in the system, from its arrival until its completion. The metric reported here is the mean flow time across all completed orders. It complements throughput: while throughput measures how much the system produces, flow time captures how long orders remain in the system before they are served.

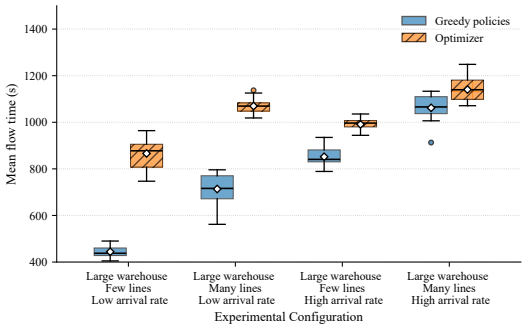
As shown in Figure 6.2 and Table 6.5, the mean flow time, computed over the completed orders only, does not consistently favour either strategy. The optimizer is



(a) Small warehouse



(b) Medium warehouse



(c) Large warehouse

Figure 6.2: Comparison of the **mean order flow time** achieved by the two decision-making strategies across the considered test instances. Boxplots report the variability across simulation seeds.

faster on the small warehouse, whereas the greedy policies are faster on the medium and large ones, in some configurations by a noticeable margin. This contrasts with the throughput, where the optimizer is ahead everywhere.

The explanation arises from how the completed orders are distributed across the number of lines per order (Appendix B). In every configuration the optimizer closes more orders than the greedy policies, and this gain comes almost entirely from few-line orders. Since these are quick to process, this alone would suggest a lower flow time for the optimizer, yet the opposite is observed especially on the large warehouses. The difference lies in how the two strategies sequence the orders. The greedy policies act reactively, processing orders in a FIFO fashion, so that orders are completed close to their arrival order. The optimizer, in contrast, plans over the whole decision epoch and is free to postpone some orders to coordinate pods and workstations more effectively. This improves throughput, but it also keeps orders waiting longer before completion, which raises their flow time.

Table 6.5: Comparison of the **mean order flow time** achieved by the two decision-making strategies across the considered test instances. Values are reported in seconds, as mean and standard deviation over the simulation seeds.

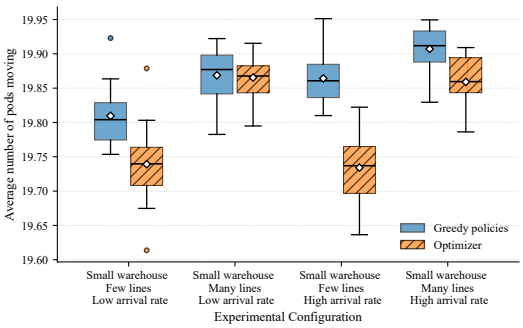
	Greedy policies		**Optimizer**	
Configuration	**Mean** [s]	**Std** [s]	**Mean** [s]	**Std** [s]
S · fl · lr	949.29	63.11	**940.28**	33.95
S · ml · lr	1124.24	64.96	**1012.04**	26.42
S · fl · hr	1227.27	49.33	**981.24**	32.92
S · ml · hr	1355.62	58.43	**1084.51**	32.39
M · fl · lr	**564.38**	49.71	850.65	57.05
M · ml · lr	**833.76**	72.39	1013.40	34.10
M · fl · hr	962.69	61.89	**946.04**	33.36
M · ml · hr	1149.28	57.31	**1076.59**	36.12
L · fl · lr	**444.14**	24.18	866.11	66.85
L · ml · lr	**713.51**	67.94	1069.77	33.73
L · fl · hr	**851.99**	45.39	991.05	25.70
L · ml · hr	**1061.97**	59.98	1140.30	53.53

The higher flow time of the optimizer is therefore the cost of a latency-for-throughput trade-off: more orders are completed, at the price of a longer wait for each. On the small warehouse the orders are the same, but the shorter travel distances let the system complete them faster, so the extra waiting introduced by reordering weighs much less on the overall flow time.

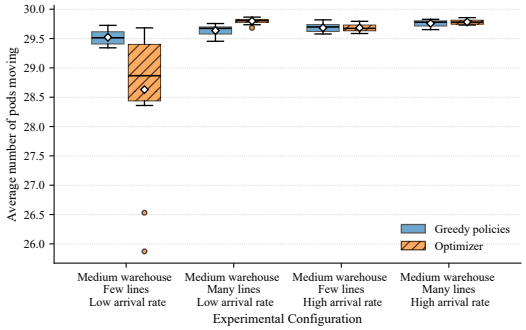
6.4 Pod Movement Analysis

The average number of simultaneously moving pods is computed as the time-weighted mean of the number of pods in transit at each instant of the simulation. In isolation, a high value is not necessarily undesirable, as it may simply reflect a fuller use of the available fleet. This interpretation is, however, limited to the scope of this thesis, since AMR routing and battery management are not modeled in detail.

