On-line Order Batching in a Single Server Automated Picking System using Reinforcement Learning

Brouwers, N.

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Mechanical Enginering | Manufacturing Systems Engineering | Dynamics & Control
Master Thesis

On-line Order Batching in a Single Server Automated Picking System using Reinforcement Learning

DC 2022.042

Eindhoven, April 14, 2022

Student:	Niek Brouwers 0961727
First supervisor:	prof.dr.ir. Ivo Adan
Second supervisor:	dr. Zümbül Atan
Daily supervisor :	ir. Bhoomica M. Nataraja

This report was made in accordance with the TU/e Code of Scientific Conduct for the Master thesis.

Abstract

Recent trends in e-commerce show a substantial increase in demand. Costumers order more and smaller orders and expect delivery in shorter lead times. New advancements in robotics and automation enable warehouses to keep up with these demands and their expected future growth. We consider an autonomous robot-based order picking and packing system that can pick customer orders directly from a storage rack in a dynamic environment, where orders arrive contiguously over time. In such a system, two decisions need to be made: (1) when to pick an order and (2) which orders to place in the pick-batch. We aim to develop an approach to answer these questions to minimise the weighted earliness and tardiness. Both earliness and tardiness have costs related to them; for earliness, those costs come in the form of holding cost, opportunity cost, or deterioration of perishable goods. For tardiness, these costs are often related to customer satisfaction. We found out that there was no work studied so far that solves on-line order batching with an objective to minimise earliness and tardiness.

We formulate a Semi-Markov Decision Process as a representation of the problem so that a Deep Reinforcement Learning (DRL) agent can be created. This agent learns based on the Proximal Policy Optimisation algorithm by interacting with the problem in a discrete event simulation. We developed five different agents of three types, each with varying complexity. The agents will answer two questions at discrete moments in time; (I) Should we batch and pick now, or should we wait? (II) If we batch now, what orders should we batch? These agents are tested against traditional and more advanced benchmark methods in scenarios with different demands. Most agents were able to outperform all conventional benchmarks, and one agent was able to keep up with and, in some cases, even exceed the more advanced baseline. Furthermore, we tested the contribution of each of these two questions to the two objectives with the help of a full factorial analysis. We found that the first question has most effect on the earliness objective, while the second question has most effect on the tardiness objective. We also provide some insights into the strategy the agents developed.

Overall, in this thesis, we provide a DRL approach that can keep up with advanced heuristics. This work lays a foundation for a DRL approach to solving the Online Order Batching Problem with both earliness and tardiness as an objective.

Executive Summary

Project Goal

This thesis is conducted as a part of the PhD project of Bhoomica Nataraja. With this project, we aim to optimise the batching strategy for one of the systems of our industry partner Pickr.AI. This system is an Automated storage Retrieval System (ASRS) where customer orders are picked and packed by a robotic arm.

Since e-commerce sales are at an all-time high and are expected to continue growing, these kinds of systems have become more and more important. Effective batching strategies are also crucial to prevent tardiness or earliness. Both earliness and tardiness have their own cost related to them. Earliness might lead to holding cost, deterioration of perishable goods and opportunity costs. A tardiness penalty is often related to customer compensation.

The objective of this thesis is to investigate the use of Deep Reinforcement Learning (DRL) in the on-line order batching process of this robot, to minimise both earliness and tardiness in an environment with variable demand by answering the questions when (1) and what (2) should we batch.

Methods

In this project, we make use of a discrete event simulation. In this simulation, orders arrive over time, with the arrival rate based on real-world data. Based on this data, we have created scenarios of *low*, *decreased*, *normal*, *increased* and *high* demand. A decision-maker can give input at decision moments. These moments are triggered once the picker becomes available or when the picker is idle if 30 seconds have passed since the last moment or an order arrives. To make the simulation compatible with DRL algorithms, its environment must be described as an Semi-Markov Decision Process (SMDP). We define an action space, state space, and reward function. An agent can learn by taking actions based on the given information in the action space. This action will be given a high or low reward, depending on how good the action was in that state. The objective of the DRL algorithm is to let the agent yield a cumulative reward that is as high as possible.

We create 3 types of agents that have different action spaces. Type 1 can take 3 different actions in total; wait, batch with a simple heuristic based on order arrival, or batch with a complex heuristic that uses queue content and the state of the storage system. Type 2 can also take these actions and select how many orders will be batched. Type 3 can select if it will pick or wait and can select which orders need to be batched, independent of any heuristic. The state-space consists of information about the current state of the environment. We create a state-space based on information about the queue length, orders that the complex heuristic will select, current time and arrival rate. For the agent of type 3, we create a second version, 3b, that gets information about the queue content and state of the storage system. Lastly, the reward function gives a penalty based on the tardiness of the orders and a reward for orders picked in time, which gets higher with smaller earliness. We have created a second version for the agent of type 1. This agent has a reward function based on the same principles but is normalised for the arrival rate.

We train the agents in episodes of 24 hours. In total, the agents take $2 \cdot 10^5$-$1.5 \cdot 10^6$ decisions during training, depending on the complexity of the agent. Each episode in a training session has either a *low*, *normal* or *high* arrival rate to train the agent for the varying arrival rate we have seen in the data. Training is done with the use of Proximal Policy Optimisation (PPO). This algorithm is famous for its ease of implementation and sample efficiency. The latter makes it particularly suitable for the longer episodes we train with.

To test the performance of the heuristics, we will also test against multiple benchmark methods. For the batching, we use First In First Out (FIFO) and Most-Common SKUs (MCS)-State Of Rack (SOR) as suggested by Nataraja et al. (2022). For "when" decisions, we use a heuristic that instantly picks, one that waits until there are enough orders for a full batch, one based on what is currently done in a test setup of Pickr.AI and a method on the newly introduced Fixed Time Window Batching with Carryover (FTWBC).

Results

The performance of the agents is measured in seven scenarios. Five of these are derived from different arrival rates. Another scenario is where the arrival rate is dependent on the time of the day. In the last scenario, we simulate the arrival pattern representing "Black Friday" and "Cyber Monday". In all these scenarios, agent 1b consistently scores high and, therefore the best performing agent. It shows that it can keep up with the performance of the advanced FTWBC heuristic combined with MCSSOR.

We also conduct a full factorial analysis to test which of the "when" and "what" decisions are more important. We show that both decisions become more critical in higher arrival. We also show that both decisions have more effect on tardiness than earliness. However, the effect of the "when" decision is more important for earliness than the "what" decision. This is the other way around for tardiness.

Conclusion

We have shown that DRL shows promising results for on-line order batching in an environment with variable demand. We have implemented five different agents, of which agent 1b shows the best performance. This works lays a foundation for a DRL approach to solving the Online Order Batching Problem with both earliness and tardiness as an objective.

Acknowledgements

With this report I conclude my thesis project at the Eindhoven University of Technology. There were a lot of people involved that I would like to acknowledge. First of all, I would like to thank Pickr.AI and their customer for providing this case and the valuable data, without which there would have been no project. I would also like to express my gratitude to a few people who have guided me through this project. First, my supervisors, Ivo Adan and Zümbül Atan, whose critical questions and feedback helped me improve this project. I would also like to thank Nitish Singh for the discussion and advice; this definitely helped me better understand Reinforcement Learning. Lastly, I would like to express my deepest gratitude to Bhoomica Nataraja, who went above and beyond to help me get the most out of this project. Thank you for all the valuable discussion, feedback and answers to my million questions. I think you did an amazing job as both my supervisor and mentor.

Furthermore, I would like to thank Ellen, who had to endure listening to my never-stopping train of thoughts. You have been nothing but helpful, and I am so happy to have you around.

Since this thesis marks the end of my student life, I would like to use this opportunity to thank those who made it special. I want to thank my sister and my parents, without their support I would have never been in this position. Finally, I want to thank all my friends who made my student life amazing, and a special thanks to the guys of HK for making it unforgettable.

Contents

List of Figures

List of Tables

Acronyms

ASRS Automated storage Retrieval System. iv, 1, 13

BMH Best Multi-Heuristic. 5

CNN Convolutional Neural Network. 8, 12

DNN Deep Neural Networks. ix, 8

DQN Deep Q-Network. 9–11

DRL Deep Reinforcement Learning. ii, iv, v, vii, 6, 8, 11, 22, 25, 29, 51, 52

FIFO First In First Out. v, 5, 23, 27, 28, 35–45, 47, 48, 51, 52

FTWB Fixed Time Window Batching. 14, 27, 51

FTWBC Fixed Time Window Batching with Carryover. v, 27, 51

GNIS Greatest Number of Identical SKUs. 4, 5

GNST Greatest Number of Sorting. 4, 5

IP Instant Picking. 27, 35–43, 45, 47–49, 51, 52

KL Kullback-Leibler. 10

MCS Most-Common SKUs. v, 4, 5, 27, 28, 34, 35, 37–39, 41–45, 48, 51, 52, 63

MDP Markov Decision Process. vii, 6, 8, 9, 15–17, 21

MLP Multilayer Perceptron. 8

NN Neural Network. 7, 8, 29

OBP Order Batching Problem. 4, 13–15

OOBP Online Order Batching Problem. ii, v, 1–3, 13–15, 51, 52

PPO Proximal Policy Optimisation. ii, v, vii, 4, 10–12, 16, 22, 29, 52

RL Reinforcement Learning. vii, 4–6, 8, 12, 15, 16, 18, 26

RNN Recurrent Neural Network. 8

SKU Stock Keeping Unit. 2, 4, 5, 19–24, 27, 30

SMDP Semi-Markov Decision Process. ii, iv, 6, 16, 21

SOR State Of Rack. v, 5, 27, 28, 34, 35, 37–39, 41–45, 48, 51, 52, 63

TRPO Trust Region Policy Optimisation. 10, 11

VTWB Variable Time Window Batching. 14, 27, 51

W20 Wait 20. 27, 34–43, 45, 51, 52

WC Wait Combined. 27, 35–39, 41–43, 45, 48, 51, 52

WFB Wait for Full Batch. 27, 35, 39, 41–43, 48

1 Introduction

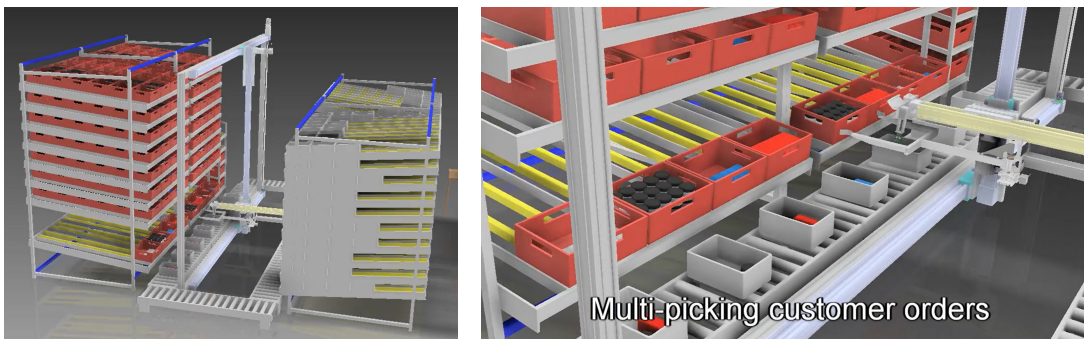
In this chapter, we introduce the problem provided by Pickr.AI. We give a small introduction to the Online Order Batching Problem (OOBP) in Section 1.1, and we discuss the primary goal of this research in Section 1.2. Lastly, in Section 1.3 we define the research questions.

1.1 Pickr.AI

Retail e-commerce sales are at an all-time high and are expected to continue growing (eMarketer Editors, 2021). Consumers are placing more and more online orders, and more of these orders become smaller (e.g. on average, an order at online retailer Amazon consists of just 1.6 items (Boysen et al., 2019b)). Furthermore, consumers expect next day, if not, same-day delivery (Boysen et al., 2019a). All these trends put the warehouses and e-fulfilment centres up for a challenge. Since most e-commerce markets are highly competitive, meeting consumers' expectations is highly prioritised. This creates the need for fast order fulfilment.

To competitively manage the many-and-small customer orders, picked from a large and varying product selection, spread over a large warehouse area, in minimal time, a flexible and efficient order fulfilment system is required. One of these systems is created by our industry partner of this thesis, Pickr.AI (Pickr.ai, 2022).

Pickr.AI has developed an automated order fulfilment system. This system is a robot-based compact storage and retrieval system that functions as an Automated storage Retrieval System (ASRS) and added functionalities for picking and packing. One embodiment of the system consists of a storage rack to store items, a robot that processes multiple orders at once, and a conveyor to transport the order boxes. In Figure 1.1a, the arrangement of the picking system is shown. The left storage rack holds the item bins (red), and the right rack holds the Consumers order boxes (grey). The boxes are transported between the racks via a closed-loop conveyor system. The robot between these two racks can load and unload the item bins and consumers boxes, pick items from the storage bins and place them in the customer order boxes.



(a) Order picking system

(b) Robot picking in action, from storage to customer boxes.

Figure 1.1: Robotic order picking system

The scope of this study is limited to the robot, the conveyor and the item storage rack, shown in Figure 1.2. Actions like restocking the storage and sending the orders will be left out of scope. The system consists of two main components: a compact storage system and a pick-and-place robot. The compact storage consists of multiple queues where items, otherwise

known as Stock Keeping Unit (SKU)s, are stored back-to-back in uniform bins. The storage system consists of the pick face and the reserve area. The pick face area is shown in Figure 1.2 with the label "Active Pick Face". The robot can directly pick the SKUs from the storage bins in this area. The conveyor in front of this pick-face area holds and moves the customer order boxes. The remaining part of the storage rack is used as a reserve storage area. The whole storage system uses gravity flow rollers, so the SKU access is according to the last-in-first-out policy. When an item is needed from a bin that is not in an active layer, the robot must first move the blocking containers and place them in other empty spots. Once all orders on the conveyor are filled, the boxes are transferred to a buffer area to wait for shipment. The process is repeated until all orders are fulfilled.

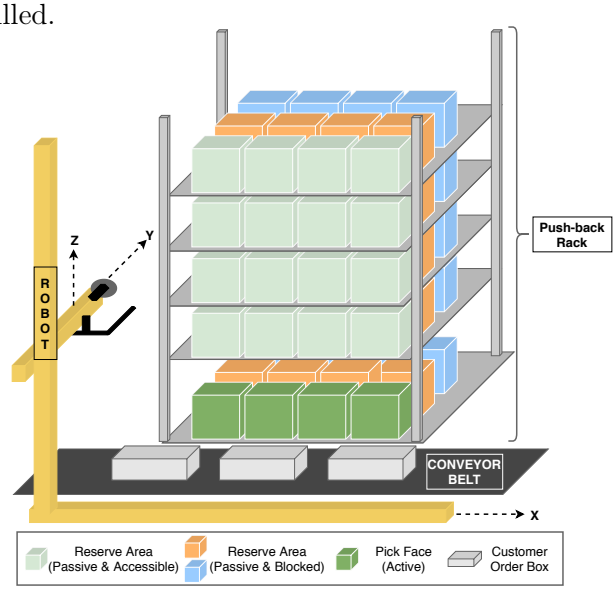


Figure 1.2: Schematic layout of the robot-based compact storage and retrieval system

1.2 Project Goal

This project is an extension to the work of Nataraja et al. (2022). Their study focused on answering two questions: (1) Which orders should form a batch and (2) in what sequence should the batches be picked? These questions are answered in a static environment. When the environment is considered dynamic, a new question arises: (3) When should a batch be scheduled? To accommodate this third question we formulate two new questions to answer simultaneously: (I) When should we pick a batch and (II) what orders should be in this batch? We address this problem as Online Order Batching Problem (OOBP). To the best of our knowledge, no research has been done that answers these questions combined using machine learning. Furthermore, limited literature is present that addresses the first question in particular. Answering these questions in an environment with fluctuating demand is also rarely found in the literature.

In this project, we strive to optimise the picking process so that both earliness and tardiness are minimised. Both earliness and tardiness have their own cost related to them. Earliness might lead to holding cost, deterioration of perishable goods and opportunity costs. A tardiness penalty is often related to customer compensation (Akyol and Bayhan, 2008).

To summarise, the primary goal of the project is to create a batching policy that can handle varying demand in a dynamic environment and can make decisions on-line.

1.3 Research Questions

This study aims to explore if a machine learning agent can help solve the OOBP. This goal can be formulated as the following research question:

How can the introduction of a machine learning agent optimise the on-line batching for an automated picking system?

In this study, we will measure performance in terms of earliness and tardiness. This means that optimising the performance means minimising the sum of weighted earliness and tardiness. This research question will be answered by the use of the following sub-questions:

1. *What are the existing methods for solving the problem of on-line batching?*
This question aims to gain more insights into the topic and get a broader understanding of the subject. We will answer this question through a literature study. The review will be two-fold. First, we will review traditional approaches such as exact methods and heuristics. Second, we will identify machine learning algorithms suitable for our problem.

2. *How does each of the decisions contribute to the overall objective?*
The agent has to make two decisions: (1) what orders to place in a batch and (2) at what point in time the batch should be released. With this sub-question, we want to find the impact of each decision on the system's performance. This will create a better understanding of the working of the agent. It will validate if the agent indeed optimises both decisions. We will answer this question with a full factorial analysis.

3. *How does the batching agent perform compared to current practices?*
Old practices will be defined based on the first sub-question. Once these practices are defined, they will be used as benchmarks to see how the agent performs in comparison to traditional solutions. This benchmark is needed to make a valid statement about the actual performance of the agent.

4. *How does the batching agent adapt to changes (e.g. seasonal trends)?*
E-fulfilment centres have to deal with highly varying demands. Days like Singles Day or Black Friday create short periods of high demands. Other events, like Christmas, might change the demand for a longer period. The system must keep reasonably high performance while undergoing these fluctuating demands. Therefore, this needs to be tested. We will conduct a sensitivity analysis to answer this question.

2 Background Information

In this section, we provide additional information necessary to understand the context of the project. First, an explanation of the difference between a dynamic and static environment is given in Section 2.1. This is followed by a summary of the previous work done on this project. Lastly, we discuss the different algorithms of Reinforcement Learning and the specific policy used for this project, Proximal Policy Optimisation (PPO).

2.1 Static and Dynamic environment

It is vital to differentiate between a static and a dynamic environment. In a static environment, all orders are known before the planning period. For example, at the beginning of the day, all orders that need to be picked and packed that day are known. While this might have been a realistic scenario in the past, with current e-commerce trends, where next day, or even same-day delivery is the standard, this is an almost Utopian scenario. E-fulfilment centres have a much more dynamic environment, i.e., orders arrive continuously throughout the day. This means that scheduling upfront is an almost impossible task and in most cases, on-line decision making is the best solution. For this study, a dynamic environment will be considered.

2.2 Previous Work

This thesis is part of a PhD project of Bhoomica Nataraja. As part of this project, Nataraja already presented a study where effective and efficient algorithms are provided which are used to integrate and solve the four main operational order picking problems; order batching, order positioning, retrieval sequencing, and bin relocation (Nataraja et al., 2022). A static environment is considered, i.e., the orders are known in advance at the beginning of the planning period. The minimisation of the retrieval time depends on how

1. the customer orders are grouped together (order Batching)
2. the order boxes are arranged on the conveyor belt (order Positioning),
3. the robot is routed in order to retrieve the items of each customer order (retrieval Sequencing), and
4. the storage bins are relocated for feasible retrieval (bin Relocating).

This first problem, the Order Batching Problem (OBP), is the main topic of this thesis and will therefore be discussed further in this section. This OBP comprises of two sub-problems: (i)Batch composition - which orders should be assigned to which batch, and (ii) Batch sequence - in which order should these batches be picked. To deal with the first sub-problem, the following two heuristics can be used:

- **H1 Most-Common SKUs (MCS) policy:** This rule first assigns an order to the batch based on the arrival order. Next, it searches for the orders that have the most similar SKUs with the orders that are already in the batch. If needed, a tie is broken based on arrival sequence. If no more similar orders are found, the first remaining order is added and the search continues. Once a batch is completed, the rule starts over and creates a new batch.
- **H2 Seed Algorithm:** This rule is a combination of the Greatest Number of Sorting (GNST) rule and the Greatest Number of Identical SKUs (GNIS) (Zhang et al., 2017b). First the seed order of a batch is selected based on the GNST rule that selects the order

that has the largest number of items. Next the GNIS rule is used to fill the batch, assigning order based on the greatest number of identical SKUs as the seed order. All orders in a batch cumulatively form the new seed order. If no similar order can be found, an order is chosen based on the GNST rule and the search continues.

To find the optimal sequence of the batches the following heuristic is proposed that will follow **H1** or **H2**:

- **H3 Earliest due-time with Order-similarity sequencing:** Batches are prioritised based on the due dates of the orders. If there are batches with similar due-dates the batch with the most similar SKUs presents in the current batch is chosen.

The above mentioned heuristics can be combined and used in sequence where first the content of the batch is decided upon and next the order in which the batches will be picked is chosen. The two heuristics mentioned below make these two decisions simultaneously.

- **H4 State Of Rack (SOR):** With this rule the orders with the most similar SKUs with the pick face of the rack are assigned to a batch. If no more similar SKUs are found, the first remaining order is added to the batch. Once a batch is finished, the policy simulates the picking of the batch to obtain the next state of the rack.

- **H5 SOR + MCS:** Here, the first order of a batch is chosen based on the SOR rule. Then, this order will start as the seed-order and the batch is filled using the MCS rules by assigning orders that are most similar to the seed. The new seed is formed by the cumulative addition of the orders in the batch. If no orders are found using the MCS rule, the SOR rule is used again.

For the other mentioned problems, different heuristics are developed. Different combinations (multi-heuristics) of these heuristics are created and tested. When compared on average number of relocations and total retrieval time, the multi-heuristic containing **H5** is considered the Best Multi-Heuristic (BMH). When tested on tardiness, a multi-heuristic containing the combination of **H1** and **H3** and a multi-heuristic containing **H2** and **H3** out-preformed the previously mentioned BMH, although this difference is minimal. When comparing the rules in a case study, the BMH reduced the retrieval time by 33.2% on average for a real world cosmetic warehouse compared to the current (First In First Out (FIFO)) practice.

2.3 Deep Reinforcement Learning

In the field of machine learning, there are three types of algorithms; supervised, unsupervised and reinforcement learning. Supervised learning algorithms learn from datasets consisting of both an input and the desired output. The algorithm will try to predict based on the input data, and parameters are adjusted based on the comparison between the prediction and the desired output. Unsupervised learning algorithms work with only a dataset consisting of input for the algorithm. Unsupervised learning is most commonly used for clustering data in unlabelled clusters. With Reinforcement Learning (RL), an algorithm interacts with the environment. Based on the changes caused by the agent, a reward is given to the algorithm so that it can learn. RL is designed for situations where no exact solution is known, but some states can be seen as better or worse than others. The goal of RL is to navigate these states so that a maximum cumulative reward is achieved.

In recent years, RL has grown in popularity due to its successes in solving tasks such as; playing Go (Silver et al., 2016), Atari (Mnih et al., 2013) and numerous other video games (Vinyals et al., 2017; Silva and Chaimowicz, 2017; OpenAI et al., 2019). Mnih et al. (2015) caused the field of Reinforcement Learning to show significant improvements by combining Reinforcement Learning with deep learning and creating Deep Reinforcement Learning (DRL).

2.3.1 Introduction to Reinforcement Learning

In the field of Reinforcement Learning, the terms model and agent both describe the algorithm that acts as the decision-maker. This agent learns by interacting with the environment. The agent "observes" the environment as a *state* S_t. This state is a representation of the environment at time t. Based on this state, the agent takes an *action* (A_t), causing the environment to transition into a new state S_{t+1}. Based on these changes, a *reward* (R_t) is calculated. This reward is used as feedback for the agent. The agent will then change its policies to optimise the cumulative reward. This agent-environment interaction is shown in Figure 2.1.

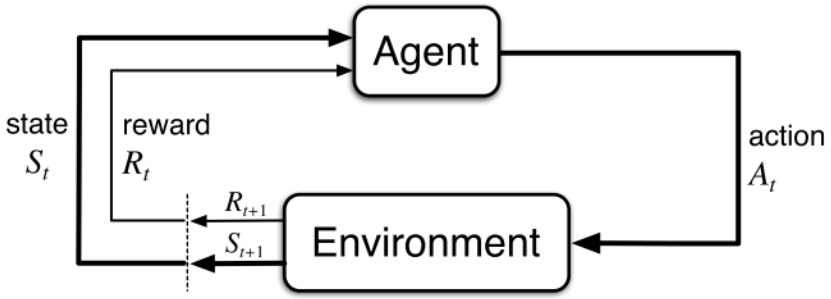


Figure 2.1: The agent-environment interaction (Sutton and Barto, 2020)

In order to solve RL problems, this framework needs to be represented as a Markov Decision Process (MDP). MDPs are a classical formation of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations or states, and through those, the future rewards (Sutton and Barto, 2020). All MDPs have to satisfy the Markov property (Sigaud and Buffet, 2013); it has to be a stochastic process that is memory-less. A specific form of the MDP is the Semi-Markov Decision Process (SMDP). An SMDP is a generalised form of the MDP that allows for transitions to occur in continuous irregular times (Khodadadi et al., 2014). These SMDPs can be represented by a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \tau, \gamma >$. This representation contains:

- set of states $\mathcal{S}$
- set of actions $\mathcal{A}$
- set of rewards $\mathcal{R}$
- set of transition probabilities $\mathcal{P}$
- set of transition times τ
- discount rate γ

As stated, the agent's goal is to maximise the cumulative reward, which means that the agent tries to take actions in a way that maximises the sum of the future rewards, otherwise known as the *expected return*, denoted by G_t. These rewards that create the return are gathered

by the agent interacting with the environment. Equation 2.1 denotes the simplest form of the return where it is just calculated by the sum of the rewards.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.1}$$

This interaction will continue until a *terminal state* (T) is reached. The time window from the start until the terminal state is referred to as an *episode*. Episodes are often naturally and logically defined sub-sequences, such as a game of Pac-Man that ends when the player either wins or dies. After each episode, the environment is reset to an initial state, independent of the previous terminal state.