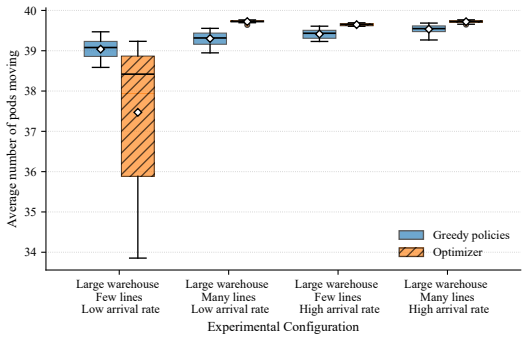
As shown in Figure 6.3 and Table 6.6, the two strategies keep a comparable number of pods in transit, even though the optimizer achieves a higher throughput. In both cases this number is close to the fleet size, so the robots are moving for most of the simulation under either strategy. Since both approaches operate at a comparable level of saturation, the throughput advantage of the optimizer cannot



(a) Small warehouse



(b) Medium warehouse



(c) Large warehouse

Figure 6.3: Comparison of the **average number of simultaneously moving pods** for the two decision-making strategies across the considered test instances. Boxplots report the variability across simulation seeds.

be attributed to a more intensive use of the fleet, but rather to a more effective coordination of its movements.

Table 6.6: Comparison of the **average number of simultaneously moving pods** for the two decision-making strategies across the considered test instances. Values are reported as mean and standard deviation over the simulation seeds.

	Greedy policies		**Optimizer**	
Configuration	**Mean**	**Std**	**Mean**	**Std**
S · fl · lr	19.81	0.05	19.74	0.06
S · ml · lr	19.87	0.04	19.87	0.04
S · fl · hr	19.86	0.04	19.73	0.05
S · ml · hr	19.91	0.03	19.86	0.04
M · fl · lr	29.52	0.13	28.63	1.13
M · ml · lr	29.64	0.09	29.80	0.05
M · fl · hr	29.68	0.08	29.68	0.06
M · ml · hr	29.76	0.06	29.78	0.04
L · fl · lr	39.04	0.27	37.47	1.92
L · ml · lr	39.30	0.17	39.72	0.03
L · fl · hr	39.41	0.12	39.65	0.03
L · ml · hr	39.54	0.11	39.72	0.04

Chapter 7

Conclusion

This thesis addressed the task design problem within an RMFS by introducing a two-stage formulation, consisting of two static optimization problems solved in sequence over an order backlog. This strategy was embedded into a rolling-horizon framework and benchmarked, within a discrete-event simulator, against reactive greedy policies that rely only on local information.

Chapter 6 showed that the optimizer outperforms the greedy policies on throughput, used in this work as the primary indicator of performance. This improvement, however, generally comes with a longer flow time, as some orders are delayed to improve the overall system efficiency. The optimizer achieves these results using the same number of AMRs, which suggests that its advantage stems from a better coordination of pod movements rather than from simply moving more pods.

These results are consistent with the trade-off discussed earlier. An optimizer with a global view of the system has, in principle, more room to coordinate decisions than reactive greedy policies acting on local information alone. This advantage, however, comes at a cost. To keep the optimization tractable within the rolling-horizon setting, the heuristic is given a fixed time budget of five minutes to produce each plan (Chapter 4). While this budget is a design choice rather than an intrinsic limit of the method, it is already a substantial fraction of the fifteen-minute decision epoch, and in some replications the search uses it entirely. In a real deployment the requirement is more demanding still: to keep the system operative, a new plan must be available almost immediately at each epoch, so that operations are not paused while waiting for the optimizer. A planning time of several minutes is difficult to reconcile with this requirement. The contribution of this thesis is therefore not a claim of superiority, but a characterization of the trade-off between the quality of system-level decisions and the time required to compute them, which any practical deployment of such a method must ultimately confront.

A further contribution of this work is the evaluation framework itself. The decision-making logic is decoupled from the rest of the system, so that different

algorithms can be tested without modifying the core simulation model. Any method able to produce the required `Task` objects, whether a different heuristic, an exact solver, or other priority-based rules, can be plugged into the same event loop and evaluated using the same set of KPIs. This makes the framework suitable not only for comparing the approaches considered in this thesis, but also for testing more advanced decision-making strategies. In particular, a promising direction for future work would be to develop an optimizer able to produce high-quality plans within a much shorter computational time, which is precisely the bottleneck identified above.

Some limitations of the current model should also be acknowledged. The management of AMRs is not modeled in detail, leaving out dispatching, routing, and battery-related decisions. Similarly, pod replenishment decisions are not considered, nor are other decisions that may interact with task generation and assignment. These aspects were kept outside the scope of the thesis to focus on the tactical decisions under investigation.

These limitations also point to several directions for future work. The simulator could be extended to include a more realistic representation of AMR operations, replenishment missions, and their interaction with task design. This would allow the proposed decision-making approaches to be evaluated in a progressively more realistic warehouse environment, while preserving the modular structure of the evaluation framework.

Appendix A

Source Code

The full implementation of the simulator and of the optimisation pipeline described in this thesis is publicly available at:

`https://github.com/MartiChiodo/OMERO_TaskDesign.git`

The repository is organised as follows.

Repository structure

```
Simulator/
├── run_simulation.py ..... entry point: reads config, builds warehouse, runs simulation
├── experiments.csv ..... numeric parameters of the simulation scenario
└── scripts/
    ├── core/
    │   ├── enums.py ..... status and event type enumerations
    │   ├── queues.py ..... priority queue with lazy deletion
    │   ├── entities.py ..... domain entities: Order, Task, Visit, Pod, Robot, Workstation, Event
    │   └── warehouse.py ..... physical layout, entity generation, travel times
    ├── sim/
    │   ├── Simulator.py ..... discrete event engine: state, clock, event loop
    │   ├── event_handler.py ..... event logic, one handler per event type
    │   └── utils.py ..... miscellaneous helpers
    ├── opt/
    │   ├── policies.py .. heuristic policies used when the optimiser is disabled
    │   ├── OptManager.py ..... optimisation pipeline and time space network
    │   ├── local_search_stage1.py .... first version of the local search for the assignment problem (Stage 1)
    │   ├── stage1_LNS.py .. LNS implementation for the assignment problem (Stage 1)
    │   ├── local_search_stage2.py .... first version of the local search for the scheduling problem (Stage 2)
    │   ├── stage2_LNS.py .. LNS implementation for the scheduling problem (Stage 2)
    │   ├── stage2_data.py ..... data structures feeding the Stage 2 problem
    │   ├── build_initial_x_stage2.py ..... construction of a feasible initial solution for Stage 2
    │   └── convert_OptSol_to_SimObj.py ... conversion of decision variables into simulator Task objects
    └── stat/
        ├── StatManager.py ..... KPI collection and reporting
        └── core.py ... sub trackers: resources, order flow times, time weighted means
```

Running the simulator

The entry point is `run_simulation.py`. A run is configured through three global variables at the top of the script:

- `EXPERIMENTS_ID` lists the identifiers of the scenarios to simulate. Each identifier refers to a row of `experiments.csv`, which stores the numeric parameters of that scenario.

- `SEEDS` lists the seeds used to initialise the random number generator, one simulation per seed.
- `OPTIM` is a boolean flag selecting the decision-making strategy: the rolling-horizon optimiser when `True`, the reactive greedy policies when `False`.

To reproduce the experiments, it is sufficient to update `EXPERIMENTS_ID` and `SEEDS` and set `OPTIM`. The script then iterates over the two lists in parallel: for each scenario and seed it configures the logging, seeds the generator, instantiates the `Warehouse` from the corresponding row of `experiments.csv`, optionally plots the layout, and finally creates a `Simulator` instance and calls `sim.run(TIME_HORIZON)`.

Appendix B

Detailed Computational Results

This appendix reports the complete per-replication results for every configuration, whose aggregate values are discussed in Chapter 6. For each metric, the two strategies are reported separately, one table per strategy, with configurations on the rows and seeds on the columns. For the throughput, the primary response variable, the replication-sizing tables and the ANOVA diagnostic plots are also included.

Throughput and replication sizing

The per-replication throughput is reported below for both strategies. The accompanying sizing tables document the two-stage replication procedure of Section 6.1, reporting for each configuration the confidence-interval half-width at $N = 14$ against the target relative precision.

Table B.1: Greedy policies: per-replication throughput (items picked over the time horizon). Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	616	613	609	603	617	615	618	626	624	617	611	619	599	618
S·ml·lr	627	620	625	612	632	608	634	641	650	655	632	641	614	633
S·fl·hr	615	602	614	611	630	610	625	633	625	612	623	623	600	609
S·ml·hr	629	621	623	608	629	609	646	646	658	647	643	639	622	631
M·fl·lr	713	697	709	727	702	728	747	723	704	720	730	705	746	724
M·ml·lr	737	714	741	738	726	736	752	728	722	730	732	720	736	748
M·fl·hr	723	713	740	740	703	723	758	738	711	723	727	720	753	739
M·ml·hr	755	717	744	739	733	743	760	737	726	737	730	726	754	742
L·fl·lr	739	712	732	709	714	725	731	745	739	719	738	725	729	727
L·ml·lr	740	722	751	730	714	740	734	747	735	744	746	723	746	739
L·fl·hr	738	723	732	715	725	734	732	743	746	727	742	744	716	736
L·ml·hr	732	724	772	731	732	744	750	748	744	745	752	735	744	748