However, such an episode is not always clearly defined. In such a case, the agent-environment interaction could continue forever, causing G_t to go to infinity. To deal with this, *discount* (γ) is introduced. This discount is given to future rewards. The further the rewards are from now, the more discount the rewards receive. Equation 2.2 shows the calculation of the *expected discounted return*, here $\gamma \in [0, 1]$ is called the discount rate.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

The agent will select the action that will yield the maximum discounted return. This action selection is done based on a policy, π. A policy is essentially a set of probabilities denoted by $\pi(a|s)$, the probability of taking action a when the environment is in state s. The agent uses this policy to predict the return with the help of *value functions*. Equation 2.3 gives the expected value of G_t for a given state s. This makes v_π the state-value function for policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S} \tag{2.3}$$

Equation 2.4 calculates the action-value function q_π for policy π. This function calculates the value of action a in state s under the policy π by calculating the expected return when action a is taken from state s.

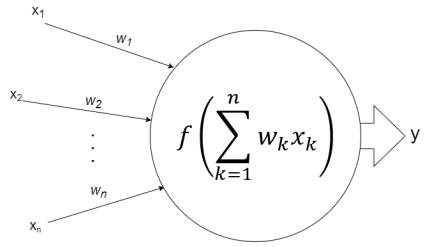
$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.4}$$

An agent will select the action based on these functions. After interacting with the environment, the agent will receive a reward. If this reward is out of line with the expected reward, the policy will be updated to align the expected and received reward.

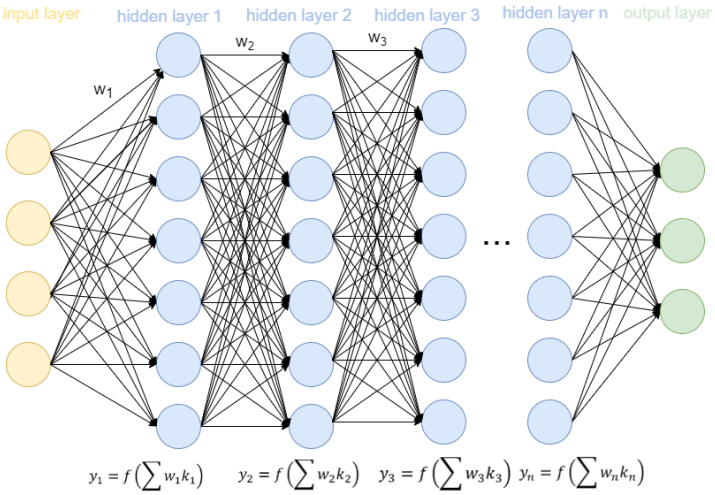
2.3.2 Neural Networks

A Neural Network (NN) is a network consisting of different layers containing multiple processing units (called neurons or nodes). Each neuron can have an arbitrary number of neurons connected to it. These connections are either an input of the neuron or an output. NN are typically organised by layers, where neurons in one layer form the input of the neurons in the next layer. A neuron aggregates all the signals passed through it from its input neurons from the previous layers. This aggregated signal is then passed through an activation function that will determine the neuronal behaviour. This function will "activate" if the aggregated signal is strong enough, forwarding these values to the neurons in the next layer. The output of a neuron will be multiplied with a weight. A single layer NN consisting of only an input and an output and is called a

perceptron, as shown in Figure 2.2a. A NN can be extended by placing "hidden layers" between the input and output layers. A Multilayer Perceptron (MLP) is an NN consisting of at least one hidden layer, and all layers must be "dense" layers (otherwise known as "fully connected" layers). MLP is a traditional version of NN that only uses feed-forward between layers. By adding self-loops for layers, Recurrent Neural Network (RNN) can create an artificial memory in the network. Together with Convolutional Neural Network (CNN), MLP and RNN form the three most common types of NN. All these types of networks have at least one hidden layer. When two or more hidden layers are present in NN, it is seen as Deep Neural Networks (DNN) (Dong et al., 2020) (Figure 2.2b).



(a) Basic principles of a perceptron



(b) basic principles of a DNN with n dense hidden layers

Figure 2.2: Deep Neural Networks explained

For the training of a DNN, a loss function is defined between the desired and computed output. The objective is to minimise the loss. With the use of a gradient descent method, the weights in the network are updated to reach this objective, using partial derivatives and back-propagation.

In the work of Mnih et al. (2015), DNN are combined with RL to create the first version of what is now known as Deep Reinforcement Learning. The input of the NN is a representation of the state. The output layer of the NN is a representation of the actions space. The network tries to predict which action it should take. This training is done based on the value function, where the reward is used in the loss function to update parameters.

2.3.3 Deep Reinforcement Learning Models

In Reinforcement Learning, problems can be described as either an MDP or a (multi-armed) bandit. The latter is a simple problem description where the probability of (reward) payout is calculated without any state dependencies. Since no related methods will be used in this project, multi-armed bandit will not be further explained. More information can be found in Sutton and Barto (2020). From here on, all discussed reinforcement learning problems and methods are related to MDP unless stated otherwise.

Within this MDP group, a distinction can be made between model-based and model-free methods. Model-based methods try to predict the often unknown values of $\mathcal{P}$ and $\mathcal{R}$ in the MDP depiction. The agent takes actions to create samples of the environment. Once all values are known, these can be directly used with planning methods. Model-free methods do not try to model the environment, but instead, they try to find an optimal policy. So, instead of focusing on the model, these methods attempt to find the optimal reward directly. Model-based methods are less efficient, or even unfit for problems with larger action- and states-spaces. When the dimensions of these spaces increase, it will drastically increase the computational power and time needed for the model to function, making it an unfit approach for this project. Therefore, only model-free methods will be considered.

Model-free methods can again be divided into two main categories; (1) value-based and (2) policy-based methods. An additional sub-category exists, which combines the two methods, (3) actor-critic (AC).

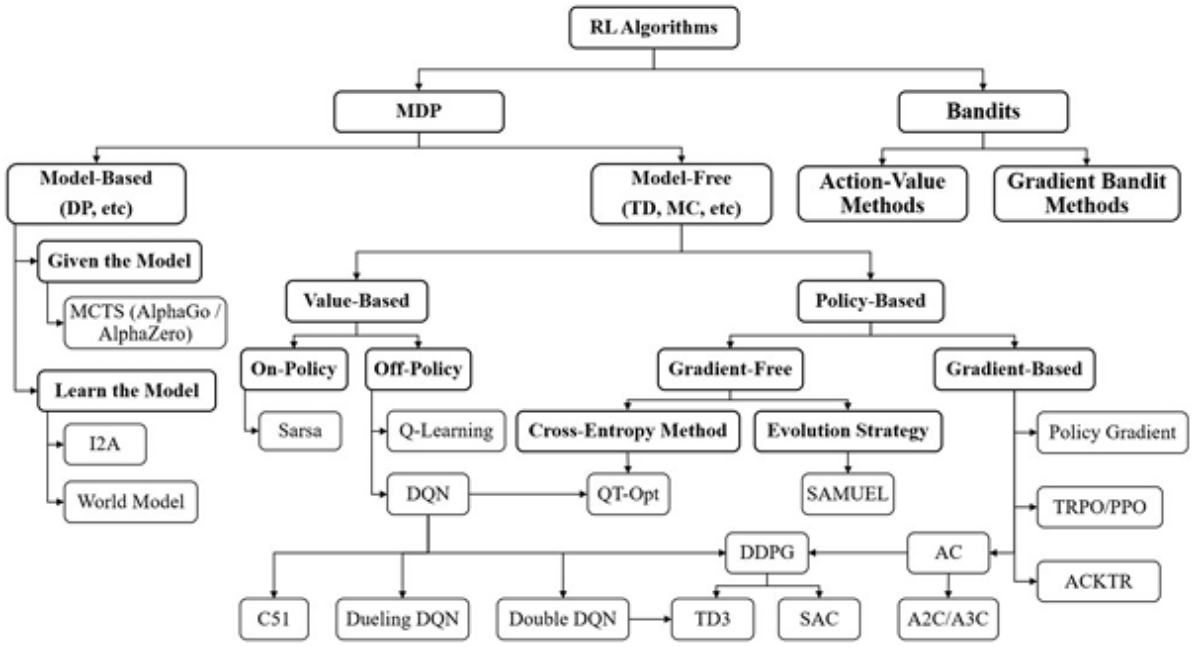


Figure 2.3: Map of reinforcement learning algorithms. Boxes with thick lines denote different categories, others denote specific algorithms (Dong et al., 2020)

Value-based

Value-based methods try to optimise the action-value function $Q^{\pi}(s, a)$ to find the optimal value function $Q^{\pi *}(s, a)$. One of the oldest value-based methods is Q-learning, where an agent tries to learn the expected discounted reward (Q-values) (Watkins and Dayan, 1992). Mnih et al. (2015) introduce Deep Q-Network (DQN) as a more stable form of Q-learning. According to Mnih et al. (2015), Q-learning experiences instability during training. Thus, the authors employ the techniques "experience replay" and "iterative updates of the action-values towards target values" to make DQN a more stable method. Experience replay selects random samples from a large dataset with all the experiences gathered over time rather than learning from consecutive samples when updating the Q-values. This approach allows for better data efficiency since samples are potentially used in many weight updates. The random samples also remove correlations that are present with consecutive samples.

From DQN many variants followed, like Double DQN (van Hasselt et al., 2015), Dueling DQN (Wang et al., 2015), Noisy DQN (Fortunato et al., 2017) and DQN with Prioritised Experience Replay (Schaul et al., 2015). Double DQN reduces overestimation of Q-values. This is done by using two separate neural nets for action selection and evaluation. Dueling DQN aims for more robust learning by decoupling values that do not depend on the action by using an advantage function combined with state values, instead of the normal Q-values. Noisy DQN provides new technique for exploration, by adding noise to the predicted Q-values. Lastly, DQN with Prioritised Experience Replay method tries to optimise the experience replay step by prioritising samples around large changes in the the state value. In the work of Hessel et al. (2017), different combinations of these methods are tried and some have shown promising results.

Policy-based

Policy-based methods focus directly on the policy by updating the policy iteratively until the return is maximised. Policy-based methods are computationally more efficient than value-based methods since only a policy needs to be updated and not all state action pairs. This is especially noticeable with larger state and action spaces. One way this updating can be done is by using policy gradient optimisation. This method uses sample trajectories to obtain an estimator of the gradient of the expected return. The most commonly used gradient estimator ($\hat{g}$) is given in Equation 2.5 (Schulman et al., 2017). This gradient is obtained by differentiating the objective (loss) function ($L^{PG}(\theta)$) given in Equation 2.6.

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) \hat{A}_t \right] \tag{2.5}$$

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta \left(a_t \mid s_t \right) \hat{A}_t \right] \tag{2.6}$$

Here, θ represents the vector of policy parameters and $\hat{A}_t$ is the estimator of the advantage function at timestep t. The calculation of the advantage is shown in Equation 2.7 where V is the state value function and Q is the action-value function (Dong et al., 2020). The advantage represents how much better or worse an action or state is than the initial estimate.

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \tag{2.7}$$

In the work by Schulman et al. (2015), a new objective is proposed that in most cases performs better and is more robust than this 'vanilla' policy gradient. This new method, called Trust Region Policy Optimisation (TRPO), uses the objective shown in Equation 2.8. When maximising this objective, the optimiser is constrained to Equation 2.9. This constraint prevents the new policy from diverging too far from the old policy by limiting the Kullback-Leibler (KL) divergence to a specific value δ. This KL divergence is a measurement of how different two probability distributions are (Dong et al., 2020).

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta \left(a_t \mid s_t \right)}{\pi_{\theta_{\text{old}}} \left(a_t \mid s_t \right)} \hat{A}_t \right] \tag{2.8}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t \left[\text{KL} \left[\pi_{\theta_{\text{old}}} \left(\cdot \mid s_t \right), \pi_\theta \left(\cdot \mid s_t \right) \right] \right] \leq \delta \tag{2.9}$$

Due to the policies not diverging much, the optimisation is more robust. Without these constraints, large losses could cause extensive policy updates. TRPO has good performance in multiple fields. However, the computational costs are high, and the implementation is difficult. These shortcomings lead to the creation of Proximal Policy Optimisation (PPO) (Schulman et al., 2017).

The idea behind PPO is the same as with TRPO; prevent destructively large policy updates. However, PPO defines this goal in a surrogate objective function that is notably less complex (Equation 2.10).

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{2.10}$$

Here, $r_t(\theta)$ is the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. ϵ is known as the clipping parameter. In Figure 2.4, it can be seen that by the use of this clipping factor, the probability ratios will not become too large or too small for a positive or negative advantage, respectively.

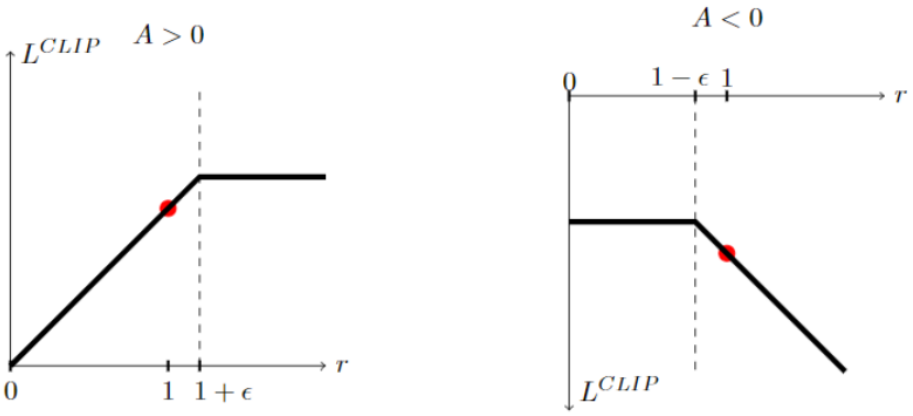


Figure 2.4: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimisation, i.e., $r = 1$. (Schulman et al., 2017)

When using a neural network where both the policy and value function are predicted with the same network, a loss function that combines the policy surrogate and a value function error term must be used. This value function loss is calculated with a squared error loss $\left(V_\theta(s_t) - V_t^{\text{targ}}\right)^2$. Sufficient exploration should also be insured by adding an entropy term, S. This all combined with two coefficients c_1 and c_2 gives the total objective function Equation 2.11.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{2.11}$$

The PPO algorithms have an actor-critic style design. First, the policy is initialised. With this policy, one or more actors will take actions for a set amount of time steps, during which the advantages will be computed. Meanwhile, the value function (critic) used for this advantage will be updated based on the experiences. Even though this value function is updating, the actor takes decisions based on the policy, so the state-dependent behaviour is not changing. Once the actors are done, the calculated advantage is then used in an optimisation step with the surrogate objective function. Based on this, the policy will update, and the actors will again take their timesteps to gather the advantage needed for the next policy update.

2.3.4 Applications of DRL

Applications of DRL in the warehousing and shipping industry vary in a wide range. DRL is commonly applied for routing autonomous vehicles. For example Li et al. (2019) solve a routing and dispatching problem with a DQN and Oxenstierna (2019) solves a routing problem with

the use of a Monte Carlo Tree Search CNN. Peyas et al. (2021) use multi-agent Q-learning for navigation and object avoidance with multiple robots in a 2D environment.

Other implementations are also found, for example online route assignment (Zeng et al., 2021), lot scheduling (Rummukainen and Nurminen, 2019) or designing inventory policies (Boute et al., 2022; Kaynov, 2020; Geevers, 2020) using PPO. Other studies that use PPO with topics closely related to our project are those of Cals et al. (2021) and Beeks et al. (2022).

2.3.5 Suitability PPO

PPO is known for its balance between computational efficiency and sample efficiency. With the use of importance sampling, PPO reuses collected data. With this, PPO addresses the problem of most model-free RL algorithms being highly data-intensive, meaning that a lot of data needs to be created for the agents to learn. This, combined with the easy implementation of the algorithm, and its proven success in related fields (see the previous section), makes PPO a fit solution for our project.

3 Literature Review

In this section, we discuss the literature related to this study. We will provide literature on the Order Batching Problem (OBP) and Online Order Batching Problem (OOBP). For these problems, we explain how they are solved in literature using heuristics, multi-heuristics, algorithms and machine learning methods.

3.1 Order Batching Problem

When multiple orders are demanded, it is expected to pick all of these orders with a minimum cost (Cergibozan and Tasan, 2019). This cost function is often expressed as service time or tardiness. The problem of minimising this cost by the means order batching is defined as the Order Batching Problem (OBP). The OBP is excessively discussed in the literature. In Cergibozan and Tasan (2019), 64 papers on the OBP and order picking process are reviewed. Approximately half of these papers are related to the OBP. Most of the solutions in these papers consider heuristics or meta-heuristics, and the others are spread out over simulations, exact approaches, and data mining approaches. Most of the reviewed papers consider a static environment, only a few are related to a dynamic environment, which is discussed in Section 3.2.

Only two papers mentioned in Cergibozan and Tasan (2019) aim on solving the OBP with the focus on both earliness and tardiness. Elsayed et al. (1993) developed a batching and sequencing procedure to optimise just in time order retrieval in a ASRS. Tsai et al. (2008) add an additional objective of minimising the travel cost of the picker. They propose a multiple generic algorithm consisting of batching algorithms that creates a batch based on these three objectives (earliness, tardiness & travel cost), and a travel algorithm that finds the optimal path for this batch. Another paper that focuses on both earliness and tardiness is the work of Hall and Posner (1991). They define the problem of job scheduling to minimise the weighted sum of earliness and tardiness for jobs with a common due date as an NP-complete problem and give a solution. In Hall et al. (1991), the problem is solved again for unweighted tardiness and earliness.

Other works that consider scheduling are Muter and Öncan (2022) and Hung et al. (2017). The first paper provides different algorithms that can find optimal solution for order batching and scheduling with regard to travel time and total makespan of the pickers. The latter provides multiple mixed integer programming solutions to optimise job scheduling in an make-to-order manufacturing environment, for both earliness and tardiness. A problem closely related to scheduling is sequencing. Jiang et al. (2018) suggest a seed algorithm to solve this order batching and sequencing problem.

3.2 On-line Order Batching Problem

As mentioned in Section 2.2, Nataraja et al. (2022) give a multi-heuristic approach for solving the OBP, with a focus on the case of Pickr.AI. These static solutions are not suitable for a dynamic environment since all these solutions assume that all orders are known before the planning period. Using static batching policies in a dynamic environment would lead to sub-optimal results since the policies are designed to deal with large queues with little uncertainty. In a dynamic environment, the queue is either small or non-existent at some points, and since not all orders are known in advance, many uncertainties exist.

When the Order Batching Problem needs to be solved in a dynamic environment where orders continuously arrive in a system, the problem is defined as the Online Order Batching Problem (OOBP). Van Nieuwenhuyse et al. (2007) consider a manual, multi-server, order picking

process in a dynamic environment. They consider a pick-and-sort system rather than the more standard sort-while-pick system. The paper tries to find an optimal batch size for the picking and sorting-and-packing process that minimises the average order throughput time.

Another popular batching method found in literature is the time window batching. This can be split into two types: Variable Time Window Batching (VTWB) and Fixed Time Window Batching (FTWB). In the case of VTWB, the picker waits a varying amount of time, dependent on how long it takes to reach a desired pick batch size. FTWB, however, waits until a specific time has passed before starting to pick the batch, regardless of the batch size. In Van Nieuwenhuyse and de Koster (2009), both methods are used to study the responsiveness of a warehouse. They consider random customer order sizes, consecutive sorting and packing processes, multiple server picking and sorting operations and general setup and service time distributions. This study follows from the previously discussed study (Van Nieuwenhuyse et al., 2007) and does not introduce new methods on how to solve the OOBP. There are other studies that use time window batching methods (Ozturk et al., 2017; Zhang et al., 2016; Leung et al., 2020; Gil-Borrás et al., 2020; Pinto and Nagano, 2019), and each of these studies have shortcomings regarding the ability to adapt to highly variable environments with changing arrival rates.

Zhang et al. (2017a) recognise the shortcomings of VTWB and FTWB in fluctuating order environments. When the arrival rate is relatively high, many customer orders will pile up in the queue, waiting to be picked with a fixed time interval in the case of FTWB. On the other hand, if the arrival rate is relatively low, orders might have to wait for a long time to form a batch in the case of VTWB. Therefore, they suggest a hybrid rule-based algorithm using multiple FTWB rules. This algorithm solves the Online Order Batching and Sequencing Problem with Multiple Pickers. This problem is a specific variant of the OOBP where, besides only creating the optimal batch and finding the optimal time to release the batch, it also focuses on the order that the batches should be released in and assignment of the batches to the pickers. The study's goal is to minimise the maximum completion time of an order. This algorithm is tested against the standard FTWB and VTWB practices and outperforms both. It is also shown that the correct assignment of a heuristic improves the picking efficiency, and the employees' work availability and employment cost and facility cost can be reduced.

In Gil-Borrás et al. (2021), this same variant of OOBP with multiple pickers is solved with the use of two heuristic approaches. These approaches are combinations of a multistart procedure with a metaheuristic called variable neighbourhood descent. The performance of the algorithms is tested with three different objective functions: minimising the total picking time, minimising the maximum completion time, and minimising the differences in workload of the pickers. The proposed heuristics have proven favorably when compared with other methods in the literature, like previous mentioned work of Zhang et al. (2017a)

Time window batching is found to be the commong method to deal with the OOBP. This can also be concluded from the work of Henn et al. (2012). In their literature review on solutions to the problem, four out of the six pieces of research used a time window batching approach. One of the approaches that do not use time window batching is proposed by Henn (2012). The author aims to modify solutions for a static batching problem into a solution that fits the dynamic situation to improve warehouse efficiency. Their approach is to fit an off-line solution into an on-line algorithm. They do this based on three types of decision points: a picker becoming available when there are still orders waiting to the processed, an order arriving when there is a picker available, and the arrival of the last order. The idea is that the online algorithm solves the OBP off-line with all known open orders at each decision point. They compare three different

static batching heuristics that can be combined in the algorithm with some selection rules. After running a numerical simulation, they found two algorithms that outperformed their benchmark.

In the work of Pérez-rodríguez and Hernández-aguirre (2015), the authors recognise that most studies considering OOBP do not consider relationships or interactions between customer orders. They offer a solution in the form of a continuous estimation of distribution algorithm.

In all these studies, no work is found that considers a dynamic environment, variable arrival-rates, dependency between batch content and pick time, and have the objective of minimising both earliness and tardiness. Most of the considered works only cover one or two of these topics.

3.3 Order Batching and Machine Learning

Machine learning is a powerful tool and has already proven to be successful in solving job scheduling problem, which is closely related to OBP and OOBP (Xie et al., 2019; Drakaki and Tzionas, 2017; Paternina-Arboleda and Das, 2005; Lang et al., 2020; Aydin and Öztemel, 2000; Rummukainen and Nurminen, 2019). These studies look at machine learning practices as a solution since more traditional methods only offer high-quality solutions at the cost of high computation times, which is unacceptable in current practices with more extensive production diversity, shorter lead times and more significant uncertainties in the supply chains. In these environments, a method is needed that can compute a suitable solution in a short amount of time. However, limited literature has been found on machine learning for solving the OBP. This can also be seen in Cergibozan and Tasan (2019), which is a literature review on OBP solutions. Only a handful of studies use data mining approaches in the form of a clustering algorithm for off-line batch creation.

One study that does use machine learning is by Zhang et al. (2019). A sequential decision-making process is addressed where a decision-maker needs to decide on (1) whether to start processing a batch or wait for more jobs to create a larger batch and (2) which batch should be processed first. The first question considers the trade-off between lower machine utilisation with larger batches and lower capacity with smaller batches. The second problem is in place since the study considers a multiple job family environment; therefore, multiple batches are created. Most decision rules proposed in literature for solving these problems are specific methods and not adaptable to the changing situations in a real-life job shop. Therefore, the authors suggest a machine learning approach with Reinforcement Learning (RL). The idea of RL is that a decision-maker (or agent) is learning how to behave in different situations by interacting with the environment.