Table B.2: Optimizer: per-replication throughput (items picked over the time horizon). Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	755	773	765	759	767	761	787	763	786	792	798	731	773	773
S·ml·lr	720	710	687	705	679	733	704	702	703	702	723	710	713	693
S·fl·hr	776	800	769	798	787	802	794	806	817	811	811	773	809	771
S·ml·hr	718	698	715	713	729	687	697	715	698	715	699	742	690	676
M·fl·lr	855	833	865	829	841	852	883	863	846	855	877	820	855	844
M·ml·lr	824	782	820	829	818	835	824	829	792	826	828	838	807	805
M·fl·hr	893	926	894	951	918	914	893	905	927	918	878	917	922	912
M·ml·hr	816	829	824	857	845	836	824	817	813	857	812	871	814	808
L·fl·lr	859	835	868	829	831	856	883	886	844	861	871	819	864	843
L·ml·lr	821	856	824	829	831	805	841	822	828	830	854	853	820	837
L·fl·hr	895	913	892	902	877	896	877	888	894	886	922	906	904	920
L·ml·hr	859	851	802	869	846	799	827	828	835	818	858	895	830	868

Table B.3: Verification of the number of replications for the greedy policies, using throughput as the response variable ($\alpha = 0.05$, $\gamma = 0.02$). For each configuration, the confidence-interval half-width $t_{N-1,\,1-\alpha/2}\, S(N)/\sqrt{N}$ computed at $N = 14$ lies below the target relative precision $\frac{\gamma}{1+\gamma}\,|\bar{X}(N)|$, confirming that fourteen replications satisfy the criterion. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

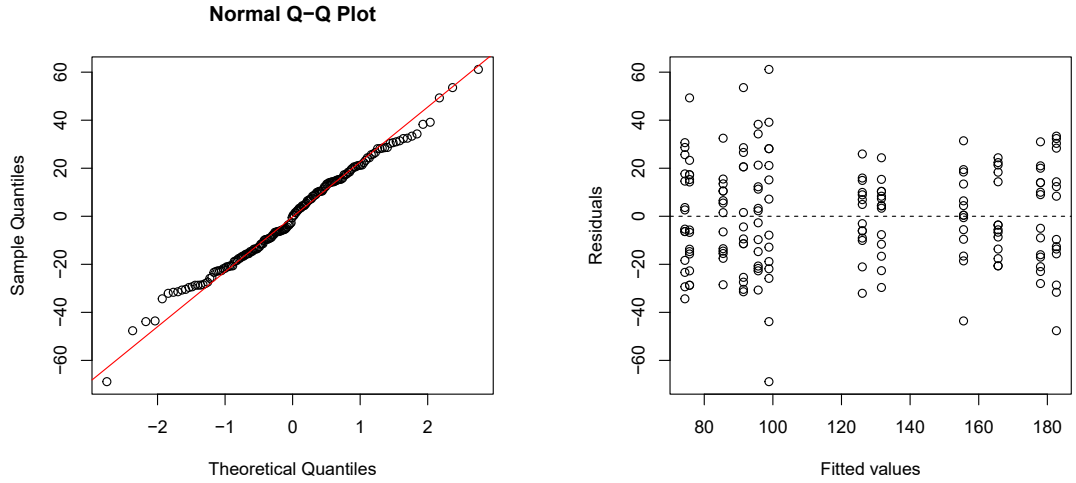
Configuration	$\bar{X}(N)$	$S(N)$	**CI half-width**	**Target**
S · fl · lr	614.64	7.33	4.23	12.05
S · ml · lr	630.29	13.86	8.00	12.36
S · fl · hr	616.57	10.09	5.83	12.09
S · ml · hr	632.21	14.91	8.61	12.40
M · fl · lr	719.64	15.55	8.98	14.11
M · ml · lr	732.86	10.54	6.08	14.37
M · fl · hr	729.36	15.90	9.18	14.30
M · ml · hr	738.79	12.15	7.01	14.49
L · fl · lr	727.43	10.97	6.34	14.26
L · ml · lr	736.50	10.88	6.28	14.44
L · fl · hr	732.36	10.06	5.81	14.36
L · ml · hr	742.93	11.89	6.87	14.57