The goal of the learning agent is always to maximise a reward or minimise a penalty. The agent is not told what actions to take or what actions will result in a reward, but rather it will undergo a trial-and-error search where it will learn the most rewarding actions. For the agent to solve the given problem, it must be defined in a Markov Decision Process (MDP). The MDP consist of two components: a decision-maker and its environment. The state (of the environment) is observed at discrete points in time so that the decision-maker can make decisions at these points based on the state. Once a decision is made, the environment returns a reward to the decision-maker. The goal of the agent is to maximise the long-term cumulative reward.

In Zhang et al. (2019), the MDP is described by a five-tuple; $MDP{=}\langle T,\ S,\ A(s),\ P(s'|s,a),\ R(s'|s,a)\rangle$. T is a set of points in time the agent makes decisions (decision epochs), S is a set of all possible states of the environment, $A(s)$ are the possible actions in the state $s, s \in S$, P is the set of probabilities that a state change from s to s' happens after taking action a and R are the

rewards corresponding to these state changes and actions. Real-time batching is considered as a Semi-Markov Decision Process (SMDP). This is a generalised form of the MDP that allows the state transitions to occur in continuous irregular times (Khodadadi et al., 2014). Since the transition probabilities are impossible to know, the selected RL technique is Neural Fitted Q Learning. This Q Learning is a model-free technique, meaning that no model details, like transition probabilities, are needed. It does leave one problem; the state-space for real-time batching is infinite. To solve this, a parameterised value function is created to obtain the values of unexplored states. They simulated with their agent and compared it with three conventional batching rules combined with a first-in-first-out sequencing rule. In general, the agent performs better than the conventional techniques.

Cals et al. (2021) proposes another solution that uses reinforcement learning. The specific technique used is called Proximal Policy Optimisation (PPO). To make PPO work, an SMDP must be defined; this is done in the same way as Zhang et al. (2019). However, the approach differs in the formulation of T. In the case of Cals et al. (2021), this is defined as the time to transition, which is almost the same as in the previous section. However, now the time is defined by a simulation. If the agent decides to wait, the waiting time is not fixed, but the time is set to wait until the next state transition happens, which could be the arrival of a new order or a picker becoming available. Besides this subtle difference, the implementation of PPO is the same as Zhang et al. (2019). This technique, just like the Q-learning method in Zhang et al. (2019), does not need to know the probabilities of state changes in the SMDP, and the events are almost similar. Cals et al. (2021) gives the agent two options - pick-by-order or pick-by-batch.

Zhang et al. (2019) and Cals et al. (2021) differ in the considered environments. The former considers a job shop with different job families, while the latter considers a warehouse consisting of picker-to-good (automated) and good-to-picker (not automated) parts with only one job type. Another entirely different part is the simulation.Cals et al. simulates two different scenarios and compares these scenarios to the best batching and sequencing rule (best tested out of 20 combined batching and sequencing rules). The scenarios can be described as low and high arrival scenarios. Both scenarios are tested for 330, 400 and 500 orders that arrive in one hour to see how the agent handles varying arrival rates. The agent outperforms the heuristic in the first scenario significantly, while in the second scenario, this only happens for 500 orders with a slight difference. It is also checked how the agent performs with 400 orders when it is trained on 500 and the other way around. This is compared with an agent trained and tested with the same amount of orders. The results show no significant differences, and it can be concluded that the agent performs well under changing arrival patterns.

4 Methods

As mentioned before, the goal of this project is to create a decision-maker that decides (1) if orders need to be picked and (2) what orders need to be picked at that point in time. These decisions are made simultaneously by a decision-maker. Based on the decision either a waiting action or a picking action need to be executed.

The second decision makes the problem drastically more complex. This significantly increases the computation time needed for the agent to learn. This allows for less exploration of different reward functions and parameters in the observation-space. For this reason, we start by using a simple version of the agent (agent 1), where the second decision is reduced to two batching heuristics that the agent needs to choose from. Since the behaviour of the agent is now reduced to only three different actions (wait, heuristic 1, heuristic 2), the agent's behaviour can be better examined and reward functions and other parameters can be tuned more easily. Based on the information gathered from this simple agent, 2 more complex agents are created, where the agent can also select how many orders it wants to batch (agent 2), and instead of a heuristic selects the orders for a batch by itself (agent 3).

In this section we will first discuss the discrete event simulation that we have created in Section 4.1. Next we discuss how we define the Markov Decision Process in Section 4.2. Lastly, we explain the details of the implementation of the agent in Section 4.3.

4.1 Simulation

Since the systems of Picker.AI are still in development, real-live testing and analyses are not possible. Instead we make use of a discrete event simulation. This simulation is created with the use of SimPy (2020), a process-based discrete-event simulation framework based on standard Python. The simulation consists of three main components; **Order generation, Order arrival** and **Order picking** and can take two input actions; *wait* and *pick*. The pick action also means that decision-maker gives the necessary information to create a batch. A diagram of how the simulation works is shown in Figure 4.1. In this diagram, the function of *Create Batch* is different per agent. In Appendix A.1, the batch creation of the most complex agent is explained in more detail.

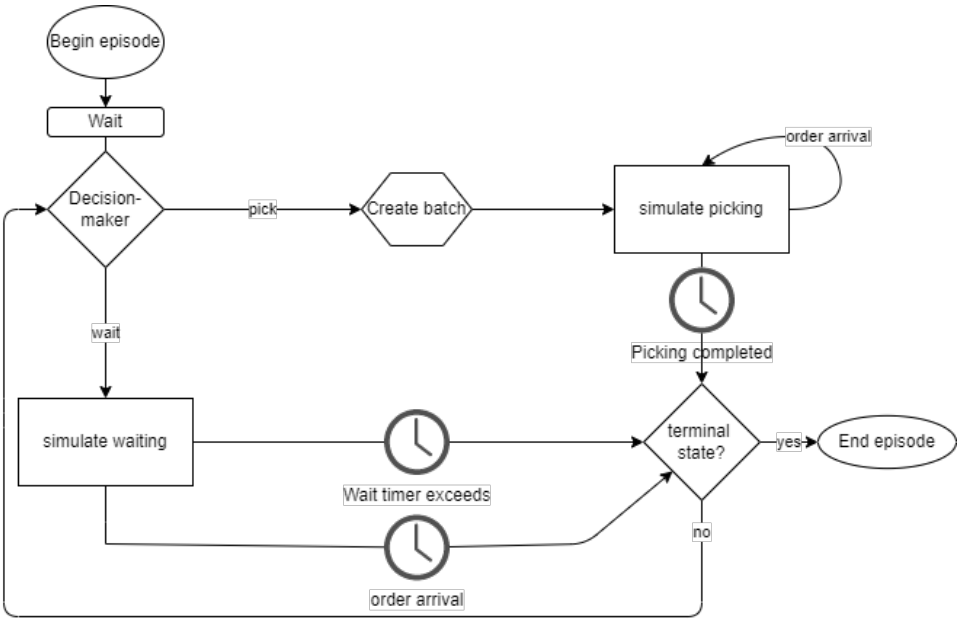


Figure 4.1: Process flow diagram of the simulation model

4.1.1 Action Simulation

The simulation starts in what we call a "decision moment". This is a state in which no time will pass, and the input of the decision-maker is needed to continue the simulation. The corresponding action will be simulated once the decision-maker has selected to perform either a *wait* or a *pick* action.

In Figure 4.2, the basic working of the action simulation is shown. If a waiting action is simulated, the discrete event simulation will run until a new order arrives. This arrival will trigger a new decision moment. Another way this moment can be triggered is by using a timer. When the simulation environment exceeds the fixed length of timer, a decision moment is encountered. This is implemented to make the behaviour of the wait-action more predictable since this timer introduces an upper-limit on how long waiting actions last. This predictability should make it easier for RL to learn. An alternative would be to either only use a fixed timer or only use order arrival or the picker becoming available as a trigger for a decision moment (Cals, 2019; Beeks et al., 2022). Only using order arrival is undesirable when dealing with low arrival rates as a wait-action would mean that the earliest next option to pick an order would be with the arrival of a new order, and this might be past the due date. A fixed waiting time might solve this problem for low arrival. However, when dealing with an environment that has a relatively high arrival rate, this might mean that orders arrive while waiting, which might cause the decision-maker not to be reactive enough and cause unnecessary tardiness. When dealing with an environment with changing arrival rates, either of these solutions will cause issues. Therefore, a combination of these methods is selected. One other option that has been considered is to let the decision-maker also decide on the duration of the waiting action. However, this would drastically increase the complexity of the problem, and is therefore not included in the scope of this project.

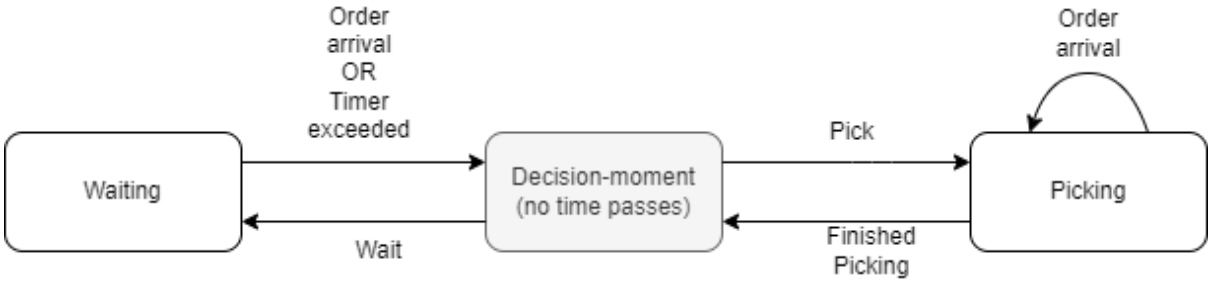


Figure 4.2: Simplified working of simulation actions

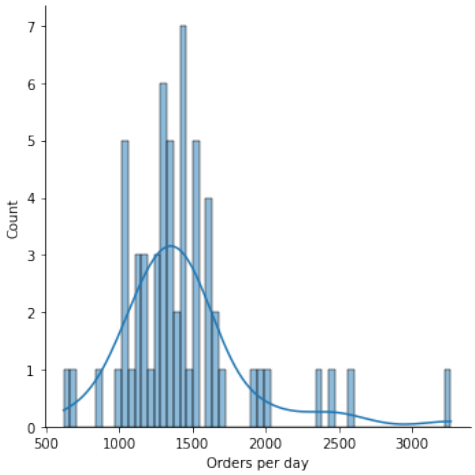
4.1.2 Order Picking

For order picking, the orders in the batch will be removed from the queue. With a function designed in Nataraja et al. (2022), the picking-time is calculated. This function uses the positions of the SKUs in the rack to calculate the robot's travel time. The function considers the picking time required by the robot to put the items in the customer totes, as well as the time to set up the totes for the picking. This function also returns an updated version of the rack so that for the next order picking, the rack is reshuffled corresponding to the action that was just performed. Once the picking of a batch is completed, a new decision moment is triggered.

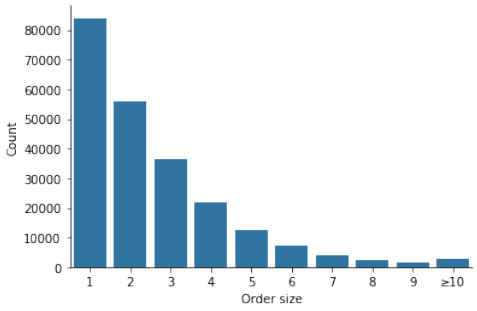
4.1.3 Order Generation and Arrival

Real world data

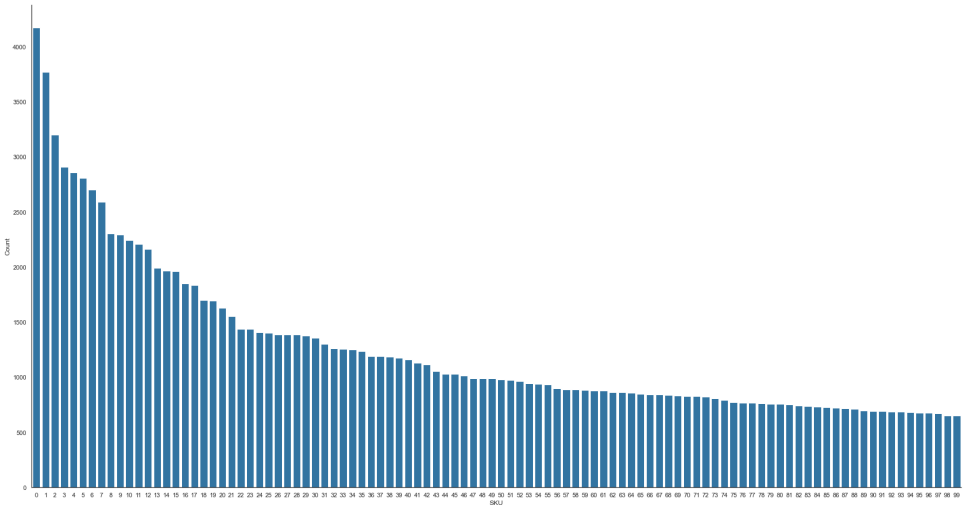
One of the companies that is implementing a pick-and-pack system of Pickr.AI provided 60 days' worth of order data. The company in question will be called by the pseudonym Kallax, to disguise the real name of the company. This data contained more than 228.000 customer orders. Unfortunately, since the picking system at Kallax is not fully operational, the data can not directly be used to run experiments. Instead, we use the data of the 100 most popular SKUs, since the considered setup also consists of 100 different SKUs. In Figure 4.3a, we show the distribution of the number of orders arriving per day. This distribution has a mean of 1432 orders/day with a standard deviation of 431 orders/day.



(a) Distribution of number of orders arriving per day



(b) Distribution of order size



(c) Occurrence of each SKU

Figure 4.3: Real world data used for the simulation

Based on this, we assume that normal day has an average of 1432 orders per day. Based on $\mu - \sigma$, $\mu + \sigma$, $\mu - 2\sigma$ and $\mu + 2\sigma$, we calculate the arrival-rate for days of *low*, *decreased*, *increased* and *high* demand, as shown in Table 4.1. Note that the low arrival rate is lower than that found in the real data, this is due to the skewed distribution.

Demand	*Low*	*Decreased*	*Normal*	*Increased*	*High*
Arrival rate [orders/day]	570	1001	1432	1863	2294

Table 4.1: Arrival for different demands

On average, orders contain 2.6 items (Figure 4.3b). In the simulation, we consider the maximum size of an order to be three. In Figure 4.3c, the distribution of the SKUs is shown. We use this distribution in the simulation when assigning an SKU to an order.

Order arrival process
In the simulation, orders arrive based on a Poisson arrival process. Once an order arrives, the size of the order is randomly determined. For each item in an order the SKU is determined based on the distribution seen in the real data. That means that the probability of each SKU to be selected is $\frac{occurrence\ SKU}{occurrence\ all\ SKUs}$. For each order arrival, the arrival time is logged and a due date is set. To select a due date, we looked at the methods used by Kallax. Orders are shipped out of the warehouse four times a day. One hour before every shipment, a cut-off time is considered. Orders that arrive after the cut-off time will be sent with the next shipment. We assume that these shipment times are equally spread over the day, each 6 hours apart. If an order arrives more than one hour before the upcoming shipment, the ship-time is set as the due date. If it arrives less than one hour beforehand, the next shipment is used. In Figure 4.4, the arrows represent the time frame of an order arriving until its' due date. Once an order is generated and added to the queue, we calculate the arrival of the next order based on a random number generator to sample from a exponential distribution based on the set demand.

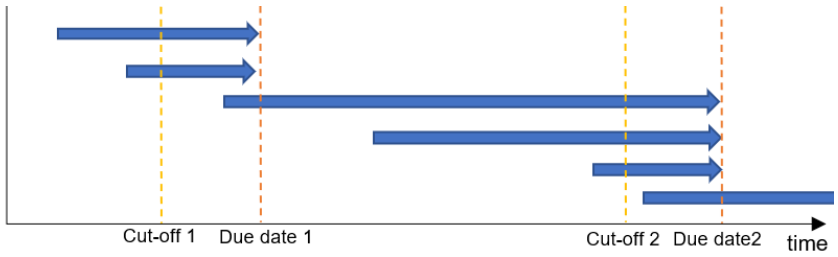


Figure 4.4: Diagram of orders and their due dates

4.2 Markov Decision Process

As explained in Section 2.3.1, the environment needs to be depicted as a Markov Decision Process (MDP). For this project, state transition times are variable. So, a Semi-Markov Decision Process (SMDP) is formed. To create this SMDP, the transition times τ, a finite state space $\mathcal{S}$, an action space $\mathcal{A}$, a reward R, and transition probabilities P are needed. Since we use a simulation model, these probabilities will be not needed.

4.2.1 Time to Transition τ

The difference between an MDP and a SMDP is the time to transition. In an MDP, all transitions are assumed to have a fixed time, whereas for an SMDP, this time might be variable. In this case, the time is dependent on the action, the state and external factors like order arrival. For this reason an SMDP is used. Here, the transition time τ is determined by the simulation, which runs for T duration.

4.2.2 State Space $\mathcal{S}$

The state space is a representation of the environment that captures all relevant data. The state space is also known as the observation space, since it is the only information the agent can "observe" from an environment. A state space can contain any information as long as it can be represented in a vector. The challenge in creating a fitting state space is giving the agent enough information for it to develop good policies, while not overdoing it and making it impossible to find correlations in the data. A larger state space gives the agent more information and so in theory it can develop better and better fitting policies for each state, but it can also make

the learning harder, if not impossible, and bring additional computational cost. The following components comprise the state space in this project.

- Queue content
- Pick-face
- Length of the Queue
- Due dates of orders
- Batch suggestion
- Time
- Arrival rate

Queue content
The queue content is necessary for the agent to know what orders are in the queue so it knows which ones it can pick. Every order o has a due-date δ_o and can have a maximum of m different items, each being one of S different SKUs. Let ω_o denote the picking completion time of order o. The order content of each order o will be represented as a vector ($\vec{O}$), and the vectors of all orders combined will be the representation of the queue. So if an order consists of two items SKU 1 and SKU 2, this order might have a representation in the state space that looks something like $\begin{bmatrix}1\\2\end{bmatrix}$. Unfortunately, PPO is not able to function with multi-discrete state spaces. This means that it would not see 1 and 2 as different SKUs but just as higher and lower values. To make a representation that would work with PPO, one-hot encoding must be used. For each order in the queue, a vector of size S will be created, each spot representing a different SKU. For each SKU in the order, the corresponding index will be 1, all other values will be 0. This does mean that a bit of information gets lost if multiple items of the same SKU are in the same order. The representation of one order $\vec{O}$ in the queue is shown below.

$$\vec{O} = \begin{bmatrix} h_1 \\ h_2 \\ . \\ . \\ . \\ h_S \end{bmatrix} \quad \text{where } h_s \in \{0,1\} \; \forall \; s \in S \tag{4.1}$$

An important aspect of the state space is that it is finite. With that comes the restriction of DRL that the input shape for the agent must be consistent. Since the length of the queue is variable, and theoretically infinite, the state space will only contain the top L_Q orders in the queue. This means that if there are less than L_Q orders in the queue, the remaining space will consist of zeros, since $\vec{O}$ will be all zero if there are no SKUs. This means that the total queue representation $\vec{Q}$ will look as shown in Equation 4.2. This vector has a total size of $S \times L_Q$.

$$\vec{Q} = \begin{bmatrix} \vec{O_1} \\ \vec{O_2} \\ . \\ . \\ . \\ \vec{O_{L_Q}} \end{bmatrix} \tag{4.2}$$

Pick face

The state of the rack is important information to create a well performing policy, as shown by (Nataraja et al., 2022). However, the rack consists of L lanes, each with depth D. Since a one-hot notation must be used, the size needed to create a representation is $D \times L \times S$. With the desired values for these parameters, this would be an extremely large vector which would cause computational issues and potentially create too many variables for the agent to be able to learn properly. To still give an indication of the state of the rack, the pick face is used instead. In the state space, $\vec{PF}$ is added (Equation 4.3), representing which SKUs are in the pick face.

$$\vec{PF} = \begin{bmatrix} h_1 \\ h_2 \\ . \\ . \\ . \\ h_S \end{bmatrix} \quad \text{where } h_s \in \{0,1\} \; \forall \; s \in S \tag{4.3}$$

Batch suggestion

The goal of the observation space is to give all the information needed for the agent to be able to find a relation between the selected actions, the changes in the state space and the given reward. For agents 1 & 2, it is vital to learn the relation between the selected heuristic and the batched orders. The first heuristic works according to FIFO. This relation is very straightforward and easy to learn for the agent since it is based on queue positions. The second heuristic is a more advanced batching heuristic, based on both the content of the orders and the pick-face of the system. To aid the decision making process of the agent, we further include the outcome of the best heuristic from Nataraja et al. (2022) in the state space. This is done with a one hot encoding for the every order in the queue, where 1 marks orders to be batched, creating vector $\vec{B}$.

$$\vec{B} = \begin{bmatrix} h_1 \\ h_2 \\ . \\ . \\ . \\ h_{L_Q} \end{bmatrix} \quad \text{where } h_o \in \{0,1\} \; \forall \; o \in \{1,2,\ldots,L_Q\} \tag{4.4}$$

Queue length, Due dates, Time & Arrival rate

Since the objective of this project is to minimise the sum of weighted tardiness and earliness, the due date is crucial information. This information will be represented in the vector $\vec{DD}$ of length L_Q. For each item in the queue, the value will represent the due date, all the other values will be zero. To actually understand how close the due dates are, the agent is also fed with the current time t in the state space.

All information considered until now is the minimum requirement for the agent to create sufficient policies. However, as explained in Section 4.1.3, the order arrival pattern fluctuates from day to day, and preferably the agents takes this into account when creating its policies. That is why an indication of the arrival rate is given. This is done by using a moving average of the inter-arrival times of the last few orders that have arrived. This scalar (λ) is added to the state space. Lastly, the scalar l is added, to represent the total length of the queue, so if the queue has more than L_Q orders, the agent still has an idea of the total amount of waiting orders.

In addition to the aforementioned elements, we have had considered others, such as total earliness or tardiness, number of picked orders, number of tardy or early orders, due date closeness $(\delta_o - t)$, or a commonality score, where orders with common SKUs score higher. However, we found that a combination of the above explained elements gave the best results.[1] The resulting state spaces are summarised in Table 4.2.

State transitions
Transitions map state-action pairs (s-a) to their subsequent state s'. When taking an action a in state s, s' is reached by updating all components of s. These updates depend on the action taken and if orders arrive during the transition time:

- **Wait**
  - *No orders arrive*: Increase t with waited time. Recalculate $\vec{B}$ and λ.
  - *Order arrives*: Set t to arrival time of the order. Update $\vec{DD}$ corresponding to the new order. Increase l by one. Update $\vec{Q}$ with $\vec{O}$ of the new order. Recalculate $\vec{B}$ and λ.
- **Pick**
  - *No orders arrive*: Increase t with the pick time. Update $\vec{DD}$ and $\vec{Q}$ corresponding to the picked orders. Decrease l by the pick batch size. Update $\vec{PF}$ based on the picking action. Recalculate $\vec{B}$ and λ.
  - *Orders arrive*: Increase t with the pick time. Update $\vec{DD}$ and $\vec{Q}$ corresponding to the picked orders and the newly arrived orders. Update l based on the number of orders that arrived and are picked. Update $\vec{PF}$ based on the picking action. Recalculate $\vec{B}$ and λ.

4.2.3 Action Space

The action space is where the three agents differ. The simple agent, agent 1, has an action space of just one scalar that can take three values; 0 for waiting, 1 for the first batching heuristic, and 2 for the other heuristic.

Agent 2 is slightly more complex. The agent decides how many orders should be batched. The action space is shown in Equation 4.5. Now the second value in the action space represents a value between 1 and the maximum batch size B.

$$\mathcal{A} = \begin{bmatrix} h \\ b \end{bmatrix} \text{ with h } \in \{0 \ldots, 2\}, \text{ b } \in \{1, \ldots, B\} \tag{4.5}$$

Agent 3 has the most complex action space. The first part is a scalar related to the first decision of the agent; "Do we batch now?". This scalar can either be 0 or 1, representing batching/picking or waiting respectively. The second part of the action space is related to the question; "which orders will be batched?". Since the action space can be multi-discrete, each element in the vector represents a spot in the batch, and each value corresponds to the queue index of the order to be placed in the batch. Since the agent can only observe the first L_Q orders,

[1] Due to the iterative nature of this project, some experiments are done in much earlier versions of the simulation environment. It is assumed that these results are still representative of the current simulation.

it is only allowed to select orders from this "observed queue". These values are natural numbers between 0 and L_Q, with 0 representing the first and $L_Q - 1$ the last order in the observed queue and L_Q represents an empty spot.

$$\mathcal{A} = \begin{bmatrix} w \\ c_1 \\ . \\ . \\ . \\ c_B \end{bmatrix} \text{ where } w \in \{0 \ldots, 1\} \text{ , } c_b \ \in \{0, \ldots, L_Q\} \ \forall \ b \in \{1, \ldots, B\} \tag{4.6}$$

As explained in Section 4.1.1, the first decision is leading. So if an agent selects the wait action combined with multiple orders, the resulting action would still be waiting. However, if an agent returns an empty batch and selects picking, the performed action would also be waiting. If an agent selects indices of orders that are not currently in the queue, these will be changed to empty spots. These kind of actions where it tries to pick non-existing orders, are marked as faulty actions. In DRL, it is common to not simulate a state change and return a penalty in case of faulty actions. After multiple experiments with this setup, this did not seem beneficial for the agent's learning behaviour and performance, and so we used the setup explained in Appendix A.1. Furthermore, we tested whether the first part of the action space related to the wait or pick action could be removed. In that case the agent would wait if the returned batch was empty. We also tested if a one-hot action space would show better performance than the multi-discrete space. Multiple combinations of the mentioned setups have been tested with the result of the current settings showing the most promising results.[1]

4.2.4 Reward Function

We have considered many different variations of objective for the reward function. These things included, but were not limited to; penalising the total tardiness of the queue, penalising earliness of picked orders, rewarding waiting actions, penalising tardiness squared, penalising tardiness of the picked batch, and rewarding waiting time. Many of these elements were tried in different combinations and using varying weights.[1] Eventually, we settled on the following reward function, where a reward r is obtained when an action a is taken and state s' is reached and let β be the size of the suggested batch if pick action is chosen.

$$r\left(a, s'\right) = \begin{cases} 0 & \text{if } a = \text{wait and } s' \text{ has no tardy orders} \\ -1 & \text{if } a = \text{pick, picked batch has no tardy orders and } s' \text{ has tardy orders} \\ & \text{or } a = \text{wait and } s' \text{ has tardy orders} \\ \sum\limits_{o \in \beta} T_o & \text{if } a = \text{pick, picked batch has tardy orders and } s' \text{ has tardy orders.} \\ \sum\limits_{o \in \beta} T_o + W_o & \text{if } a = \text{pick and } s' \text{ has no tardy orders} \end{cases} \tag{4.7}$$

The reward function contains two components T_o and W_o. The equations used to calculate these values are shown below, where ω_o represents the time an order or batch is completed, δ_o is the due date of an order and $x^+ = \max(0, x)$.

$$T_o = \frac{-[\omega_o - \delta_o]^+}{T} \tag{4.8}$$

$$W_o = \frac{T - [\delta_o - \omega_o]^+}{T} \tag{4.9}$$

T_o is an expression of the relative tardiness of an order o. It divides the tardiness in seconds by the simulation time in seconds. W_o is an expression that outputs a value that nears 1 as earliness gets closer to 0. Both expressions use *simulation duration*, T, as an upper limit for maximum tardiness or earliness. Thus, the output value of the functions is normalised so that their absolute values are between 0 and 1. This is done because most RL algorithms tend to learn better with normalised rewards.

In Figure A.2, a decision tree is shown of how the reward is calculated. First we check if there are any tardy orders in the queue after the action has been executed. If this is not the case the reward is either T_o and W_o of the batch (corresponding to the last case in Equation 4.7), or 0 if the agent was waiting (first case of Equation 4.7). If there are tardy orders in the queue, the reward for picking is only determined by T_o (third case of Equation 4.7). If the agent does not pick any tardy orders even though there are tardy orders in the queue, this could happen by either picking different orders or by not picking at all, we deem this as the worst possible action and therefore give a penalty of -1 (second case of Equation 4.7). In Figure 4.5, the potential reward of each order is shown over time.

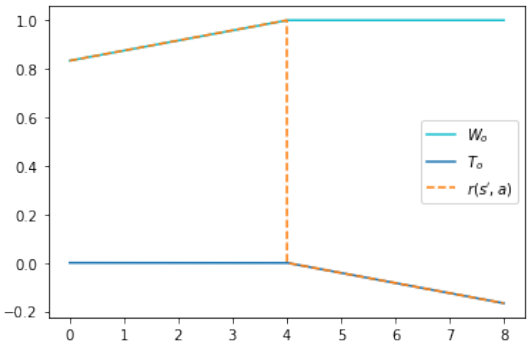


Figure 4.5: Potential reward of an order over time with due date at 4 hours

The negative value of T_o ensures that picking an order early is always rewarded higher than picking an order tardy. If there are tardy orders in the queue, and the agent does not pick them, we deem this as the worst possible action, and therefore give a relative high penalty of -1. Due to the assumption of common due dates between orders, we can assume that usually once orders become tardy, there are multiple tardy orders in the system, and thus the reward will behave as shown in Figure 4.5. Note that there is one unique edge case where the reward function shows different behaviour: if after picking there are no tardy orders in the queue, but (some of) the picked orders are tardy, the reward of this action is calculated by both W_o and T_o. In this case W_o is 1 due to its formulation. Therefore, the reward given for picking tardy orders will be positive which normally would have been negative. However, in this case, the desired behaviour still receives the highest possible reward. If the agent would have picked non-tardy or no orders at all, the reward would have been -1. If the agent would pick only one of the tardy orders the reward would only be based on T_o since there are still tardy orders left in the queue. So the best action would still be to pick all tardy orders.

Multiple variants of this reward function have been tried, including different values instead of -1 and different weights for W_o and T_o. We also tried a function for W_o that would progress from 0 to 1 starting from the moment of arrival and ending at the due date. It showed promising results but did not outperform the current one.

4.3 Implementation

In this section we first set the scope of the project. Next we introduce the baseline decision-makers. Afterwards we explain the used parameters for the simulation and agents followed by an introduction of all the used agents. Lastly, the training process of these agents is explained.

4.3.1 Scope

For this project, we consider a set up of Pickr.AI that has 25 lanes of 7 bins deep. 10 lanes have an active slot in the pick-face. We assume a product range of 100 different SKUs. Lastly, we take a maximum of 3 items per customer order.

4.3.2 Baseline

We will consider eight different multi-heuristics as baselines. These heuristics consist of combinations of four waiting and two batching heuristics. The following waiting heuristics are used:

- **Wait for Full Batch (WFB)** is a VTWB method where the agent waits until 5 items are in the queue to batch.
- **Wait 20 (W20)** is an adapted FTWB method that we call Fixed Time Window Batching with Carryover (FTWBC). This method makes batches of all orders in the queue on a fixed time interval. When the interval has passed, the orders that not have been processed will be carried over to the new group of orders, where new batches will be made. This guaranties a more optimal batching strategy. For W20, we set the time interval to 20 minutes. In other words, the W20 batches all orders in the queue and optimises this every 20 minutes.
- **Instant Picking (IP)** is a VTWB method where the agent waits until a minimum of 1 order is in the queue to batch. i.e. an pick action will start as soon as an order arrives, or a robot completes the picking of a batch.
- **Wait Combined (WC)** combines W20 and WFB and waits either 20 minutes or until 5 items have arrived in the queue. However, these 20 minutes will only start counting after arrival of first order.

For the batching heuristics, we consider both FIFO and MCSSOR as explained in Section 2.2. Note that FIFO is effectively the same as an earliest-due-date heuristic since due dates are assigned based on the cut-off moments.

4.3.3 Simulation and Agent Parameters

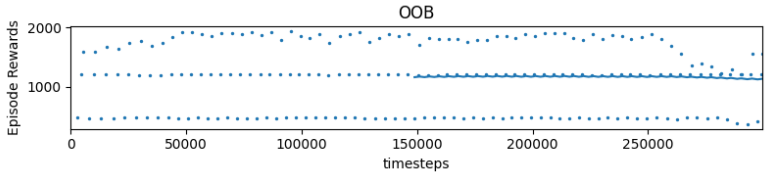
We have set L_Q to 30 and the maximum time a wait-action can last is 30 seconds. For a fair comparison of results, the same values are considered for the heuristics. These values result from a series of tests considering computational time and the agents' performance. We found that episodes of 24 hours simulation time were sufficient for training. Shorter episodes were not sufficient for the agent to learn the consequences of too much waiting and thus caused an

overflowing queue. Due to an increase in computational time, and the seemingly slower learning behaviour of the agent, longer episodes are not preferred.

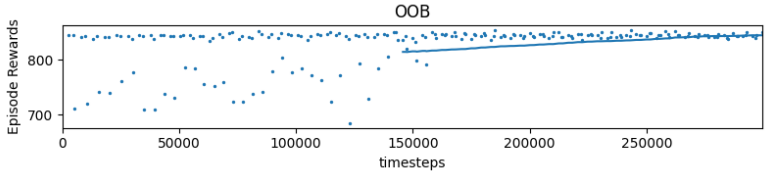
Like the work of Cals (2019), we initially planned to train the agent in an environment with steady demand and then test how the agent performs when the demand decreases or increases. However, the performance proved better when training was done for multiple arrival rates. Part of the reason is just simply that agents tend to do better in familiar environments, so if we train an agent in a normal demanding environment and then test its performance for high demand, it is unfamiliar with this situation. It will not perform as good as an agent that has seen a similar environment during training. In addition to this, it allows for faster (and in some cases even better) learning. The agent will learn the best picking strategies to decrease tardiness when training in a high demanding environment. On the other hand, it will learn the best waiting strategy when presented with a low demanding environment. Thus, we trained the agents with a changing demand between episodes. The different demands allow for more variable situations during training and therefore, decrease the need of exploration. Training will be done in either a *low* (570 orders/day), *normal* (1432 orders/day) or *high* (2296 orders/day) demanding environment (Table 4.1). We have also considered adding *decreased* and *increased* arrival rates, or even more additional arrival rates, but we found that this did not improve performance.

4.3.4 The Agents

Three different type of agents will be used, all with a different action space. The agents that can select between heuristics (1, 1b and 2) make a selection between FIFO and MCSSOR. In Table 4.2, we show the details of all different agents for this project. Agents 1b and 3b have not been discussed before. For agent 1b, the reward function is multiplied with a factor $\frac{1000}{arrival\ rate}$, with arrival rate in orders per day. This normalises the reward between episodes with different arrival rates, since the reward depends on how many orders can be picked. With this normalised function the maximum possible reward for every arrival rate should approximately be the same. This reward function did not show significant differences in performance compared with the non-normalised function. However, it did show interesting learning behaviour, as is shown in Figure 4.6. These figures show the cumulative reward per training episode as dots, and the moving average of these rewards as a line. Since the non-normalised function has already proven successful in earlier test, this will be used as the main reward function, and the normalised will be tested along side.



(a) Agent 1



(b) Agent 1b

Figure 4.6: Learning behaviour of agent 1 and 1b

We create two versions of the complex agent - 3 and 3b. The observation space given to agents 1, 1b, 2 and 3 gives limited information about order content, with which optimal batching strategy cannot be reached. To let the agent develop its own batching strategy, we give a large observation space to agent 3b. Here the agent has all information about the pick-face and queue that it needs to develop sufficient batching method. However, this state space has a size of $S \times (L_Q + 1) + L_Q + 3$, compared to the other used spaces $(2L_Q + 3)$. Theoretically, agent 3b should show same or even better performance than agent 3, since it is not limited to the bias of the used heuristics. However, the large size of the observation space makes training the agent significantly harder. This is for an already complex action space with more than 57 million possible combinations. This large action space of agent 3 and 3b also allows for infeasible batches to be created. What this means and how we deal with that is explained in Appendix A.1.

Name	Observation space	Action space	Reward function
Agent 1	$\begin{bmatrix} l \\ \vec{DD} \\ \vec{B} \\ t \\ \lambda \end{bmatrix}$	$\begin{bmatrix} z \end{bmatrix}$ with z $\in \{0 \dots, 3\}$	default
Agent 1b	$\begin{bmatrix} l \\ \vec{DD} \\ \vec{B} \\ t \\ \lambda \end{bmatrix}$	$\begin{bmatrix} z \end{bmatrix}$ with z $\in \{0 \dots, 3\}$	normalised
Agent 2	$\begin{bmatrix} l \\ \vec{DD} \\ \vec{B} \\ t \\ \lambda \end{bmatrix}$	$\begin{bmatrix} h \\ b \end{bmatrix}$ with h $\in \{0 \dots, 2\}$, b $\in \{1, \dots, B\}$	default
Agent 3	$\begin{bmatrix} l \\ \vec{DD} \\ \vec{B} \\ t \\ \lambda \end{bmatrix}$	$\begin{bmatrix} w \\ c_1 \\ . \\ . \\ . \\ c_B \end{bmatrix}$ with w $\in \{0 \dots, 1\}$, $c_{1 \dots B} \in \{0, \dots, L_Q\}$	default
Agent 3b	$\begin{bmatrix} \vec{Q} \\ \vec{DD} \\ \vec{PF} \\ t \\ \lambda \\ l \end{bmatrix}$	$\begin{bmatrix} w \\ c_1 \\ . \\ . \\ . \\ c_B \end{bmatrix}$ with w $\in \{0 \dots, 1\}$, $c_{1 \dots B} \in \{0, \dots, L_Q\}$	default

Table 4.2: Summary of the created agents

4.3.5 Training

For the creation and training of the models, we have used Stable Baselines (sta). This allows for easy creation and training of DRL models. By simply entering the environment and the desired policy, an NN will be created and trained for a set amount of time-steps. Each decision-moment will be seen as one time-step. We have decided to use PPO due to its previous successes in the field (Cals, 2019; Beeks et al., 2022).

During training, the environment is reset after an episode so that the arrival rate is changed. We also tested randomly selecting between the entered arrival rates; however, this seemed to decrease the agent's performance1. Possibly caused by a few episodes with the same arrival in a row, causing a policy update fitted for that arrival rate, but not for the others.

When resetting the environment, everything is set to its initial values. However, the state of the rack is not; this is to prevent the rack from starting in the same position every time and training the agent for a specific scenario. One more thing that is changed is the probabilities of each SKU occurring. When resetting the environment, SKUs connected to each probability are shuffled. The advantage is that the agent does not learn policies based on SKUs, so it becomes more generic. However, it does take away the possibility to create a specific policy fitted to a specific scenario. Note that this is only a problem for agent 3b since all other agents do not have SKUs as part of their observation space. One more advantage of reshuffling the SKUs is that it acts as another method of reshuffling the rack so that not always all popular products are close at the start, in a more computationally efficient way.

Callback
Saving the model during training is done using a callback function. The agent does not always converge or does not necessarily end on the best performing agent of that training session. The callback function saves the best performing agent during training. We use an evaluation callback, where the agent is evaluated in a different environment every set amount of time-steps. We set it up in such a way that the evaluation is done three times for each of the arrival rates, so nine times in total. After these nine simulations, the mean of the cumulative reward is used to check if the current policy is better or worse than the previous policy. If the mean is high, the policy is saved. The callback function does not influence the training; it only influences when a model gets saved.

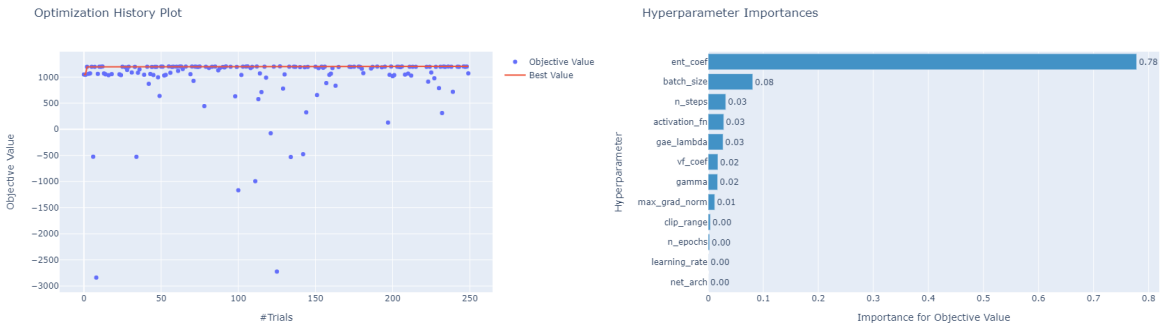
Hyper-parameter tuning
Besides the observation space and reward function, the learning behaviour of the agent is heavily influenced by hyper-parameters. With Stable Baselines Zoo, (SBz) these parameters can easily be tuned by training the agent with a different set of parameters and then checking the best performance based on an evaluation callback function. This process continues iteratively, where the tuner bases the new parameters on old results and tries to maximise the average reward given by the final callback.

First, a regular training session is done to find the number of time-steps for the agents to perform well. Depending on the complexity of the agent, each agent needs a different amount of time-steps to converge to adequate performance. For agents 1 and 1b, this is 2e5 time-steps. Agent 3 and need approximately 1.5e6. Due to time constraints, agents 2 and 3b were not elaborately tested before hyper-parameter tuning. For agent 2, we assume that 3e5 time-steps should be sufficient since this agent is a bit more complex than agent 1. For agent 3b, we consider 1.5e6 time-steps. Even though it is more complex than agent 3, a longer training time would be too time-consuming. Training for 1e5 time-steps takes 15-20 minutes to run[2], in which approximately 190 days are simulated. This number can heavily fluctuate due to the significant difference in duration between a waiting and picking action. These numbers are without the callback function, which adds a significant amount of time to the training process. Since hyper-parameter tuning will repeatedly train an agent many times, we evaluate every training session only 20 times.

[2]Runs are done on a AMD Ryzen™ Threadripper™ 3990X Processor CPU @ 2.8 GHz

Figure 4.7 shows the output of the hyper-parameter tuning session of agent 1. It can clearly be seen that there is a significant difference in performance between different trials, caused only by the hyper-parameters. From Figure 4.7b we can conclude that entropy coefficient is the most influential hyper-parameter. This parameter influences the exploration rate. Second most important is the batch size, which is the number of samples used for each gradient descent update.

Due to time constraints and limited resources, we could only finalise the tuning sessions of agents 1 and 1b for 250 trails. Agents 2, 3 and 3b were stopped at 194, 37, and 36 trails, respectively. The plots of all the hyper-parameter tuning sessions and the best hyper-parameters per agent can be found in Appendix A.3.



(a) Optimisation history

(b) Parameter importance

Figure 4.7: Results of hyperparameter tuning of agent 1

5 Experiments

In this chapter, we explain the set up of the different experiments conducted. The experiments are done to test the performance of the agent, compare the performance to those of the heuristics, and find an answer to the research questions.

5.1 Changes in Demand

For the last two research questions, we want to know how the agents handle different environments and compare them to the baselines. We have split this into three sections. First, we compare how the agents and heuristics perform on days with different demands. Secondly, we try to see how the agent adapts to sudden changes in the environment. Lastly, we test how the agent would perform in a real-world scenario, where the arrival pattern is less stable than previously assumed. The following KPIs will measure the performance:

- mean earliness, tardiness and lateness
- percentage of orders picked early, tardy or in time

In Appendix B.1, we explain more on how the percentages of early and in time orders are calculated and why these metrics should only serve as a rough indication of the performance. However, when combined with the other KPIs, a well-informed conclusion can be drawn.

5.1.1 Different Demands

In this first part of the experiment, we want to see how the decision-makers would perform in different demanding environments. To test this, we perform a sensitivity analyses, where the agent and heuristics are subjected to five different scenarios, each with different demand. We use the demands noted in Table 4.1. Each simulation will simulate five days. We chose this length mainly for *high* demand scenarios. If an agent would not pick enough and leave some orders in the queue, this would not seem to affect the performance, but in the long run, this would cause a bullwhip effect and lead to overflowing queues and, therefore, extreme tardiness. To check if buffers are not overflowing and see the actual performance of an agent, we measure over five days. Each run is repeated ten times for different random seeds to increase the precision of the results.

5.1.2 Peak Demand

Kallax provided us with additional information on their sales in the holiday season (November-January). This showed that their promotional sales week, "Black Week", is the busiest time in all four months of data that we have analysed. This promotional week ended with "Black Friday" followed by "Cyber Monday". These days have an extremely *high* demand of 470% and 360% *normal* demand. Both these days are followed by two days with *decreased* demand.

In this part of the experiment, we will measure the performance of all decision-makers for these sudden increases in demand. We start with a day of *normal* arrival followed by *high* to create some backlog of orders to represent the effect of "Black Week". We will simulate a run of 15 days where arrival varies per day according to Table 5.1. We have chosen to simulate for 15 days since the peak arrival causes a large backlog of orders that the robot needs a significant amount of time to recover from. This time to recover will be used in combination with the previously mentioned KPIs to judge the performance of each decision-maker. Each simulation is repeated ten times for different random seeds.

	simulation day														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
demand	*normal*	*high*	470%	*decreased*	*decreased*	360%	*decreased*	*decreased*	*normal*	*normal*	*normal*	*normal*	*normal*	*normal*	*normal*
orders/day	1432	2294	6730	1001	1001	5155	1001	1001	1432	1432	1432	1432	1432	1432	1432
due date extension	-	-	48h	48h	48h	48h	48h	48h	-	-	-	-	-	-	-

Table 5.1: Arrival rate per day of peak demand simulation

5.1.3 Real arrival pattern

For the last part of this experiment, we take a new look at the data provided by Kallax. Up until this point, we assumed that orders arrive somewhat evenly throughout the day, with the arrival rate according to a Poisson arrival process. However, if we look at the data, we see that there is a distinct arrival pattern shown in Figure 5.1. In this figure, you can see the probability distribution of an order arriving during the day. This figure shows that sales are much higher during the day, with peaks in the morning and evening, and sales are meagre during the night when customers are likely sleeping.

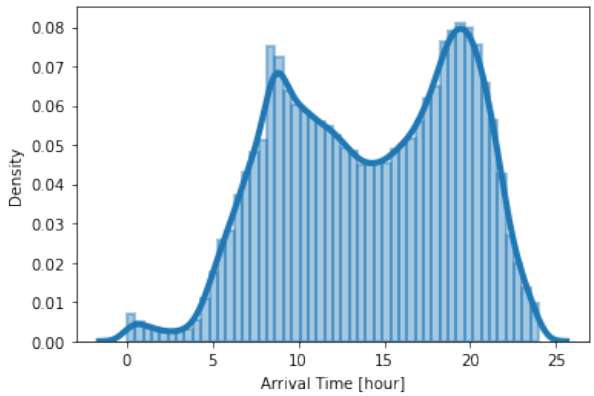


Figure 5.1: Probability density plot of order arrivals during 24 hours

This test simulates this arrival pattern for a day with *normal* demand. Based on the data, we have created average arrival times for ten-minute long increments throughout the day. During the simulation, the arrival rate will be dependent on the (simulation) time of the day and the arrival rate of the corresponding increment. Previously we assumed due dates to be equally distributed throughout the day. However, we use the exact due dates as Kallax uses for this experiment.

5.2 Full Factorial Analyses

To measure the effect of each decision on the contribution to the objective, we employ 2^k *factorial design* as is done by Nataraja et al. (2022). This is a strategy for designing experiments, where k factors can influence the outcome (Box et al., 2005). In our case, factors are decisions, what (A) and when (B) to batch. Both decisions can either be random (-) or optimal (+). This gives 4 runs in total (Table 5.2). We will measure the effect of the decisions for earliness and tardiness separately. We will also measure these effects for each demand scenario.

Decision Combination	Decision A	Decision B	Response
1	-	-	(1)
2	+	-	a
3	-	+	b
4	+	+	ab

Table 5.2: Design matrix for 2^2 factorial experiment

We simulated each combination for $n = 10$ replications. We set W20 and MCSSOR as optimal. Random "when" decisions are made by randomly selecting between *wait* and *pick* at each decision moment. For a random "what" decision we select 5 random orders from the queue, if less than 5 orders are present, we pick all orders. The effect is calculated by means of the following equations:

$$\begin{aligned} e_A &= \frac{(a-1)(b+1)}{2^{k-1}} \\ &= \frac{ab+a-b-(1)}{2} \end{aligned} \tag{5.1}$$

$$e_B = \frac{(a+1)(b-1)}{2^{k-1}} \tag{5.2}$$

$$e_{AB} = \frac{(a-1)(b-1)}{2^{k-1}} \tag{5.3}$$

In these equations, e_j denotes the effect of decision j, with $j = AB$ for the combined effect. For example, if we want to know what effect decision A has on tardiness, we will use Equation 5.1. The values a, b, ab and (1) denote the objective value for runs with compositions based on Table 5.2. For example, in the case of tardiness, a would be the average tardiness of orders for a run where A is optimised and B is random. Once the effect is calculated it can be used as an indication of how strong the influence of the decision is on the objective. Based on the absolute value of the effects we can see which of these decisions has the most impact on the objective value.

6 Results

In this chapter, we will discuss the results of the experiments. We start by comparing the performance of the agent with the benchmarks in different demanding scenarios, including a scenario where orders arrive in a realistic arrival pattern, and one where days of peak demands are simulated. This is followed by the results of the full factorial analyses. In the last section, we examine the behaviour of the agents.