Table B.4: Verification of the number of replications for the optimizer, using throughput as the response variable ($\alpha = 0.05$, $\gamma = 0.02$). For each configuration, the confidence-interval half-width $t_{N-1,\,1-\alpha/2}\, S(N)/\sqrt{N}$ computed at $N = 14$ lies below the target relative precision $\frac{\gamma}{1+\gamma}\,|\bar{X}(N)|$, confirming that fourteen replications satisfy the criterion. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Configuration	$\bar{X}(N)$	$S(N)$	**CI half-width**	**Target**
S · fl · lr	770.21	17.25	9.96	15.10
S · ml · lr	706.00	14.14	8.17	13.84
S · fl · hr	794.57	16.54	9.55	15.58
S · ml · hr	706.57	17.61	10.17	13.85
M · fl · lr	851.29	17.61	10.17	16.69
M · ml · lr	818.36	16.25	9.38	16.05
M · fl · hr	912.00	18.38	10.62	17.88
M · ml · hr	830.21	19.98	11.54	16.28
L · fl · lr	853.50	20.54	11.86	16.74
L · ml · lr	832.21	14.66	8.47	16.32
L · fl · hr	898.00	14.12	8.15	17.61
L · ml · hr	841.79	27.02	15.60	16.51

ANOVA assumption checks

The analysis of variance discussed in Section 6.2 assumes that the model residuals are normally distributed and have constant variance. Figure B.1 reports the diagnostic plots supporting these assumptions. The normal Q–Q plot shows the residuals lying close to the reference line, consistent with normality, while the residuals-versus-fitted plot shows no systematic pattern or funnel shape, consistent with constant variance.



(a) Normal Q–Q plot of the residuals. **(b)** Residuals versus fitted values.

Figure B.1: Diagnostic plots for the analysis of variance on the per-seed difference $\mathcal{D}$.

Mean flow time

The per-replication mean flow time is reported below for both strategies, followed by the number of completed orders broken down by the number of lines per order, which supports the discussion in Section 6.3.

Table B.5: Greedy policies: per-replication average flow time (s) total over all order sizes. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	921.8	893.2	945.1	898.3	980.9	949.6	1061.4	1038.2	842.7	990.5	1029.9	922.7	901.4	914.4
S·ml·lr	1139.6	1152.1	1110.0	1079.0	1142.2	1195.3	1178.8	1137.6	1116.1	1169.7	1117.2	923.4	1146.2	1132.3
S·fl·hr	1257.2	1184.0	1223.8	1175.5	1241.0	1236.4	1286.6	1299.3	1146.6	1269.9	1289.8	1215.4	1177.5	1178.8
S·ml·hr	1313.7	1398.4	1331.2	1303.5	1345.7	1441.9	1446.2	1377.3	1292.9	1436.5	1356.3	1254.5	1347.5	1333.4
M·fl·lr	560.0	562.5	579.2	537.9	641.1	552.4	590.3	642.8	559.9	549.2	623.4	538.1	495.4	468.8
M·ml·lr	833.2	890.1	801.4	723.3	923.0	790.6	785.9	893.3	913.9	892.0	884.7	701.9	765.6	873.6
M·fl·hr	952.4	897.1	929.3	890.8	1027.8	948.8	1048.9	1046.0	958.4	982.3	1063.8	942.5	900.6	889.0
M·ml·hr	1116.7	1236.7	1125.0	1094.2	1155.5	1206.3	1121.6	1148.0	1152.2	1203.9	1201.6	1030.4	1092.2	1205.7
L·fl·lr	429.8	450.9	427.3	461.5	456.7	418.6	490.2	431.7	404.9	474.7	471.6	424.8	439.1	436.3
L·ml·lr	746.8	783.7	671.6	671.7	760.1	664.6	677.0	685.4	772.4	764.0	780.1	561.9	654.0	795.9
L·fl·hr	850.8	835.7	838.1	837.0	874.0	828.4	934.9	895.1	800.9	883.6	924.0	788.9	843.2	793.1
L·ml·hr	1121.0	1114.1	1010.4	1006.3	1133.2	1054.4	1062.1	1040.9	1069.6	1130.9	1079.5	912.9	1035.7	1096.8

Table B.6: Optimizer: per-replication average flow time (s) total over all order sizes. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	960.8	913.0	993.8	879.0	921.1	937.5	934.3	968.1	917.1	937.0	915.6	994.1	976.9	915.5
S·ml·lr	955.5	1002.3	1015.1	1053.9	1006.2	1048.9	1032.7	1013.9	1020.2	989.6	980.6	1004.7	1008.8	1036.0
S·fl·hr	977.8	972.9	1045.6	942.9	963.2	954.3	1025.8	989.8	927.3	973.6	964.4	995.1	1024.1	980.5
S·ml·hr	1027.9	1039.9	1155.2	1064.5	1085.4	1066.2	1101.4	1116.2	1080.1	1092.1	1081.9	1064.0	1112.8	1095.7
M·fl·lr	889.0	815.6	894.9	773.1	793.5	883.0	866.7	914.1	816.2	858.5	923.0	734.5	905.4	841.4
M·ml·lr	983.4	985.7	1018.1	1030.5	1030.2	1033.1	1045.1	1026.2	1042.5	961.8	975.0	986.4	986.4	1083.0
M·fl·hr	985.7	925.6	975.5	892.0	895.7	962.8	968.0	974.4	935.4	937.4	989.7	897.7	958.2	946.4
M·ml·hr	1048.9	1114.2	1111.5	1097.6	1077.3	1045.6	1091.0	1056.3	1088.8	1062.7	1003.4	1033.4	1115.0	1126.5
L·fl·lr	864.6	746.9	948.1	784.0	878.5	906.0	904.9	964.1	882.5	876.6	931.0	788.5	867.3	782.6
L·ml·lr	1033.5	1045.8	1079.0	1054.7	1074.8	1068.9	1094.5	1084.5	1070.1	1034.3	1055.5	1018.5	1137.2	1125.4
L·fl·hr	991.3	944.2	1035.5	944.0	996.9	1009.7	1014.6	997.1	984.2	996.0	1006.2	968.9	1007.5	978.6
L·ml·hr	1071.0	1122.8	1169.0	1074.5	1106.0	1248.5	1186.3	1156.3	1170.6	1114.5	1095.9	1076.9	1187.2	1184.7

Table B.7: Greedy policies: mean number of closed orders by order size, averaged over seeds. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
S·fl·lr	178.2	127.0	36.6	10.8	3.0	1.1	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	357.0
S·ml·lr	114.1	81.5	43.4	20.9	10.4	5.6	2.9	0.9	0.7	0.6	0.2	0.1	0.1	0.0	0.1	281.4
S·fl·hr	179.8	126.4	37.0	10.9	3.1	1.2	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	358.6
S·ml·hr	113.8	82.5	42.9	20.9	10.2	5.6	2.9	1.0	0.7	0.6	0.2	0.1	0.1	0.0	0.1	281.6
M·fl·lr	211.2	147.1	42.4	13.2	3.3	1.4	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	418.9
M·ml·lr	134.5	94.6	49.5	24.2	12.3	5.9	3.4	1.2	0.6	0.9	0.2	0.1	0.1	0.0	0.1	327.5
M·fl·hr	214.7	149.1	42.6	13.4	3.4	1.3	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	424.8
M·ml·hr	135.1	95.5	50.6	24.1	12.1	5.9	3.6	1.1	0.5	0.9	0.2	0.1	0.1	0.0	0.1	329.9
L·fl·lr	214.4	147.8	42.9	13.2	3.1	1.3	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	422.8
L·ml·lr	135.9	93.9	49.7	23.9	12.1	5.4	3.4	1.0	0.5	0.9	0.3	0.1	0.1	0.0	0.1	327.2
L·fl·hr	217.3	147.1	43.4	13.4	3.1	1.2	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	425.8
L·ml·hr	137.6	94.9	49.6	23.9	12.0	5.6	3.4	1.1	0.5	0.9	0.3	0.1	0.1	0.0	0.1	330.1

Table B.8: Optimizer: mean number of closed orders by order size, averaged over seeds. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
S·fl·lr	247.1	163.6	40.0	10.8	2.9	0.9	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	465.4
S·ml·lr	197.8	121.6	37.6	14.0	5.6	3.9	1.4	0.8	0.8	0.2	0.1	0.0	0.1	0.0	0.0	384.0
S·fl·hr	313.1	163.0	31.9	8.6	2.4	0.6	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	519.9
S·ml·hr	225.8	121.5	33.6	12.1	5.4	3.4	1.3	0.9	0.6	0.2	0.0	0.0	0.1	0.0	0.1	404.8
M·fl·lr	248.2	175.7	51.7	14.8	4.0	1.6	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	496.3
M·ml·lr	198.8	124.0	45.9	19.4	9.4	6.5	2.4	1.1	0.7	0.4	0.1	0.1	0.1	0.0	0.0	408.9
M·fl·hr	321.1	187.8	42.9	12.0	3.0	1.4	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	568.3
M·ml·hr	235.7	129.7	41.9	17.3	8.9	5.3	2.4	1.1	0.6	0.4	0.2	0.0	0.0	0.0	0.0	443.5
L·fl·lr	248.2	176.0	51.8	15.0	4.0	1.7	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	496.9
L·ml·lr	193.0	110.6	44.8	23.1	11.4	7.1	2.4	1.3	0.7	0.4	0.1	0.1	0.1	0.0	0.1	395.1
L·fl·hr	311.8	170.1	45.3	14.4	3.9	1.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	547.0
L·ml·hr	224.4	105.4	45.2	21.1	10.1	6.4	2.9	1.4	0.9	0.6	0.1	0.0	0.1	0.0	0.1	418.8

Pods moving simultaneously

The per-replication average number of pods moving simultaneously.