6.1 Changing Demand

We will compare the five agents with all the batching heuristics (MCSSOR, FIFO) and waiting heuristics (IP,WFB,W20,WC) combinations. We have selected four decision-makers for comparison. W20-MCSSOR and agent 1b are selected for their overall performance. FIFO is one of the most famous benchmarks used in literature. IP is probably one of the most commonly used practices since this will yield the highest utilisation. One can imagine that any e-commerce warehouse equipped with a robotic system would not want to keep the robot idle. For this reason, IP-FIFO is added to the comparison. Lastly, Kallax is currently using WC-FIFO in their test setup, and thus, this is also compared. All results can be found in Appendix C, and summarised in Appendix E. For the *low* and *high* arrival results, the results of before and after hyperparameter tuning are shown. The after tuning results shown on the right-hand side after are marked with 'HP' in these plots. This shows the effect of hyperparameters on the behaviour of the agent. All results are averages of 10 episodes of 5 days.

6.1.1 Low Demand

Figure 6.1 shows that none of the decision-makers cause any tardiness. Agents 1, 3 and 3b seem to show similar performance as IP heuristics. Agent 2 shows a significant increase of in-time orders of 1 percent-point, which is around 16%. Agent 1b has the most in-time orders. On a yearly basis, agent 1b would pick almost than 5000 more in-time orders than W20-MCSSOR in environments with constant *low* demand. This is an increase of more than 20%. Compared to WC-FIFO and IP-FIFO the difference would even be more than 6800 and 13100 orders per year, respectively.

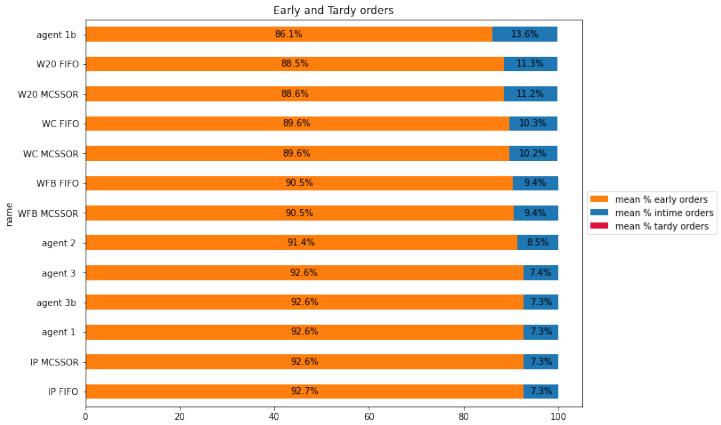


Figure 6.1: Performance comparison of agents and heuristics under *low* demand arrival.

In Figure 6.2, the blue line represents the due date. We can see the distribution of when orders are picked compared to their due date. The distribution of agent 1b seems to be flattened a bit more, and of all the decision-makers, agent 1b picks orders closer to the due date. On average, an order is picked 3.61 hours before the due date. This is ten minutes later than what

Kallax is currently doing (WC-FIFO), without introducing any tardy orders. On a daily basis that would mean 114 hours less earliness, which would substantially reduce the holding cost.

In Figure C.1, the improving effect of hyperparameter tuning can be seen on almost all agents, most notably on agents 3 and 3b. At first, both agents had a significant amount of tardy orders, they now perform similar to IP-FIFO. In Figure E.1, we have plotted the average reward

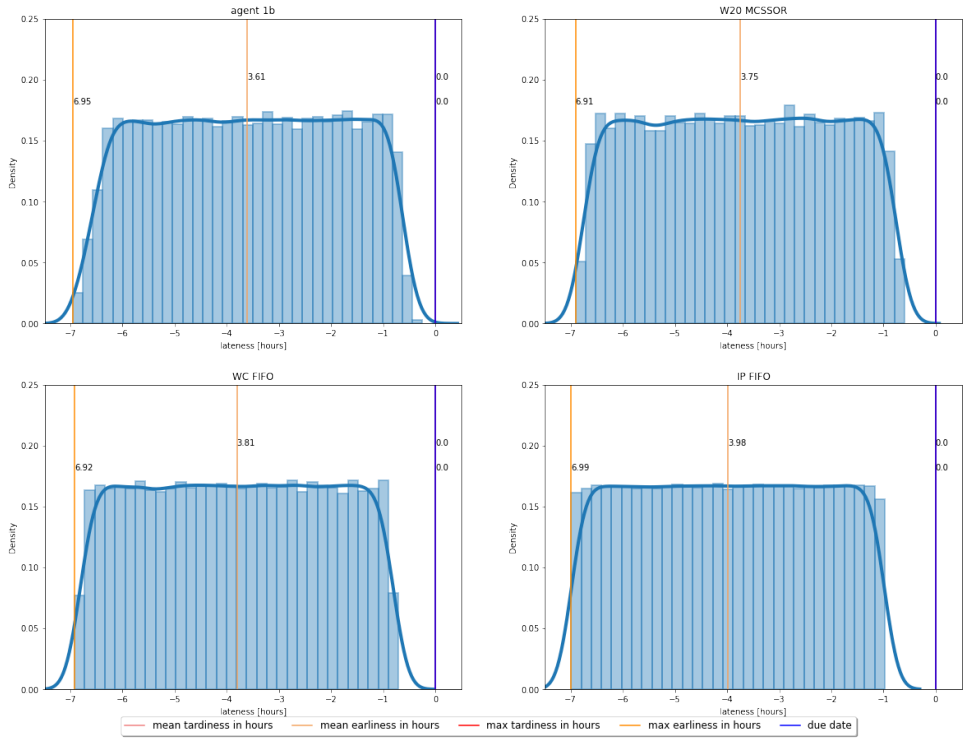


Figure 6.2: Distribution of picking completion time of orders along the due-date for *low* arrival.

per picked order. This reward is calculated based on the reward function used in agents 1, 2 3 and 3b. This value indicates the earliness and tardiness cost related to each order. A higher value would mean a lower cost. In this figure, we can see that agent 1b has the lowest cost per order, followed by the W20 heuristics.

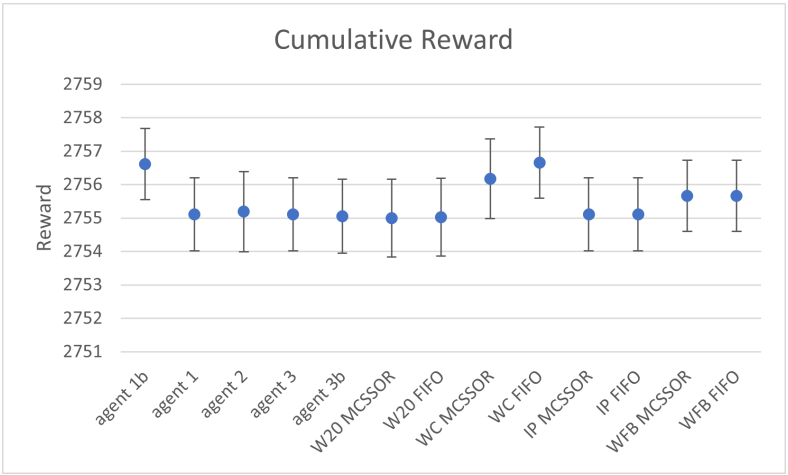


Figure 6.3: Mean cumulative reward per decision-maker for *low* demand with episodes of 5 days

In Figure 6.3 we show the mean cumulative reward per episode with a confidence interval of 95%. Based on the relatively small size of these intervals we can state that the found mean reward is robust.

Note that in Figure 6.1 and some similar graphs that will follow, the sum of percentages of early, in-time and tardy orders do not add up to 100%. This occurs due to the following logic. If an unpicked order is tardy, it will still be marked as tardy. However, if an unpicked order in the queue is not tardy, it will be marked as neither early nor tardy nor in-time. The percentages are calculated based on all orders, this includes the picked orders as well as the orders remaining in the queue. However, the percentages of non-picked non-tardy orders are not shown in the graphs. All graphs are sorted primarily on percentage of tardy orders and subsequently on the percentage of in-time orders.

6.1.2 Decreased Demand

For *decreased* demand, we observe similar performance among W20-FIFO, W20-MCSSOR, and agent 1b, with orders being slightly less early for the heuristics. On a yearly basis W20-MCSSOR would pick around 1400, or around 1.4% more in-time orders than agent 1b on a yearly basis. Both these decision-makers would have more than 7300 more in-time order on a yearly basis than WC -FIFO and even more than 14200 than IP-FIFO. It is noticeable that decision-makers with the same waiting heuristic have very similar performance. This is also visible in the results of *low* arrival.

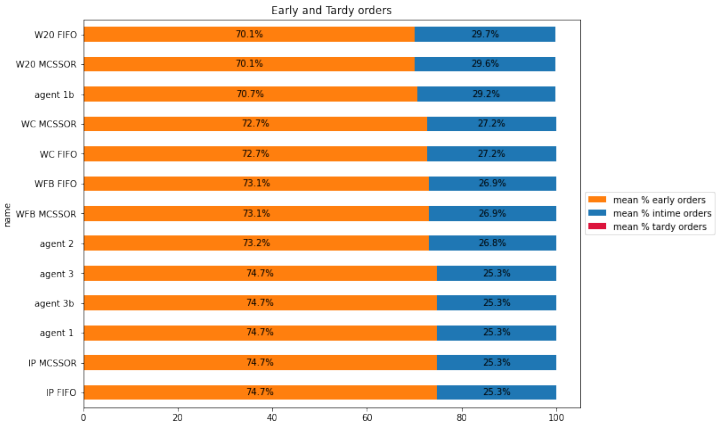


Figure 6.4: Performance comparison of agents and heuristics under *decreased* demand arrival.

For the *decreased* demand scenario, the same observation can be made as with *low* demand. Agent 1, 3 and 3b show similar performance, and very close to the IP heuristics. Agent 2 shows a decrease in mean earliness and agent 1b shows an even larger increase (Figure C.3).

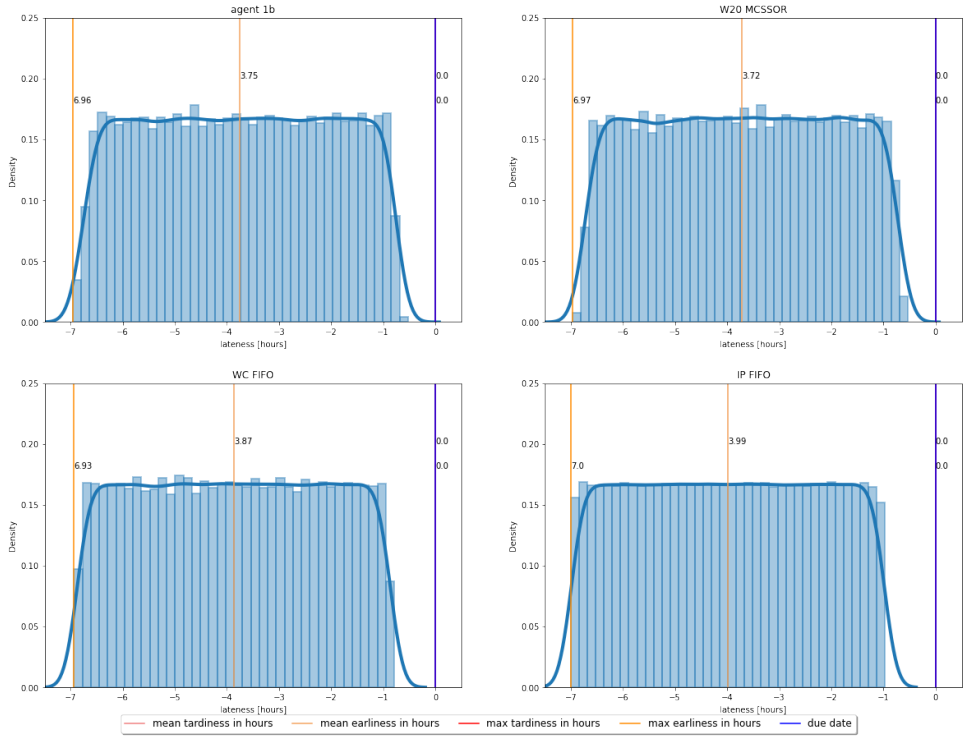


Figure 6.5: Distribution of picking completion time of orders along the due-date for *decreased* arrival.

Figure E.1 show that the W20 heuristics have the lowest cost per order, closely followed by agent 1b.

6.1.3 Normal Demand

In scenarios with *normal* demand, we see the grouping of same waiting heuristics that has been seen before, i.e. decision-makers with the same waiting heuristic perform similar, regardless of the batching heuristic. Heuristics with the W20 heuristic are showing the least amount of early orders, followed by agent 1b. From Figure C.5, it becomes clear that, compared with other decision-makers, both W20 heuristics and agent 1b pick more orders closer to the due date. However, agent 1b has a higher earliness per order indicating a slightly lesser performance. This can also be concluded from Figure E.1, where the difference in cost for the W20 heuristics and agent 1b is higher than before.

Overall agent 1b could pick around 10450, or around 4% less in-time orders than W20-MCSSOR on a yearly basis. Agent 1b would pick close to 6800, or around 3% more in-time orders than WC-FIFO and 15150 (more than 6%) more in-time orders than IP-FIFO.

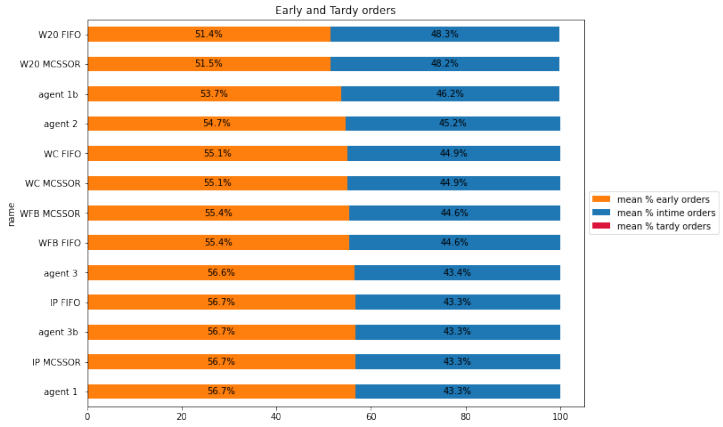


Figure 6.6: Performance comparison of agents and heuristics under *normal* demand arrival.

Again the performance of the IP heuristics and agents 1, 3 & 3b are very similar. Again, agent 2 shows some increase in in-time orders and decrease in maximum and mean earliness per order. This time it even outperforms the WC and WFB heuristics on mean earliness and number of in-time orders.

Based on Figure 6.7 we can state that agent 1b has 172 more hours of earliness than W20-MCSSOR. Agent 1b also shows more than 114 hours less earliness than the current practice of Kallax, WC-FIFO. Compared with IP-FIFO, the difference is even more than 243 hours per day.

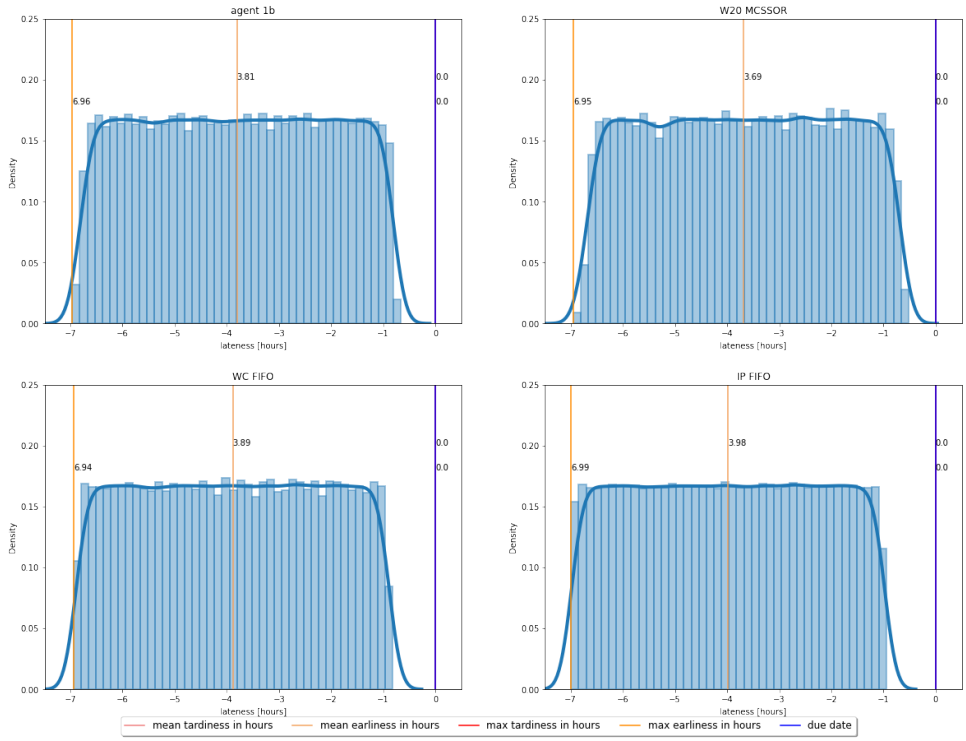


Figure 6.7: Distribution of picking completion time of orders along the due-date for *normal* arrival.

6.1.4 Increased Demand

With *increased* demand, decision-makers with the same waiting heuristics, but different batching heuristics start to deviate more in performance. In Figure C.7 and Figure 6.8, we can see that for the first time other agents show less earliness than agent 1b. Agent 2 shows very similar performance with a slightly lower maximum earliness. Agent 3 has the same maximum tardiness, but orders are on average 1.8 minutes less early. W20 shows the least earliness, and when combined with FIFO, it also drastically decreases the maximum earliness.

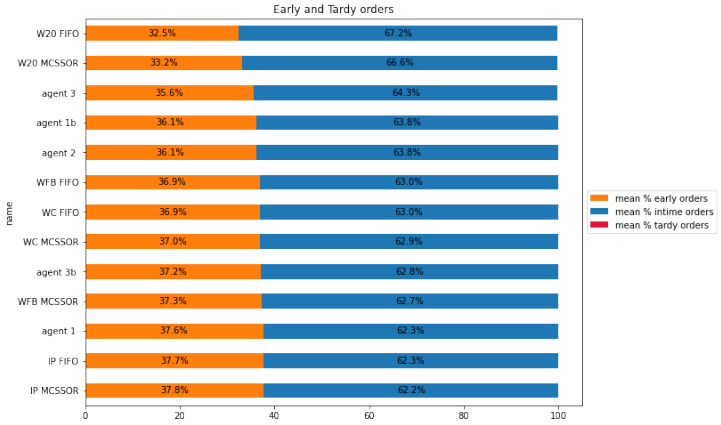


Figure 6.8: Performance comparison of agents and heuristics under *increased* demand arrival.

Agent 1 is still performing similar to the IP heuristics. Agent 3b shows an increase in percentage of in-time orders, compared with agent 1. Agents 1b and 2 perform very similarly, and for the first time, agent 3 has the least number of early orders and lowest cost of all agents (Figure E.1). Previously there was a clear distinction between agents' performance with the normal reward function, and agent 1b with the normalised reward function. With an *increased* demand, this difference seems to fade.

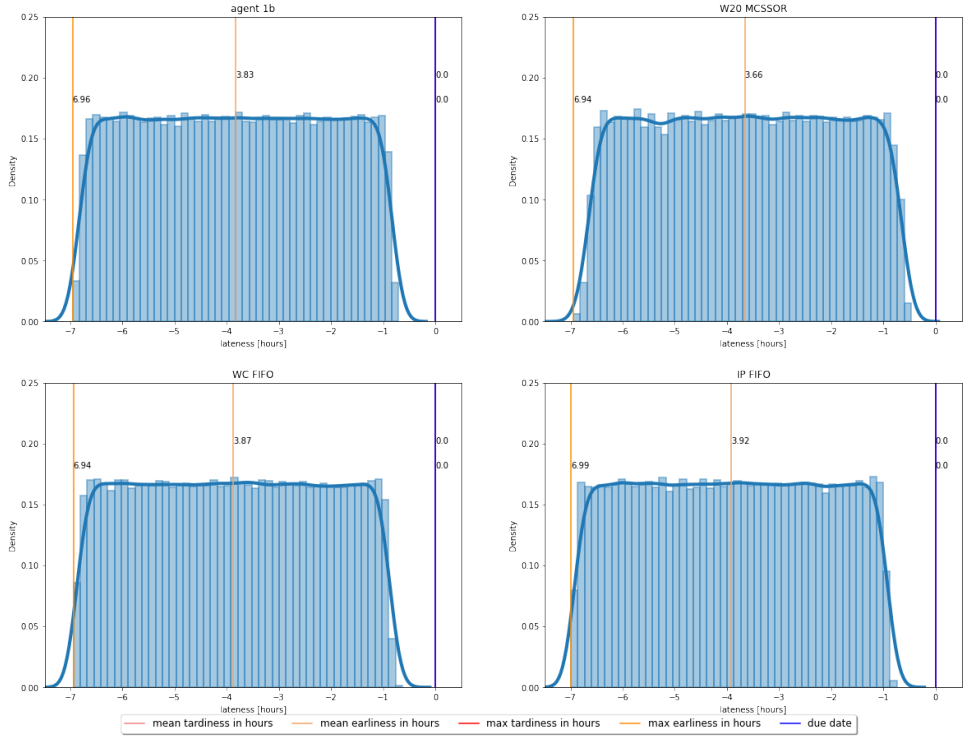


Figure 6.9: Distribution of picking completion time of orders along the due-date for *normal* arrival.

Based on Figure C.7 we can state that agent 1b has 316 more hours of earliness than W20-MCSSOR. Agent 1b also shows more than 74 hours less earliness than WC-FIFO. Compared with IP-FIFO, the difference is more than 167 hours per day.

6.1.5 High Demand

When simulating *high* demand, we see tardy orders. In Figure 6.10, orders are sorted first on number of tardy orders and then on number of in-time orders. Figures C.9 and C.10 show that IP-MCSSOR is the only decision-maker without any tardy orders. This is followed by WFB-MCSSOR which has at least one order with a tardiness of 0.09 hours. When comparing agent 1b and W20-MCSSOR, it can be seen that the distribution of orders picked is slightly more skewed to the right for the heuristic, causing significant lower earliness. However, this comes with an increase in number of tardy orders and maximum tardiness.

Unlike the observations in lower demanding scenarios where waiting heuristics are grouped together, we see here that batching heuristics perform similar. This can be seen in Figure 6.11, where both the decision-makers use FIFO. The performance is very similar, with IP having less average tardiness.

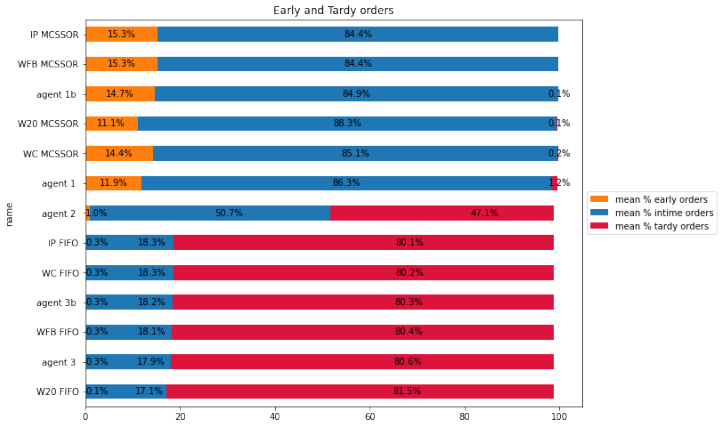


Figure 6.10: Performance comparison of agents and heuristics under *high* demand arrival.

Of all agents, 1b has the least amount of tardiness, and 1 has the most orders picked in-time. Agent 2 drastically increases the amount of tardy orders but both agents 3 and 3b increase this even more. Putting their performance close to all the FIFO heuristics.

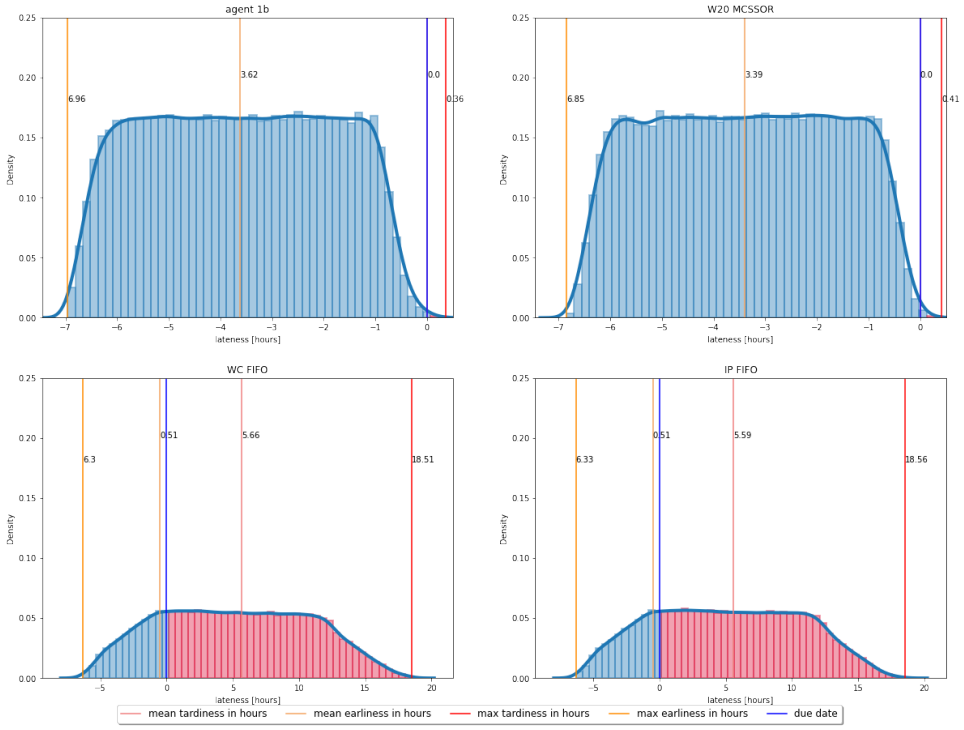


Figure 6.11: Distribution of picking completion time of orders along the due-date for *high* arrival.