Table B.9: Greedy policies: per-replication average number of pods moving simultaneously. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	19.82	19.75	19.92	19.81	19.78	19.79	19.86	19.83	19.77	19.80	19.81	19.76	19.85	19.77
S·ml·lr	19.90	19.90	19.92	19.85	19.88	19.83	19.88	19.83	19.88	19.88	19.84	19.78	19.87	19.92
S·fl·hr	19.83	19.83	19.95	19.89	19.85	19.87	19.92	19.89	19.85	19.87	19.86	19.81	19.87	19.82
S·ml·hr	19.94	19.93	19.95	19.89	19.92	19.89	19.93	19.85	19.91	19.91	19.89	19.83	19.91	19.95
M·fl·lr	29.67	29.48	29.73	29.62	29.45	29.55	29.61	29.47	29.39	29.39	29.66	29.34	29.61	29.34
M·ml·lr	29.70	29.76	29.69	29.60	29.61	29.66	29.69	29.52	29.52	29.71	29.72	29.45	29.57	29.70
M·fl·hr	29.74	29.62	29.82	29.73	29.62	29.72	29.75	29.67	29.63	29.58	29.76	29.60	29.73	29.58
M·ml·hr	29.79	29.83	29.80	29.73	29.75	29.77	29.79	29.70	29.69	29.81	29.80	29.65	29.71	29.82
L·fl·lr	39.32	38.87	39.47	39.19	38.94	38.98	39.17	39.24	38.59	38.82	39.30	38.86	39.20	38.60
L·ml·lr	39.44	39.56	39.40	39.14	39.21	39.28	39.32	39.32	39.14	39.42	39.46	38.95	39.13	39.46
L·fl·hr	39.52	39.29	39.61	39.51	39.33	39.40	39.51	39.48	39.35	39.30	39.54	39.26	39.47	39.23
L·ml·hr	39.61	39.69	39.59	39.46	39.50	39.53	39.57	39.50	39.44	39.62	39.62	39.27	39.47	39.65

Table B.10: Optimizer: per-replication average number of pods moving simultaneously. Configurations are labelled as `size·orders·rate`: warehouse size `S`/`M`/`L` (small/medium/large), order lines `fl`/`ml` (few/many lines), arrival rate `lr`/`hr` (low/high).

Config.	30201	50102	130944	189463	293874	300871	301060	343310	470129	570183	612937	789124	846208	987034
S·fl·lr	19.75	19.77	19.80	19.72	19.74	19.70	19.88	19.71	19.67	19.74	19.73	19.77	19.76	19.61
S·ml·lr	19.82	19.84	19.92	19.86	19.87	19.88	19.91	19.91	19.86	19.88	19.86	19.83	19.79	19.88
S·fl·hr	19.74	19.77	19.77	19.69	19.76	19.68	19.82	19.64	19.73	19.69	19.72	19.76	19.81	19.71
S·ml·hr	19.81	19.80	19.91	19.84	19.89	19.86	19.91	19.88	19.90	19.79	19.84	19.86	19.91	19.84
M·fl·lr	29.27	28.38	28.90	26.53	28.61	28.84	29.44	29.68	28.60	29.55	29.28	25.87	29.51	28.36
M·ml·lr	29.78	29.68	29.81	29.74	29.80	29.77	29.84	29.87	29.82	29.78	29.80	29.82	29.83	29.85
M·fl·hr	29.63	29.64	29.79	29.63	29.69	29.73	29.77	29.73	29.63	29.70	29.66	29.59	29.74	29.63
M·ml·hr	29.74	29.73	29.78	29.75	29.74	29.82	29.85	29.78	29.84	29.75	29.79	29.76	29.78	29.86
L·fl·lr	37.87	33.85	38.68	35.37	39.05	37.43	38.71	38.76	39.08	38.90	39.23	34.63	38.16	34.88
L·ml·lr	39.74	39.70	39.73	39.72	39.74	39.73	39.77	39.74	39.70	39.64	39.71	39.71	39.74	39.75
L·fl·hr	39.68	39.61	39.69	39.63	39.66	39.65	39.68	39.65	39.65	39.61	39.65	39.62	39.68	39.61
L·ml·hr	39.74	39.69	39.75	39.72	39.75	39.76	39.77	39.74	39.70	39.66	39.65	39.72	39.73	39.72

Bibliography

[1] M. Merschformann, T. Lamballais, M.B.M. de Koster, and L. Suhl. «Decision rules for robotic mobile fulfillment systems». In: *Operations Research Perspectives* 6 (2019), p. 100128. ISSN: 2214-7160. DOI: 10.1016/j.orp.2019.100128 (cit. on pp. iii, 4, 6).