In Figure E.1, we see that IP-MCSSOR has the lowest cost per order, followed by WFB-MCSSOR,W20-MCSSOR and agent 1b, respectively. We can also see that a large number of tardy orders for agents 3 and 3b and the FIFO heuristics cause a drastic decrease in the reward, meaning a substantial increase in cost.

As stated, agent 1b picks less tardy orders than W20-MCSSOR. On a yearly basis this would cause around 251 and 837 tardy orders respectively. Both these numbers are small compared with more than 836000 tardy orders per year IP-FIFO and WC-FIFO would cause. The reduction

of tardiness for agent 1b comes with a price of additional holding cost. For agent 1b the total earliness per day increases to with 566 hours compared to W20-MCSSOR.

6.1.6 Real Demand

Due to the combination of different intervals between cut-off times, and the variance in arrival rate, the distributions shown in Figure 6.13 have a unique pattern. From Figures C.12 and 6.12, it can be concluded that MCSSOR heuristics cause significantly less tardiness. As a consequence of this, the distribution is pushed more to the left of the due date and therefore cause an increase in earliness. Of these heuristics, W20-MCSSOR seems to have the highest number of orders picked in-time. Compared to this heuristic, agent 1b has significantly less orders in-time, but has half the number of tardy orders. Both WC-FIFO and IP-FIFO have significantly more tardy but also in-time orders. A significant decrease in maximum earliness and mean earliness per order can be seen when compared to the other graphs in Figure 6.13.

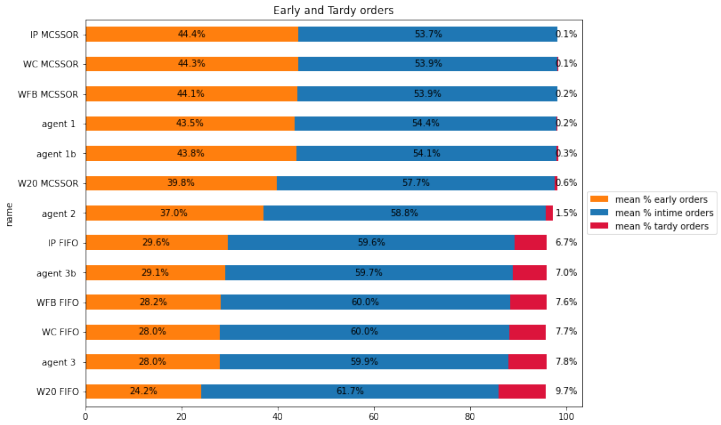


Figure 6.12: Performance comparison of agents and heuristics under *real* demand arrival.

Agent 1 has slightly less tardiness and slightly more in-time orders than 1b, making agent 1 the best performing in real arrival, but closely followed by 1b. Agent 2 introduces a significant amount of tardy but also in-time orders. Agent 3 and 3b repeat this, but with significantly more tardy than in-time orders.

In Figure E.1, we see that the low tardiness translates to lower cost for IP-MCSSOR, WFB-MCSSOR,WC-MCSSOR, and agent 1. Agent 2 seems to make up for it's tardy orders with having less earliness, and therefore gets an lower average cost per order than agent 1b.

Based on Figure 6.13 we can see that WC-FIFO had the lowest earliness related cost of the decision-makers shown in the figure. IP-FIFO already shows an increase of more than 157 hours of earliness per day. W20-MCSSOR shows even more than 1646 hours increase and agent 1b tops it all with an increase of 1904 hours of earliness per day. This earliness comes with a cost of tardy orders however. On a yearly basis the current practises of Kallax (WC-FIFO), would leave around 40250 tardy orders per year. IP-FIFO could decrease this to around 35000. Both these values are large compared to the 3140 tardy orders of W20-MCSSOR or the 1570 orders of agent 1b, a decrease of more than 96%.

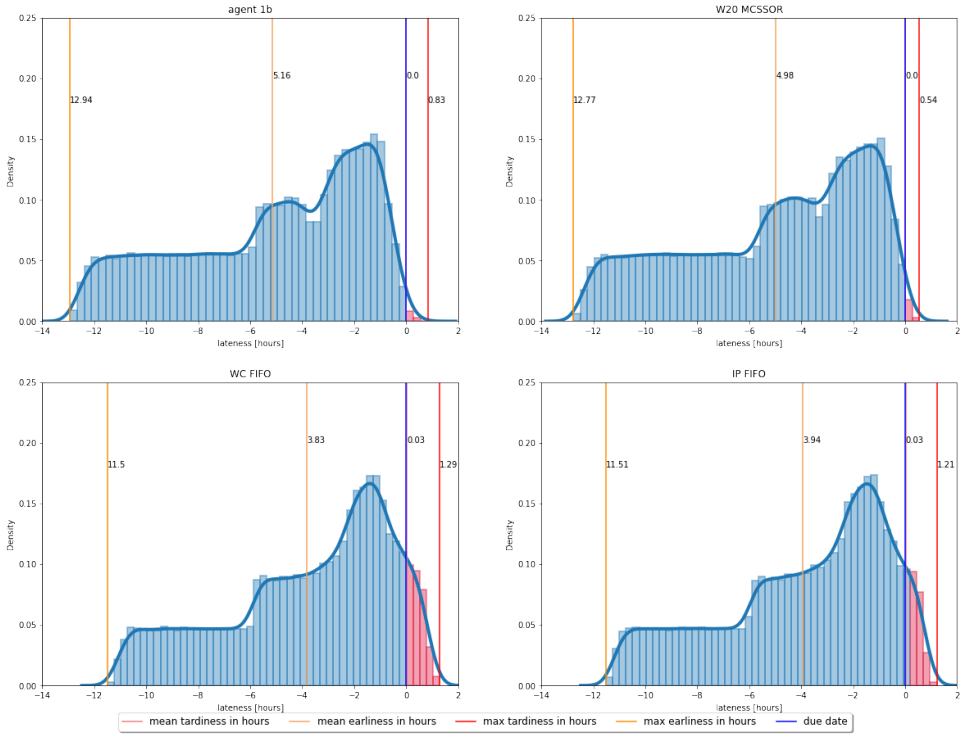


Figure 6.13: Distribution of picking completion time of orders along the due-date for *real* arrival.

6.1.7 Peak Demand

The percentages of early, tardy and in-time orders are shown in Figure C.15 per day of the simulation. On day 1, agent 3 and 3b already show tardiness while the first real peak is on day 2. Agent 2 starts to have tardy orders at the first peak, but significantly less than agents 3 and 3b. All decision-makers show tardiness on day 8, which is the first *normal* day, with no extended due dates. So in all cases the bullwhip effect of the peak is still in the system. Agent 1b shows the most in-time orders picked on days 9 and 10. On day 11, agents 1 and 1b, and all MCSSOR heuristics show the expected performance for a *normal* day. Agent 2 is fully recovered by day 13. Agents 3 and 3b, and all FIFO heuristics do not recovered by the end of the simulation, which is more than a week after the last peak.

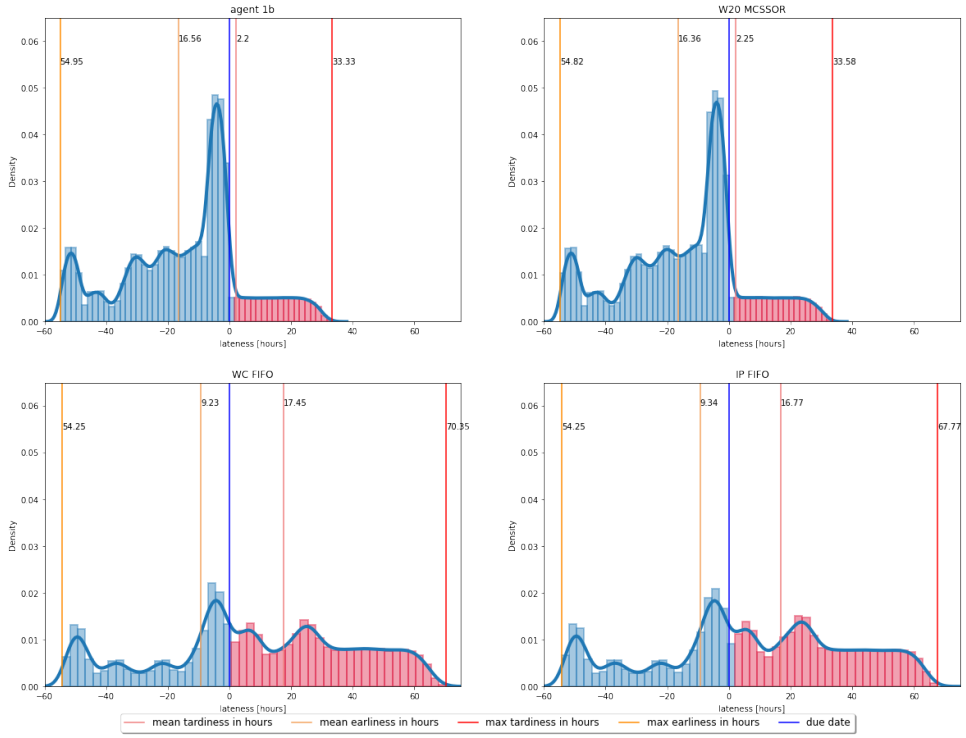


Figure 6.14: Lateness distribution for the full simulation of peak demands

Figure 6.14 shows that both heuristics WC-FIFO and IP-FIFO have significantly more tardiness. Overall the lateness is more evenly distributed, with a lower earliness than the other two graphs. W20-MCSSOR seems to have the distribution slightly more right of the due date, with a slightly higher maximum and average tardiness. This also gives a slightly lower maximum and average earliness. Overall, agent 1b seems to recover the fastest, with having the least tardiness on non peak days. Once W20-MCSSOR is also recoverd, it shows a bit more in-time orders as is seen in the *normal* demand analyses.

6.2 Experiment 2

We measured the effect of both the "what" (A) and "when" (B) decision by the means of a full factorial analysis. In Figures 6.15a and 6.15b, the effect can be seen on earliness and tardiness for different demands, respectively. Figure 6.15c shows the range on which the average earliness and tardiness per order per episode occurred.

To determine whether the gathered results are statistically significant, we repeat each test 10 times. We define e_j^i as the main effect of decision j for replication i (with $i \in (1, 10)$), calculated according to the formulas in Section 5.2. The mean effect will be calculated according to the equation below:

$$\bar{e}_j = \frac{\sum_{i=1}^{n} e_j^i}{n} \tag{6.1}$$

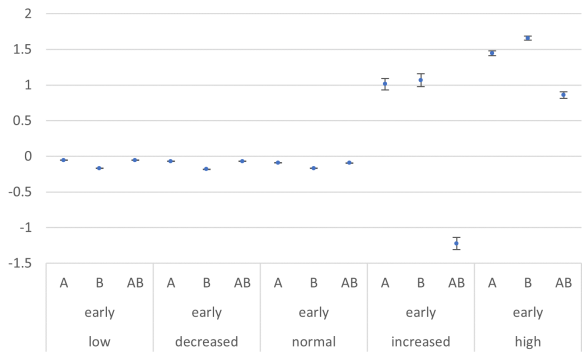
We also define S_j^2 (variance):

$$S_j^2 = \frac{\sum_{i=1}^{n} \left[e_j^i - (\bar{e})_j \right]^2}{n-1} \tag{6.2}$$

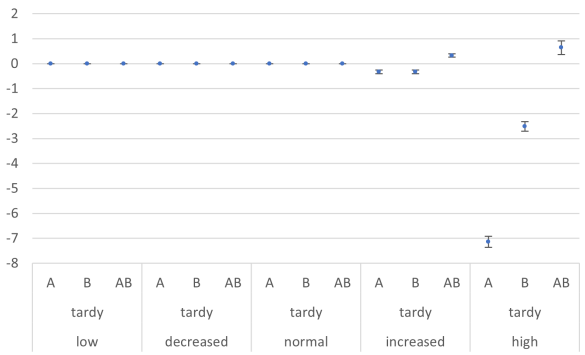
Finally we define a confidence interval for the expected main effect ($\mathbb{E}(e_j)$) with the use of the student-t value ($t_{n-1,1-\alpha/2}$). This is an approximate interval for a confidence of $100(1-\alpha)$ percent confidence.

$$\bar{e}_j \pm t_{n-1,1-\alpha/2} \sqrt{S_j^2/n} \tag{6.3}$$

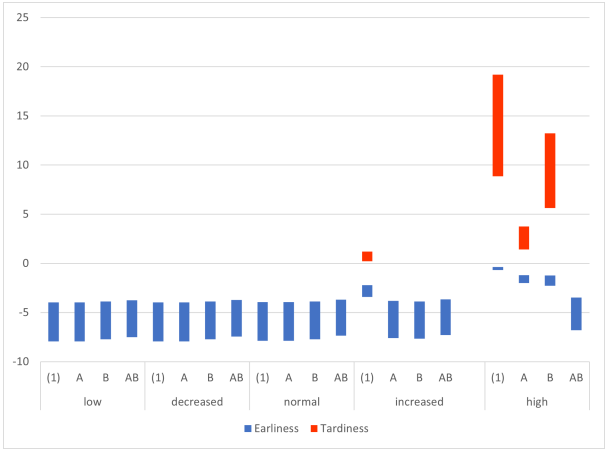
If the confidence interval for $\mathbb{E}(e_j)$ contains 0, we conclude that the results are not significant. Otherwise, we can conclude that the measured effect is significant (Law, 2015). The error-bars for these intervals are plotted in Figure 6.15a and 6.15b. Since the interval is very small in most cases, the results are also shown in Table C.1. For both decisions in *low*, *decreased* and *normal* demand, we do not measure any effect on tardiness. All other measurements are shown to be significant.



(a) Effect of earliness



(b) Effect of tardiness



(c) earliness and tardiness per optimised decision, with (1) no optimisation

Figure 6.15: Results of the 2^2 factorial design

From these figures we can make the following inferences:

1. Effect on Earliness (Figure 6.15a)

   (a) For *low*, *decreased*, and *normal* demand scenarios, we see that the magnitude of the effect of B is larger than the effect of A. This means that the effect of B is more substantial. We can see that, if the arrival rate increases, the effect of both decisions become larger. This is more clear in Table C.1.

   (b) For *increased* and *high* demands, we see a drastic increase in the magnitudes of effects, meaning that the effect of both decisions is more substantial for higher arrival rates. The effect of B is still the largest of the two in for both demands, but we see *increased* demand that the effect AB has an even larger magnitude. This means there is a dependency between the decisions.

2. Effect on Tardiness (Figure 6.15b)

   (a) For *low*, *decreased* and *normal* scenarios, the effect of both decisions is zero. This means that no effect is found for both decisions in scenarios with lower arrival rate.

   (b) For *increased* demand, we can see both decisions have an effect on tardiness. However, the effect is not greater as that on earliness, as can be seen in Figure 6.15a.

   (c) Under *high* demand scenario, we see a drastic increase in the magnitudes of both effects, with effect of decisions A being the most substantial for *high* demand. In both *increased* and *high* demand scenarios, A has the largest effect.

6.3 Strategy Analyses

In this section we take a closer look at the behaviour of the agents. We try to get a better understanding of the strategies they use for the on-line order batching process.
For each analysis, we use the plots in Appendix D, the statistics in Table 6.1, and Figure 6.16. In this last figure, we show what the robot is doing on a day under each kind of decision-maker. We have split the waiting action into waiting with and without orders in the queue. We do this to distinguish when an agent is "deliberately" waiting (wait) or if it has to wait since there are no other options (empty queue).

6.3.1 Agent 1

Under agent 1, most of the time when the robot is inactive, there are no items in the queue (Figure 6.16). In Table 6.1, we see that agent 1 seldom opts for a waiting action. This makes the strategy look a lot like IP. This behaviour is also apparent in Figures D.4 and D.5. As soon as an order arrives, it is picked, whereas, under other decision-makers, we see that orders spend more time in the queue.

In Table 6.1, we can see that for *high* arrival, the agent acts more like FIFO than for *low* arrival. We can also see that for low arrival, the average batch size is 1. This means that the heuristics do not influence the batched orders since only one order will be picked at all times. For low arrival, we look at Figure D.6 to see if any logic for the increased use of FIFO can be found. Spotting when FIFO is used is difficult. A few indications that could point to FIFO being used are longer pick times and orders that have been in the queue for a longer time. One candidate

could be the batch that is done picking at 11.30. No apparent reason for the increased use of FIFO could be found.

The agent is not sensitive to the due dates of the orders since no FIFO is used, mainly because the strategy does not seem to change once the cut off time is passed. Orders that arrive after this cut-off have a different due date, larger than the ones arriving before. If the agent considered due dates in the strategy, we would expect to see some changes in the strategy here.

This strategy could be best described as a combination of IP with a preference for MCSSOR.

6.3.2 Agent 1b

Agent 1b waits a lot in comparison to agent 1. Overall, agent 1b waits the most. The agent only opts to use MCSSOR and does not use any FIFO. From Figure 6.16, we can see that the agent's wait and pick pattern is most comparable with that of a WC of WFB heuristic. In the last plot, however, we can see that the strategy differs a bit since it spends some more time waiting than the heuristics during periods of low demand on a day with variable arrival.

In the Figures D.4 and D.5, we observe that the agent seems to pick within a somewhat fixed time interval. For *normal* arrival, this time interval seems to be smaller and vary a bit more than for *low*, and it does not wait anymore in *high* demand scenarios.

This strategy could be best described as a time window batching combined with MCSSOR, where the time window is adaptive to the demand.

6.3.3 Agent 2

Agent 2 shows the same trend for different demanding scenarios as agent 1b. In general, the batch sizes are smaller, and the agent waits less. Agent 2 also uses FIFO considerably more than the previous two agents, but uses is relatively little in the *high* demanding scenarios. This could mean that agent 2 uses smaller batch sizes and less efficient batching heuristics (FIFO) to decrease tardiness. This strategy is less effective for earliness in higher demanding environments, as can be seen in Figure D.6.

The strategy of agent 2 combines waiting, FIFO, and smaller batch sizes too, presumably, reduce earliness.

6.3.4 Agent 3

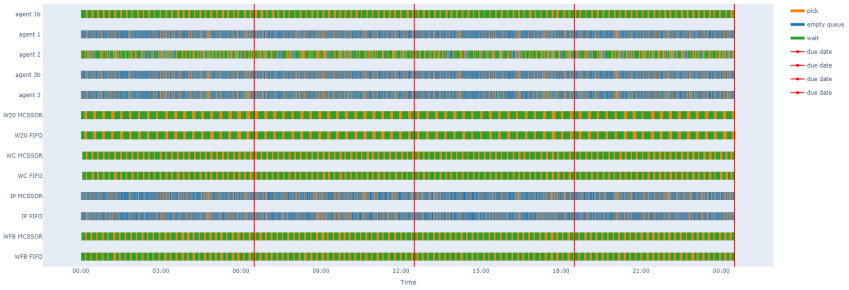
In Table 6.1, we see that agent 3 only opts for waiting for 13% of the decision-moments in *low* arrival. We see that this number decreases for higher arrival rates, meaning that the agent has some sort of understanding of its environment. In Figure D.4, we can see that the agent, in some cases, does wait with orders in the queue for a short amount of time. In Figures D.6 and D.3, we can see that in *high* demand, the agent is constantly picking. In Table 6.1, we can see that it does this with one of the smaller batch sizes of the decision-makers. This might be one of the reasons for the large number of tardy orders in this scenario.

Unfortunately, for both agents 3 and 3b, finding insights on the batching strategy is difficult due to its complexity. We can only draw a conclusion on the waiting heuristic, which is, in this case, imitates an IP with some added waiting depending on the demand.

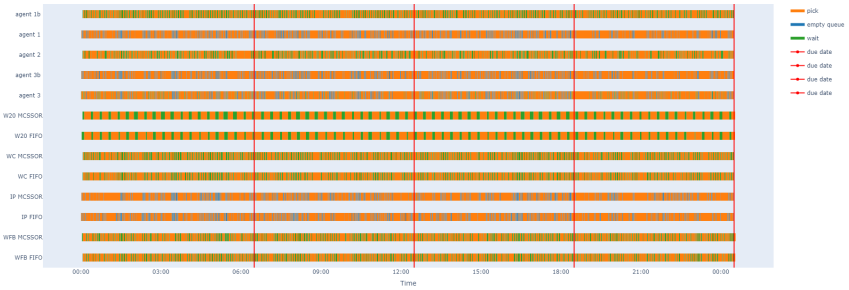
6.3.5 Agent 3b

In Table 6.1, we see that in all scenarios, agent 3b tries to pick orders 100% of the decision-moments. This makes it effectively IP since an order is picked as soon as an it arrives. We can see that agent 3b does make larger batches than agent 3, which might be the reason for the slightly better performance on the tardiness of this agent.

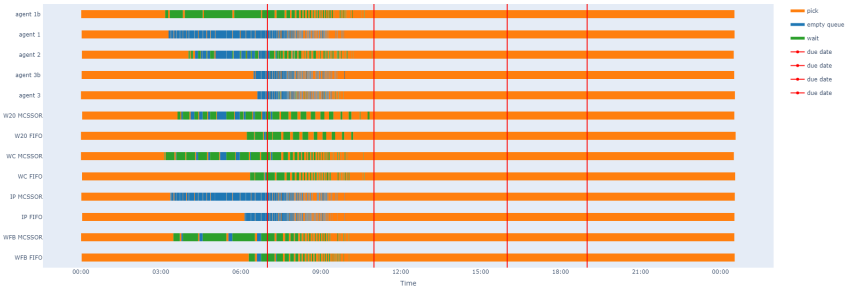
Again, we did not succeed in finding any insights on the batching heuristics, but the waiting strategy is ultimately IP.



(a) *Low* demand



(b) *Normal* demand



(c) Variable demand

Figure 6.16: The robots' activity over time in different demanding environments.

		agent 1	agent 1b	agent 2	agent 3	agent 3b	W20-MCSSOR	IP-FIFO
Low	**Wait**	0.01	95.26	80.76	13.59	0.00	93.96	79.44
	Pick	99.99	4.738	19.24	86.41	100.00	6.04	20.56
	FIFO	8.24	0.00	9.73	x	x	x	20.56
	MCSSOR	91.75	4.74	9.52	x	x	6.04	x
	Batch size	1.00	5.00	1.73	1.00	1.00	4.00	1.00
Normal	**Wait**	0.03	80.12	66.39	4.33	0.00	78.69	38.76
	Pick	99.97	19.88	33.61	95.67	100.00	21.31	61.24
	FIFO	5.75	0.00	22.02	x	x	x	61.24
	MCSSOR	94.22	19.88	11.59	x	x	21.31	x
	Batch size	1.06	5.00	3.01	1.11	1.06	4.69	1.06
High	**Wait**	0.00	0.73	0.60	0.00	0.00	2.71	0.15
	Pick	100.00	99.27	99.40	100.00	100.00	97.29	99.85
	FIFO	11.25	0.00	39.37	x	x	x	99.85
	MCSSOR	88.75	99.27	60.03	x	x	97.29	x
	Batch size	4.99	5.00	3.90	4.69	4.97	5.00	4.99
Real	**Wait**	0.02	66.28	42.16	6.48	0.00	65.14	42.77
	Pick	99.98	33.72	57.84	93.52	100.00	34.86	57.22
	FIFO	10.56	0.00	24.67	x	x	x	57.23
	MCSSOR	89.42	33.72	33.15	x	x	34.86	x
	Batch size	3.72	4.97	3.63	3.55	3.75	4.84	3.77

Table 6.1: Percentages of each action taken and average batch size

7 Conclusion and Discussion

In this section we conclude on the results of this project. Subsequently, we will discuss these results with managerial implications, the limitations of this project and possible future work.

7.1 Conclusion

In Appendix E, we have summarised all the results in matrices. Based on the analyses, we can conclude that W20-MCSSOR is the overall best heuristic approach. This decision-maker shows that the newly introduced FTWBC approach can outperform current practices and commonly used on-line order batching methods like FTWB.

Multiple agents have proven to be successful in at least one scenario. Among those, agent 1b is the best overall performing agent that can keep up with the performance of W20-MCSSOR with limited training time. Noteworthy is the performance of agent 2 in scenarios with *low* too *increased* demand. It can outperform the simpler agent 1, showing that a more complex agent shows potential, especially regarding minimising earliness.

Based on the analyses in Section 6.3, we can conclude that agents 3 and 3b have trouble understanding their action and/or observation space. The performance of agent 3 is significantly worse than those of simpler heuristics, where the only difference is the action space. Since the agents have (almost) the same waiting strategy as IP, the difference in performance is due to the batching strategy. Based on this, we can conclude that in the current implementation of the agents, DRL cannot solve a more complex version of the OOBP, where it needs to make both decisions fully independent.

In Section 3, we show that much research has been done on the OOBP. Most solve the problem using some form FTWB or VTWB combined with various degrees of complexity for the batching heuristics. Very few works consider earliness in the OOBP. We have found no papers that focus on minimising tardiness and earliness as an objective in a dynamic environment with fluctuating demand. This answers our first sub-question: What are the existing methods for solving the problem of on-line batching?

Our second question; - *How does each of the decisions contribute to the overall objective?* - is answered in Section 6.2. We showed that the "when" decision has the most influence on earliness, and the "what" decision on tardiness. In higher arrival rates, the influence of both decisions becomes larger, with the latter one having significantly more effect. In the current setup of the experiment, both decisions have no effect on tardiness in lower demanding scenarios, making "when" more critical in these cases and "what" more critical once the arrival rate increases.

Next, we answer the question; *How does the batching agent perform compared to current practices?*. Agent 1b can outperform all the traditional baselines that we tested against in most cases. In all cases, it can outperform the current WC-First In First Out (FIFO) practice that Kallax is currently using.

Lastly, we check; *How does the batching agent adapt to changes?* In Section 6.3 we show that most of the agents can adapt to changes in the environment. Agent 1b adapts the time between picking actions and the number of waiting actions based on the demand it is dealing with. Agent 1b is also the first to recover from the bullwhip effect caused by the peak demands.

Finally, we address the main research question of this project:
How can the introduction of a machine learning agent optimise the on-line batching for an automated picking system?
The on-line order batching process can be optimised by the introduction of a DRL using PPO. When the agent is given an action space where it can choose between waiting or picking with different heuristics, the agent shows a good performance. When using a reward function that is not normalised for the arrival rate, letting the agent select the batch size shows significant improvement in most scenarios.

Based on the obtained results in Section 6.1, we can state that we have successfully implemented a DRL agent to tackle the OOBP in an environment with variable demand.

7.2 Discussion

7.2.1 Implications

The experiment with realistic demand could be seen as a simplified simulation of reality. Based on the results gathered in this experiment, we can say that agents 1, 1b and 2 show potential for implementation in the setup of Picker.AI in the warehouse of Kallax, which is currently using WC-FIFO. These agents will significantly decrease the number of tardy orders by the robot at a small cost of increased earliness. If Kallax prioritises in-time orders over tardy orders but still want to decrease their tardiness, we would recommend agent 2. This will decrease the number of orders picked in time from 60 % to 58.8% and decrease tardiness from 7% to 1.5%. W20–MCSSOR would be the most fitting heuristic, with the focus a bit more on decreasing tardiness compared with agent 2. If tardiness is the highest priority, we recommend agent 1b, since it has a similar performance as agent 1, but is more adaptive to changes. Another option could be upgrading their current WC-FIFO heuristic to a WC-MCSSOR or IP-MCSSOR, which will drastically decrease the number of tardy orders.

7.2.2 Limitations

For most DRL problems, the possibilities are endless. In this project, many different features for the observation space have been investigated and in all kinds of combinations. The same is true for the reward function. Any new feature or different setting in the environment gives rise to new possible combinations and exponentially increases the time needed to test everything. This calls for efficient testing and assumptions to find combinations of settings and features that work well. Each of these tests depends on the hyperparameters of the training model. After hyperparameter tuning, it seems likely that the default parameters of Stable Baselines that we used are not sufficient for most of the settings that we tested for. This might have caused us to reject models that would have given good results for different hyperparameters.

We also recognise that the training time set for the agents might have been too little to develop complex strategies and cause the agent to settle for a local optimum. If we look at the closely related work of Cals et al. (2021), we see that the agent is trained for $7.5 \cdot 10^5$ time-steps. The model they train has a complexity comparable with agent 1 or 2. Both these models are trained for less than $7.5 \cdot 10^5$ time-steps. We selected the number of time-steps based on the point where the agent's reward started to converge, and no more improvement was found. However, if longer runs are done (also with different hyperparameters), maybe the agent will show the capability of learning more complex strategies than we have seen.

Lastly, we want to mention that most of the exploration for the reward function and state-space has been done using agent 1. Due to time constraints, we assumed that the settings would be sufficient for the other agents. However, it might be possible that the other agents, agent 2 and 3, could perform better if the state space and/or reward function were slightly modified.

7.2.3 Future Work

The current methods show promising results, but a lot could be done to improve this. Our first suggestion would be to create an agent 2b. Agent 2 shows better performance than agent 1, with the only difference in the action space. Agent 1b shows even better performance, with the only difference from agent 1 in the reward function. Combining the reward function of 1b with the action space of 2 might show an even better performance.

In the strategy analyses, we show that agent 1b does not adapt its strategy to the due date. This is needed for better performance. To emphasise this due date, one could consider adjusting the formula of W_o to give an exponentially higher value for orders picked closer to the due date. Or one could drastically increase the reward given after ε.

As mentioned before, agents 3 and 3b seem to have difficulty with order batching. The complexity of this problem is significantly larger than the complexity of "when" to batch. This can be seen in the larger action space of agents 3 and 3b with more than 57 million combinations. This, combined with the larger observation space of agent 3b, will give many more possible combinations, causing the agent to need more samples to learn from, drastically increasing the computational time and power needed to train. In theory, the best practice would be to start picking as late as possible and create batches that are picked as fast as possible. Finding a strategy like this would be extremely hard for the agent since, with a multi-objective, the decisions are codependent. Two decisions need to be optimised for two objectives. Moreover, the objectives are contradicting, since decreasing earliness might cause an increase in tardiness. All these combined problems make the current approach of complex agents unsuitable for finding a solution. So a possible solution might be to create a multi-agent approach, where both decisions are separated and have their own objective. Agent 1b has already proven to be successful in learning when to wait or pick to decrease earliness while keeping tardiness at a minimum. This means that we only need the second agent to learn how to batch. This can be done with a reward function that rewards shorter pick-times for larger batches and penalises tardiness. This agent can be trained in a simpler training environment, with shorter episodes, which will significantly improve the computational speed, and can therefore learn from more samples. Once this agent is trained, we can train the "when" agent using a simpler observation space and an action space where it can only choose between wait or pick. While the "when" agent wants to pick, the "what" agent is used to create the batch.

This approach changes the problem of learning through one agent 2 decisions with 2 objectives into a problem of two agents, each with a multi-objective but a single decision. Separating the training environments of these agents might make the translation between the taken decision and the received reward easier to understand.

References

Github - dlr-rm/rl-baselines3-zoo: A training framework for stable baselines3 reinforcement learning agents, with hyperparameter optimization and pre-trained agents included. *https://github.com/DLR-RM/rl-baselines3-zoo*.

Github - dlr-rm/stable-baselines3: Pytorch version of stable baselines, reliable implementations of reinforcement learning algorithms. *https://github.com/DLR-RM/stable-baselines3*.

D. E. Akyol and G. M. Bayhan. Multi-machine earliness and tardiness scheduling problem: An interconnected neural network approach. *International Journal of Advanced Manufacturing Technology*, 37(5-6):576–588, 2008. ISSN 02683768. doi: 10.1007/s00170-007-0993-0.

M. E. Aydin and E. Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3):169–178, nov 2000. ISSN 0921-8890. doi: 10.1016/S0921-8890(00)00087-7.

M. Beeks, R. Refaei Afshar, Y. Zhang, R. Dijkman, C. van Dorst, and S. de Looijer. Deep reinforcement learning for a multi-objective online order batching problem. In *Proceedings of the International Conference on Automated Planning and Scheduling*. Association for the Advancement of Artificial Intelligence (AAAI), 2022.

R. N. Boute, J. Gijsbrechts, W. van Jaarsveld, and N. Vanvuchelen. Deep reinforcement learning for inventory control: A roadmap. *European Journal of Operational Research*, 298 (2):401–412, 2022. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2021.07.016. URL *https://www.sciencedirect.com/science/article/pii/S0377221721006111*.

G. E. Box, J. Hunter, and W. Hunter. Statistics for experimenters: Design, innovation, and discovery., 2005.

N. Boysen, R. de Koster, and F. Weidinger. Warehousing in the e-commerce era: A survey. *European Journal of Operational Research*, 277(2):396–411, sep 2019a. ISSN 03772217. doi: 10.1016/j.ejor.2018.08.023.

N. Boysen, K. Stephan, and F. Weidinger. Manual order consolidation with put walls: the batched order bin sequencing problem. *EURO Journal on Transportation and Logistics*, 8(2):169–193, jun 2019b. ISSN 21924384. doi: 10.1007/s13676-018-0116-0. URL *https://doi.org/10.1007/s13676-018-0116-0*.

B. Cals, Y. Zhang, R. Dijkman, and C. van Dorst. Solving the online batching problem using deep reinforcement learning. *Computers and Industrial Engineering*, 156(2020), 2021. ISSN 03608352. doi: 10.1016/j.cie.2021.107221.

B. J. H. C. Cals. The order batching problem a deep reinforcement learning approach. Master's thesis, Technische Universiteit Eindhoven, 2019.

Ç. Cergibozan and A. S. Tasan. Order batching operations: an overview of classification, solution techniques, and future research, 2019. ISSN 15728145.

H. Dong, Z. Ding, and S. Zhang. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Springer, 2020. ISBN 9789811540950.

M. Drakaki and P. Tzionas. Manufacturing scheduling using Colored Petri Nets and reinforcement learning. *Applied Sciences (Switzerland)*, 7(2), 2017. ISSN 20763417. doi: 10.3390/app7020136.

E. Elsayed, M.-K. Lee, S. Kim, and E. Scherer. Sequencing and batching procedures for minimizing earliness and tardiness penalty of order retrievals. *International Journal of Production Research*, 31(3):727–738, 1993. doi: 10.1080/00207549308956753. URL *https://doi.org/10.1080/00207549308956753*.

eMarketer Editors. Worldwide ecommerce will approach $5 trillion this year, 2021. URL *https://www.emarketer.com/content/worldwide-ecommerce-will-approach-5-trillion-this-year*.

M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017. URL *http://arxiv.org/abs/1706.10295*.

K. Geevers. Deep Reinforcement Learning in Inventory Management. Master's thesis, 2020.

S. Gil-Borrás, E. G. Pardo, A. Alonso-Ayuso, and A. Duarte. Fixed versus variable time window warehousing strategies in real time. *Progress in Artificial Intelligence 2020 9:4*, 9 (4):315–324, aug 2020. ISSN 2192-6360. doi: 10.1007/S13748-020-00215-1. URL *https://link.springer.com/article/10.1007/s13748-020-00215-1*.

S. Gil-Borrás, E. G. Pardo, A. Alonso-Ayuso, and A. Duarte. A heuristic approach for the online order batching problem with multiple pickers. *Computers & Industrial Engineering*, 160:107517, 2021. ISSN 0360-8352. doi: https://doi.org/10.1016/j.cie.2021.107517. URL *https://www.sciencedirect.com/science/article/pii/S0360835221004216*.

N. Hall, W. Kubiak, and S. Sethi. Earliness-tardiness scheduling problems, ii: Deviation of completion times about a restrictive common due date. *Operations Research*, 39, 10 1991. doi: 10.1287/opre.39.5.847.

N. G. Hall and M. E. Posner. Earliness-tardiness scheduling problems, i: Weighted deviation of completion times about a common due date. *Operations Research*, 39(5):836–846, 1991. ISSN 0030364X, 15265463. URL *http://www.jstor.org/stable/171212*.

S. Henn. Algorithms for on-line order batching in an order picking warehouse. *Computers and Operations Research*, 39(11):2549–2563, 2012. ISSN 03050548. doi: 10.1016/j.cor.2011.12.019. URL *http://dx.doi.org/10.1016/j.cor.2011.12.019*.

S. Henn, S. Koch, and G. Wäscher. Order batching in order picking warehouses: a survey of solution approaches. In *Warehousing in the global supply chain*, pages 105–137. Springer, 2012. doi: 10.1007/978-1-4471-2274-6_6.

M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL *http://arxiv.org/abs/1710.02298*.

Y.-f. Hung, J.-s. Bao, and Y.-e. Cheng. Minimizing earliness and tardiness costs in scheduling jobs with time windows. *Computers & Industrial Engineering*, 113:871–890, 2017. ISSN 0360-8352. doi: 10.1016/j.cie.2016.12.023. URL *https://doi.org/10.1016/j.cie.2016.12.023*.

X. Jiang, Y. Zhou, Y. Zhang, L. Sun, and X. Hu. Order batching and sequencing problem under the pick-and-sort Order batching and sequencing problem under the pick-and-sort strategy in online supermarkets strategy in online supermarkets. *Procedia Computer Science*, 126: 1985–1993, 2018. ISSN 1877-0509. doi: 10.1016/j.procs.2018.07.254. URL *https://doi.org/10.1016/j.procs.2018.07.254*.

I. Kaynov. Deep Reinforcement Learning for Asymmetric One-Warehouse Multi-Retailer Inventory Management. Master's thesis, Technische Universiteit Eindhoven, 2020.

A. Khodadadi, P. Fakhari, and J. R. Busemeyer. Learning to maximize reward rate: A model based on semi-Markov decision processes. *Frontiers in Neuroscience*, 8(8 MAY), 2014. ISSN 1662453X. doi: 10.3389/fnins.2014.00101. URL */pmc/articles/PMC4033239//pmc/articles/PMC4033239/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC4033239/*.

S. Lang, F. Behrendt, N. Lanzerath, T. Reggelin, and M. Muller. Integration of Deep Reinforcement Learning and Discrete-Event Simulation for Real-Time Scheduling of a Flexible Job Shop Production. In *Proceedings - Winter Simulation Conference*, number Gershwin 2018, pages 3057–3068, 2020. ISBN 9781728194998. doi: 10.1109/WSC48552.2020.9383997.

A. M. Law. *Simulation modeling and analysis*. McGraw-Hill, 2015. ISBN 9780073401324.

K. H. Leung, C. K. Lee, and K. L. Choy. An integrated online pick-to-sort order batching approach for managing frequent arrivals of B2B e-commerce orders under both fixed and variable time-window batching. *Advanced Engineering Informatics*, 45:101125, aug 2020. ISSN 1474-0346. doi: 10.1016/J.AEI.2020.101125.

M. P. Li, P. Sankaran, M. E. Kuhl, R. Ptucha, A. Ganguly, and A. Kwasinski. Task selection by autonomous mobile robots in a warehouse using deep reinforcement learning. In *2019 Winter Simulation Conference (WSC)*, pages 680–689, 2019. doi: 10.1109/WSC40007.2019.9004792.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *Nature*, 388:539–547, dec 2013. URL *http://arxiv.org/abs/1312.5602*.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 14764687. doi: 10.1038/nature14236.

I. Muter and T. Öncan. Order batching and picker scheduling in warehouse order picking. *IISE Transactions*, 2022. doi: 10.1080/24725854.2021.1925178.

B. M. Nataraja, Z. Atan, and I. J. B. F. Adan. Integrated decisions for robot-based order picking/fulfillment systems. 2022.

OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. (December), 2019. URL *http://arxiv.org/abs/1912.06680*.

J. Oxenstierna. Warehouse vehicle routing using deep reinforcement learning, 2019.

O. Ozturk, M. A. Begen, and G. S. Zaric. A branch and bound algorithm for scheduling unit size jobs on parallel batching machines to minimize makespan. *International Journal of Production Research*, 55(6):1815–1831, 2017. ISSN 1366588X. doi: 10.1080/00207543.2016.1253889. URL *https://www.tandfonline.com/doi/full/10.1080/00207543.2016.1253889?casa_token=2KbalEOFGAAAAAAA%3AAPjdYASYoVMDV-u2_xAXKZsgWdcMFJJwlP5pA7eye_P2me91P5Bo26HYxR1yF25MN9AQrVSjniFmEA*.

C. D. Paternina-Arboleda and T. K. Das. A multi-agent reinforcement learning approach to obtaining dynamic control policies for stochastic lot scheduling problem. *Simulation Modelling Practice and Theory*, 13(5):389–406, 2005. ISSN 1569190X. doi: 10.1016/j.simpat.2004.12.003.

R. Pérez-rodríguez and A. Hernández-aguirre. A continuous estimation of distribution algorithm for the online order-batching problem. *The International Journal of Advanced Manufacturing Technology*, pages 569–588, 2015. doi: 10.1007/s00170-015-6835-6.

I. S. Peyas, Z. Hasan, M. R. Rahman Tushar, A. Musabbir, R. M. Azni, and S. Siddique. Autonomous warehouse robot using deep q-learning. *TENCON 2021 - 2021 IEEE Region 10 Conference (TENCON)*, Dec 2021. doi: 10.1109/tencon54134.2021.9707256. URL *http://dx.doi.org/10.1109/TENCON54134.2021.9707256*.

Pickr.ai. PICKR.AI, 2022. URL *https://www.pickr.ai/*.

A. R. F. Pinto and M. S. Nagano. An approach for the solution to order batching and sequencing in picking systems. *Production Engineering 2019 13:3*, 13(3):325–341, apr 2019. ISSN 1863-7353. doi: 10.1007/S11740-019-00904-4. URL *https://link.springer.com/article/10.1007/s11740-019-00904-4*.

H. Rummukainen and J. K. Nurminen. Practical Reinforcement Learning -Experiences in Lot Scheduling Application. *IFAC-PapersOnLine*, 52(13):1415–1420, jan 2019. ISSN 2405-8963. doi: 10.1016/J.IFACOL.2019.11.397.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015. URL *http://arxiv.org/abs/1511.05952*. cite arxiv:1511.05952Comment: Published at ICLR 2016.

J. Schulman, M. Jordan, S. Levine, P. Moritz, and P. Abbeel. Trust Region Policy Optimization, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017. URL *http://arxiv.org/abs/1707.06347*.

O. Sigaud and O. Buffet. *Markov Decision Processes in Artificial Intelligence: MDPs, beyond MDPs and applications*. John Wiley & Sons, 2013. ISBN 9781848211674. doi: 10.1002/9781118557426.

V. d. N. Silva and L. Chaimowicz. MOBA: a New Arena for Game AI. *ArXiv*, 2017. URL *http://arxiv.org/abs/1705.10443*.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. ISSN 14764687. doi: 10.1038/nature16961. URL *http://dx.doi.org/10.1038/nature16961*.

SimPy. Documentation for SimPy — SimPy 4.0.2., 2020. URL *https://simpy.readthedocs.io/en/latest/contents.html*.

R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, volume 2. l, 2020. ISBN 9780262039246. doi: 10.1108/k.1998.27.9.1093.3.

C.-Y. Tsai, J. J. H. Liou, and T.-M. Huang. Using a multiple-ga method to solve the batch picking problem: considering travel distance and order due time. *International Journal of Production Research*, 46(22):6533–6555, 2008. doi: 10.1080/00207540701441947. URL *https://doi.org/10.1080/00207540701441947*.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL *http://arxiv.org/abs/1509.06461*.

I. Van Nieuwenhuyse and R. B. de Koster. Evaluating order throughput time in 2-block warehouses with time window batching. *International Journal of Production Economics*, 121(2):654–664, oct 2009. ISSN 09255273. doi: 10.1016/j.ijpe.2009.01.013.

I. Van Nieuwenhuyse, R. de Koster, and J. Colpaert. Order batching in multi-server pick-and-sort warehouses. *DTEW-KBI_0731*, pages 1–29, 2007.

O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. StarCraft II: A New Challenge for Reinforcement Learning. *ArXiv*, 2017. ISSN 2331-8422. URL *http://arxiv.org/abs/1708.04782*.

Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL *http://arxiv.org/abs/1511.06581*.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, may 1992. ISSN 0885-6125. doi: 10.1007/bf00992698. URL *http://link.springer.com/10.1007/BF00992698*.

S. Xie, T. Zhang, and O. Rose. Online Single Machine Scheduling Based on Simulation and Reinforcement Learning. In *Simulation in Produktion und Logistik 2019*, 2019.

H. Zeng, Y. Liu, D. Zhang, K. Han, and H. Hu. A deep reinforcement learning approach for constrained online logistics route assignment. *CoRR*, abs/2109.03467, 2021. URL *https://arxiv.org/abs/2109.03467*.

J. Zhang, X. Wang, and K. Huang. Integrated on-line scheduling of order batching and delivery under B2C e-commerce. *Computers and Industrial Engineering*, 94:280–289, apr 2016. ISSN 03608352. doi: 10.1016/j.cie.2016.02.001.

J. Zhang, X. Wang, J. Ruan, and F. T. Chan. On-line order batching and sequencing problem with multiple pickers: A hybrid rule-based algorithm. *Applied Mathematical Modelling*, 45: 271–284, 2017a. ISSN 0307904X. doi: 10.1016/j.apm.2016.12.012. URL *http://dx.doi.org/10.1016/j.apm.2016.12.012*.

T. Zhang, S. Xie, and O. Rose. Real-time batching in job shops based on simulation and reinforcement learning. *Proceedings - Winter Simulation Conference*, 2018-December:3331–3339, 2019. ISSN 08917736. doi: 10.1109/WSC.2018.8632524.

Y. Zhang, Y. Wu, and W. Ma. Seed combine accompanying selection rule of order-batching methods in a Multi-Shuttle Warehouse System. In *Proceedings - 2017 Chinese Automation Congress, CAC 2017*, volume 2017-January, pages 6228–6232. Institute of Electrical and Electronics Engineers Inc., dec 2017b. ISBN 9781538635247. doi: 10.1109/CAC.2017.8243899.

Appendices

A Methods

A.1 Faulty Action Correction Process

Since the action space is not variable, agents need to predict values for all elements of the action space. For agents 3 and 3b, this means that the agent could select a waiting action and return orders to the batch. In this case, the created batch will be ignored, and a waiting action will be executed. However, if the agent decides to pick, some outputs need correction to keep the simulation working as supposed. The agent may select orders that do not exist. For example, if there are only five items in the queue, and the agent wants to select order number eight, this is impossible. In such a case, we ignore all infeasible orders and only select the subset of orders from the batch that are present in the queue. If no such feasible subset exists, the picking action is changed to a waiting action. If the action presented by the agent needs any of these corrections, the action is labelled as a faulty action. The way we correct the faulty selection of batch is shown in Figure A.1.

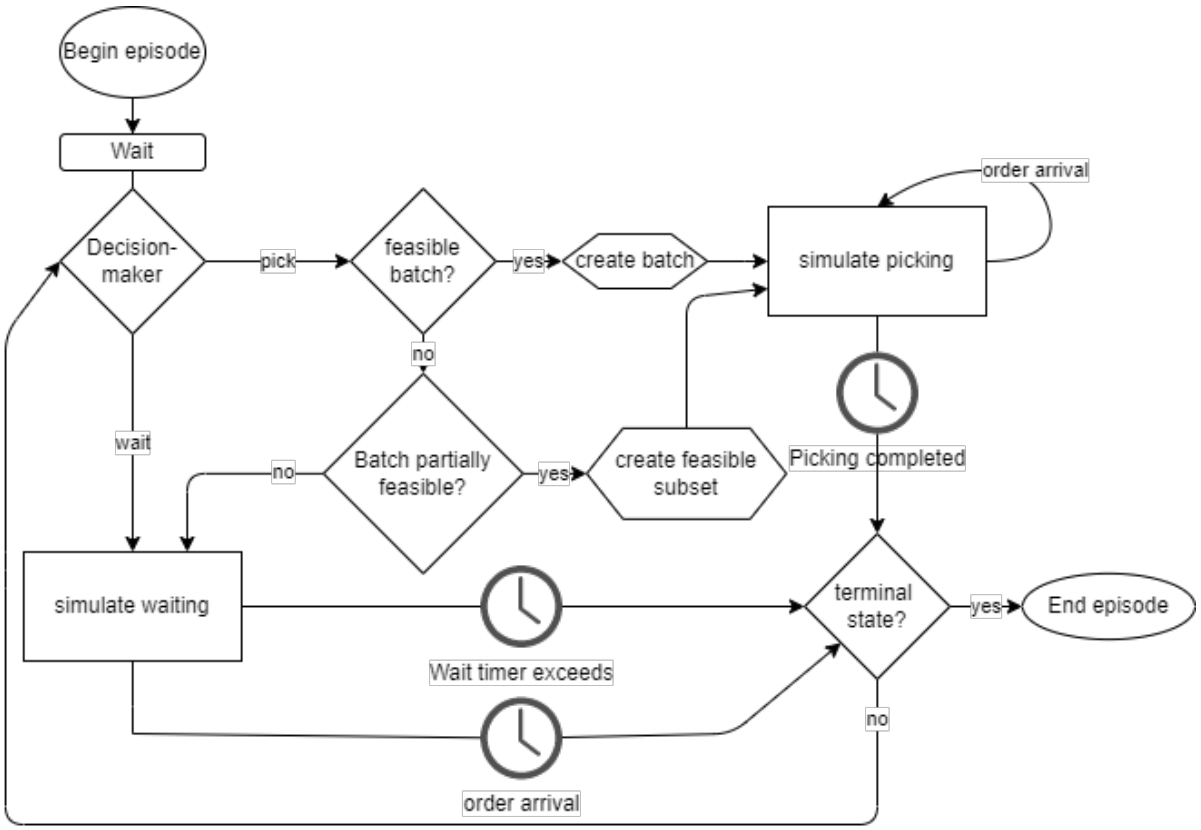


Figure A.1: Process flowchart of simulation with faulty action correction.