[2] Adrien Rimélé, Michel Gamache, Michel Gendreau, Philippe Grangier, and Louis-Martin Rousseau. «Robotic mobile fulfillment systems: a mathematical modelling framework for e-commerce applications». In: *International Journal of Production Research* 60.11 (2022), pp. 3589–3605. DOI: 10.1080/00207543.2021.1926570. URL: https://doi.org/10.1080/00207543.2021.1926570 (cit. on pp. 1, 4).

[3] Nils Boysen and René de Koster. «50 years of warehousing research—An operations research perspective». In: *European Journal of Operational Research* 320.3 (2025), pp. 449–464. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2024.03.026. URL: https://www.sciencedirect.com/science/article/pii/S0377221724002261 (cit. on p. 3).

[4] Guoshuai Jiao, Haobin Li, and Min Huang. «Online joint optimization of pick order assignment and pick pod selection in robotic mobile fulfillment systems». In: *Computers & Industrial Engineering* 175 (2023), p. 108856. ISSN: 0360-8352. DOI: 10.1016/j.cie.2022.108856 (cit. on pp. 4, 6, 7).

[5] Russell Allgor, Tolga Cezik, and Daniel Chen. «Algorithm for Robotic Picking in Amazon Fulfillment Centers Enables Humans and Robots to Work Together Effectively». In: *INFORMS Journal on Applied Analytics* 53.4 (2023), pp. 266–282. DOI: 10.1287/inte.2022.1143. eprint: https://doi.org/10.1287/inte.2022.1143. URL: https://doi.org/10.1287/inte.2022.1143 (cit. on pp. 4, 5).

[6] Cristiano Arbex Valle and John E Beasley. «Order allocation, rack allocation and rack sequencing for pickers in a mobile rack environment». In: *Computers & Operations Research* 125 (2021), p. 105090. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2020.105090. URL: https://www.sciencedirect.com/science/article/pii/S0305054820302070 (cit. on pp. 4, 5).

[7] Lu Zhen, Zheyi Tan, René de Koster, and Shuaian Wang. «How to Deploy Robotic Mobile Fulfillment Systems». In: *Transportation Science* (Sept. 2023). DOI: 10.1287/trsc.2022.0265 (cit. on pp. 4, 6, 7).

[8] Cynthia Barnhart, Alexandre Jacquillat, and Alexandria Schmid. «Robotic warehousing operations: a learn-then-optimize approach to large-scale neighborhood search». In: *ArXiv* abs/2408.16890 (2024). URL: https://api.semanticscholar.org/CorpusID:272310402 (cit. on pp. 4, 7, 11–13, 40).

[9] Jingwen Wu, Zhiyuan Yang, Lu Zhen, Wenxin Li, and Yiran Ren. «Joint optimization of order picking and replenishment in robotic mobile fulfillment systems». In: *Transportation Research Part E: Logistics and Transportation Review* 194 (2025), p. 103930. ISSN: 1366-5545. DOI: 10.1016/j.tre.2024.103930 (cit. on pp. 4, 6, 7).

[10] Yanling Zhuang, Yun Zhou, Yufei Yuan, Xiangpei Hu, and Elkafi Hassini. «Order picking optimization with rack-moving mobile robots and multiple workstations». In: *European Journal of Operational Research* 300.2 (2022), pp. 527–544. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2021.08.003 (cit. on pp. 4, 6, 7).

[11] Marius Merschformann, Lin Xie, and Hanyi Li. *RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems*. 8th ed. DE: Bundesvereinigung Logistik (BVL) e.V, 2018. DOI: 10.23773/2018_8. URL: https://doi.org/10.23773/2018_8 (cit. on p. 6).

[12] Jan-Erik Justkowiak, Mikhail Y. Kovalyov, and Erwin Pesch. «A dynamic programming algorithm for order picking in robotic mobile fulfillment systems». In: *Networks* 84.4 (2024), pp. 481–490. DOI: 10.1002/net.22245 (cit. on p. 7).

[13] Paul Shaw. «Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems». In: *Principles and Practice of Constraint Programming — CP98*. Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431. ISBN: 978-3-540-49481-2 (cit. on p. 21).

[14] David Pisinger and Stefan Ropke. «Large Neighborhood Search». In: Sept. 2010, pp. 399–419. ISBN: 978-1-4419-1663-1. DOI: 10.1007/978-1-4419-1665-5_13 (cit. on p. 21).

[15] Averill M. Law. *Simulation Modeling and Analysis*. 5th. McGraw-Hill Education, 2015. ISBN: 978-0-07-340132-4 (cit. on pp. 30, 46).

[16] *Numpy Documentation*. https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.geometric.html. Accessed: 13-07-2026 (cit. on p. 40).