A.2 Reward Function

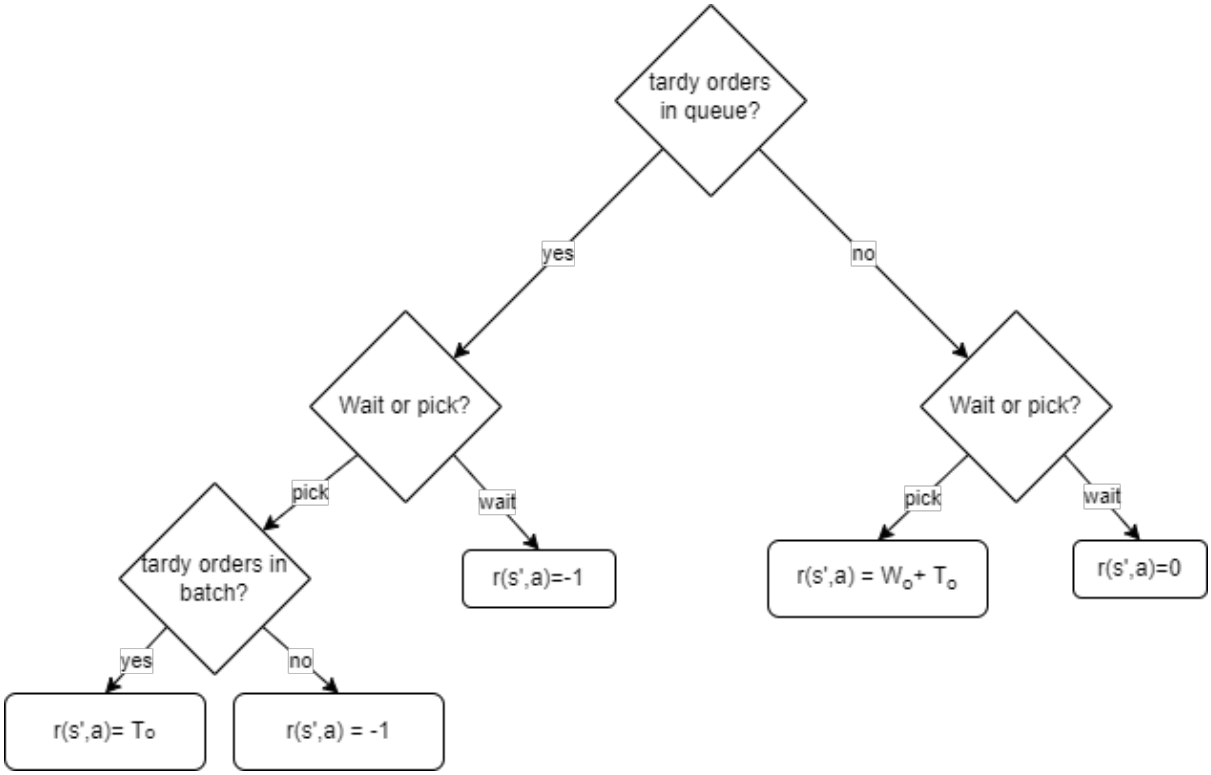
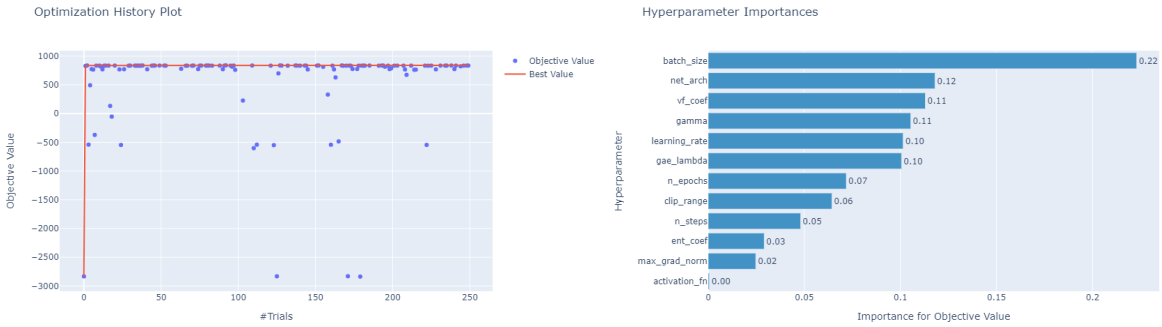


Figure A.2: Flowchart representation of reward function

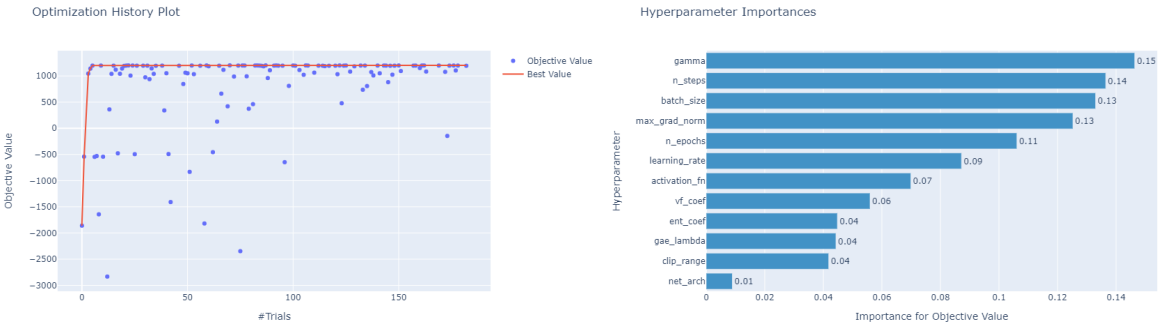
A.3 Hyperparameter Tuning



(a) Optimisation history

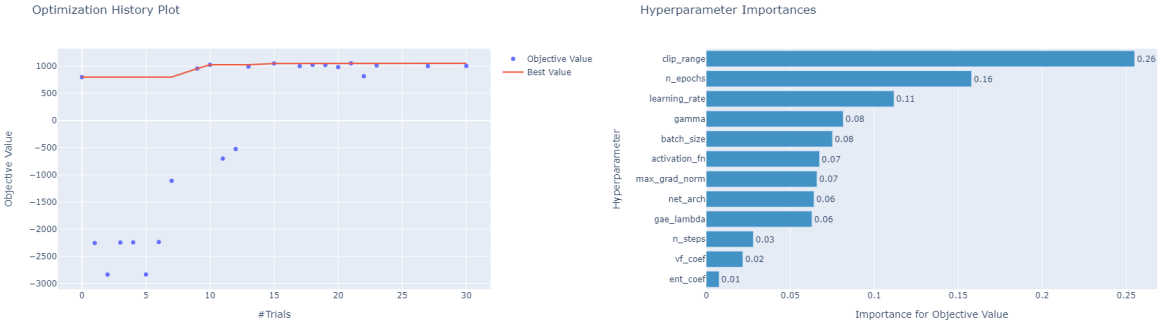
(b) Parameter importance

Figure A.3: Results of hyperparameter tuning of agent 1b



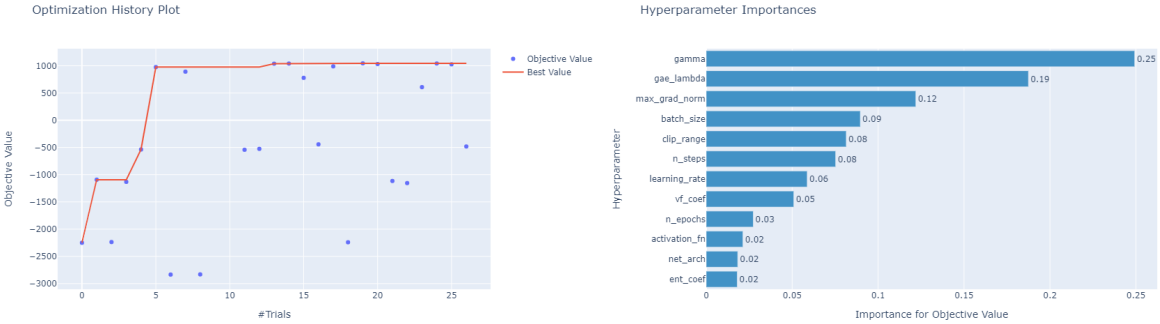
(a) Optimisation history (b) Parameter importance

Figure A.4: Results of hyperparameter tuning of agent 2



(a) Optimisation history (b) Parameter importance

Figure A.5: Results of hyperparameter tuning of agent 3



(a) Optimisation history (b) Parameter importance

Figure A.6: Results of hyperparameter tuning of agent 3b

	agent 1	agent 1b	agent 2	agent 3	agent 3b
batch_size	64	256	512	64	16
n_steps	2048	8	256	64	64
gamma	0.98	0,99	0,98	0,95	0,9
learning_rate	9.814225821531694e-05	0.00027253913720722104	3.6925967541667603e-05	1.4113447761306228e-05	0.00014391182469395268
ent_coef	4.091993456512464e-06	1.8071034555545138e-08	0.008843732660054768	0.0006560231751191768	1.5971147549297592e-07
clip_range	0.3	0,1	0,3	0,1	0,2
n_epochs	5	5	1	20	1
gae_lambda	0.8	0,99	0,98	1,0	0,92
max_grad_norm	0.6	0,33	0,9	0,9	0,9
vf_coef	0.09220012500781494	0.6900207139180505	0.5791862838341625	0.26551629189032355	0.7461883203789238
net_arch	small	small	medium	small	small
activation_fn	tanh	relu	tanh	tanh	relu

Table A.1: Best hyperparameters after tuning

B Experiments

B.1 % In Time Calculation

To calculate the percentage of orders picked in time, we look at a static environment. In a static environment, the robot has a maximum throughput of approximately 100 orders/hour using MCSSOR (μ_{max}). After the simulation has run, we check how many orders are received with one common due date i (δ_i). We denote this number as N_i. This gives us a theoretical latest possible time that the robot needs to start picking without creating any tardy orders (ε_i).

$$\delta_{\mathrm{i}} - \varepsilon_{\mathrm{i}} = \frac{N_i}{\mu_{\mathrm{max}}} \tag{B.1}$$

For example, if 400 orders have arrived with a due date at 16:00h, the robot's latest possible time to start picking is at 12:00h. This would be in a static environment; in a dynamic environment, not all orders would be in by noon, and so possibly less efficient batches can be made. This makes t_{early} a strict upper-bound, and the real value is likely lower. However, since the exact solution to this problem is NP-hard (Nataraja et al., 2022), this simple variant of t_{early} will be used to set the time frame for in time orders. As illustrated in Figure B.1, orders fulfilled before t_{early} are marked as early, orders picked between t_{early} and the due date are labelled as in time, and all orders that are picked after the due date are considered tardy.

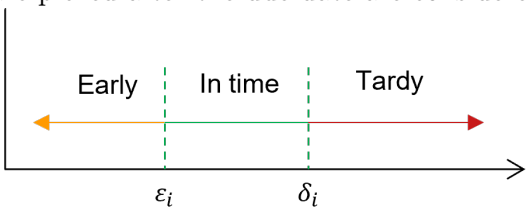


Figure B.1: Labelling of orders as early, in time, or tardy

C Results

C.1 Experiment 1

C.1.1 Low Demand

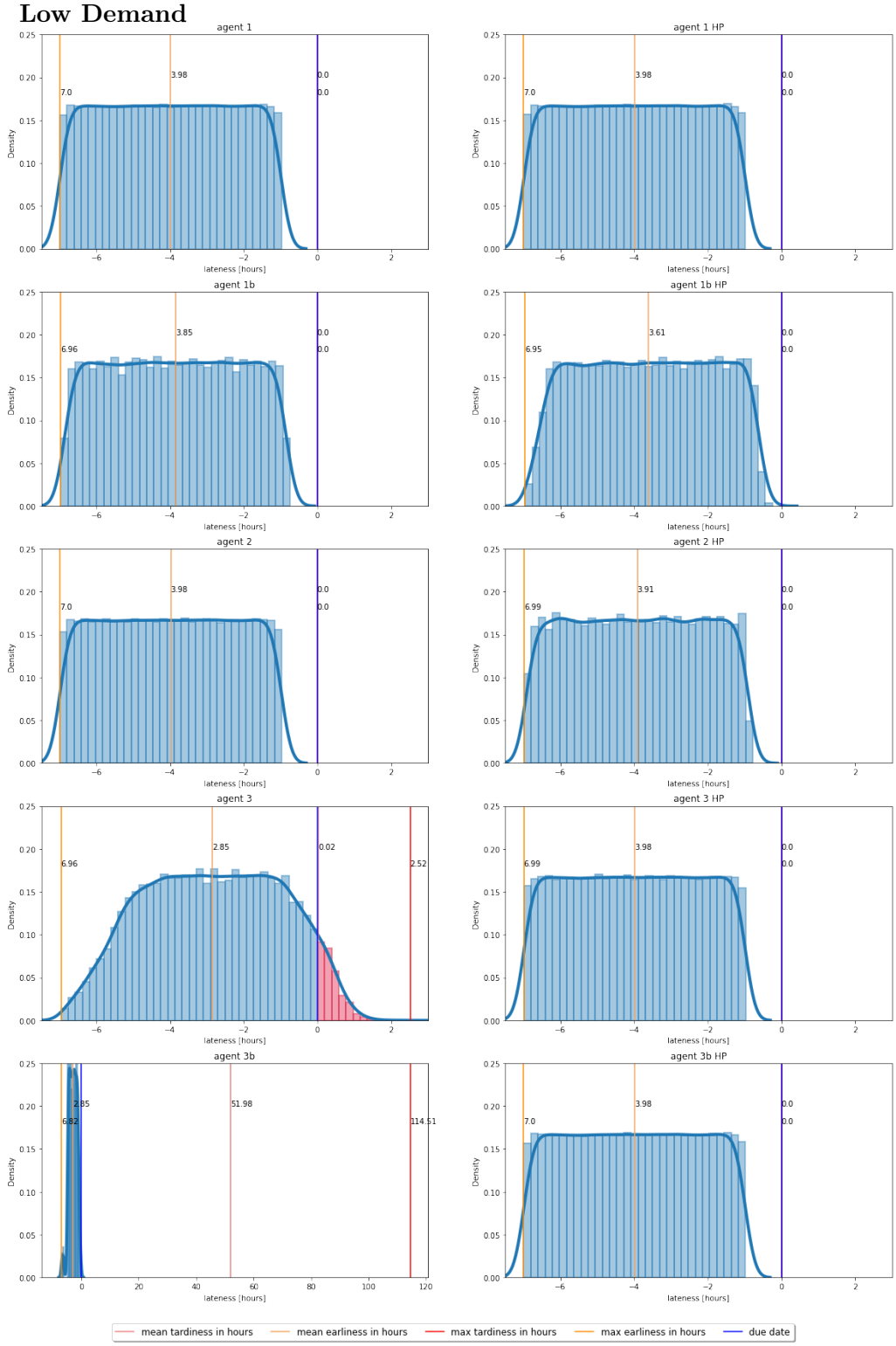


Figure C.1: Distribution of picking completion time of orders along the due-date for agents before and after hyperparameter tuning, in *low* demand.

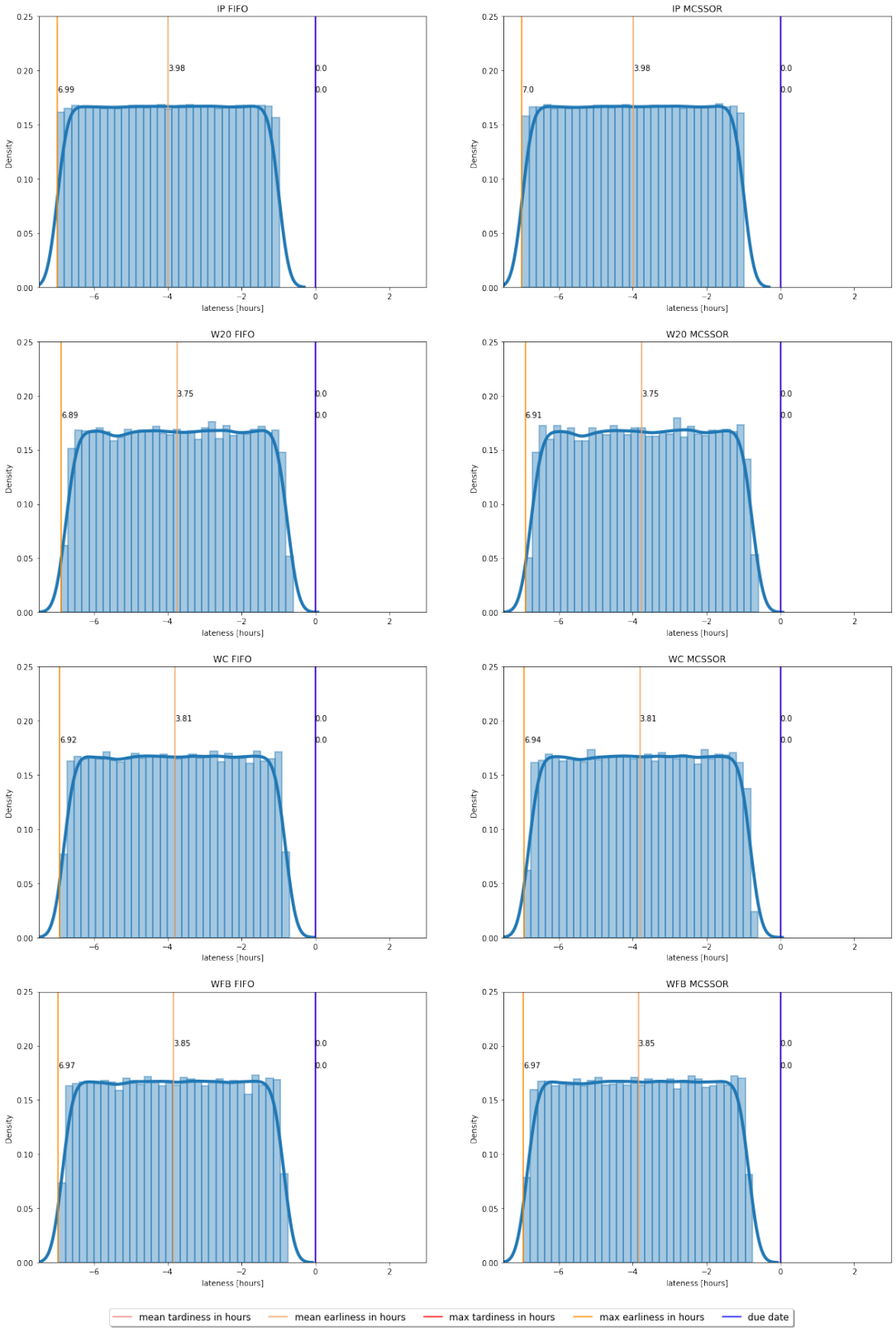


Figure C.2: Distribution of picking completion time of orders along the due-date for heuristics in *low* demand.

C.1.2 Decreased Demand

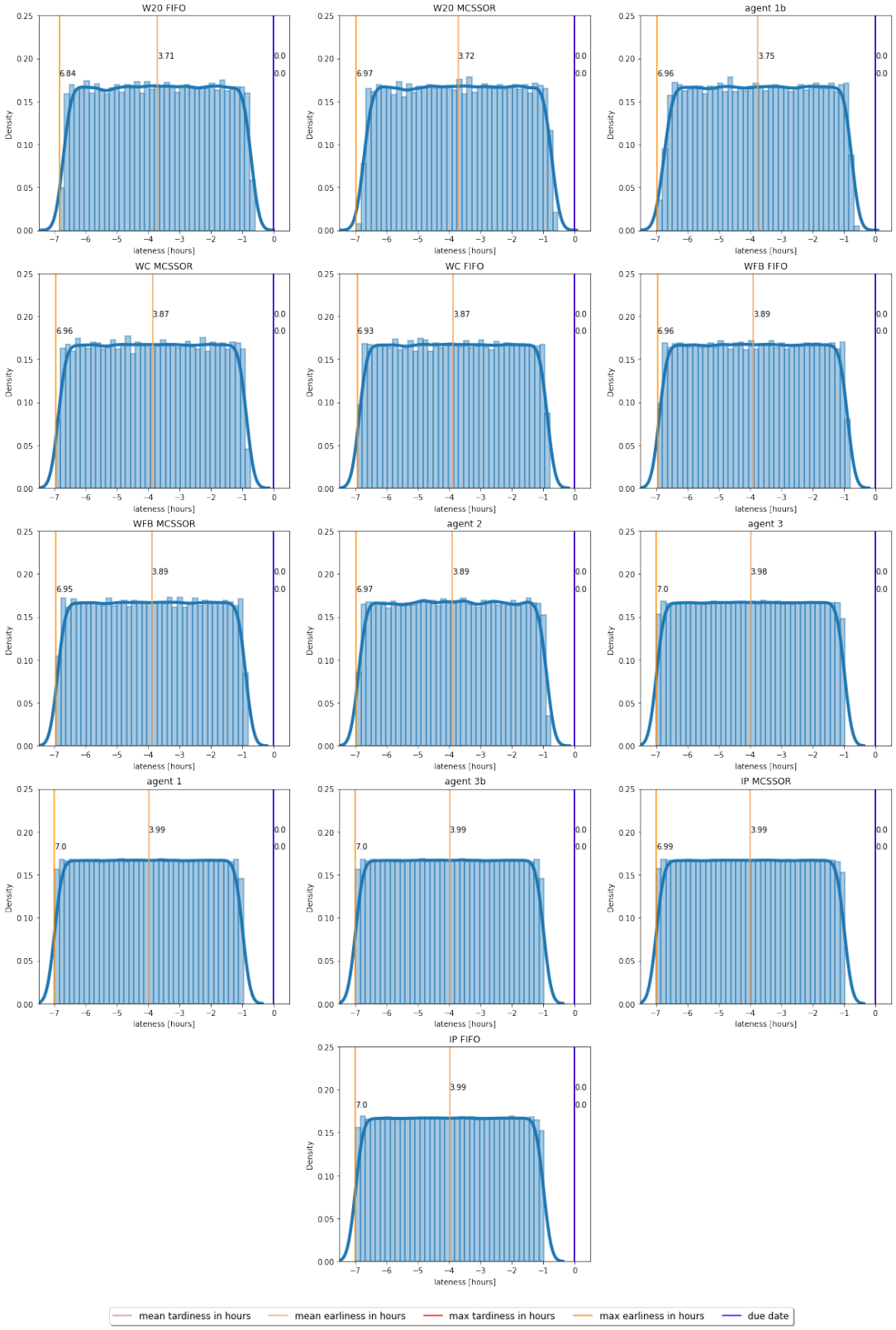


Figure C.3: Distribution of picking completion time of orders along the due-date for all decision-makers in *decreased* demand.

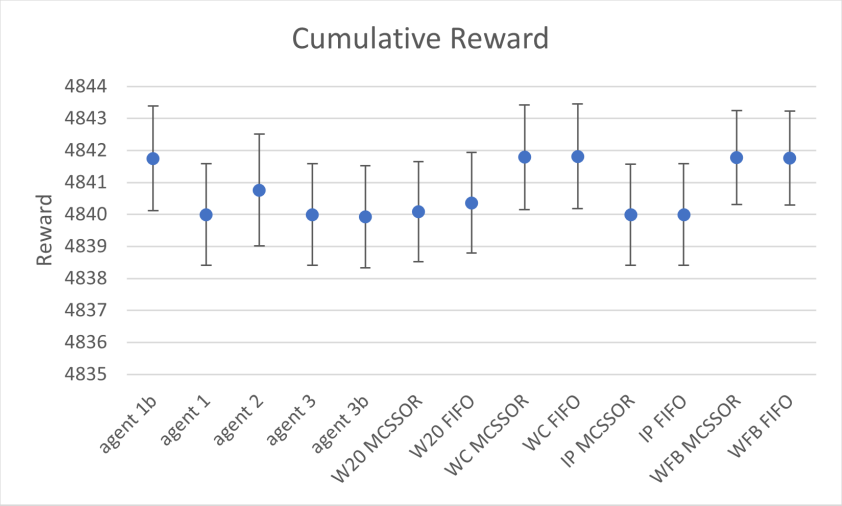


Figure C.4: Mean cumulative reward per decision-maker for *decreased* demand with episodes of 5 days

C.1.3 Normal Demand

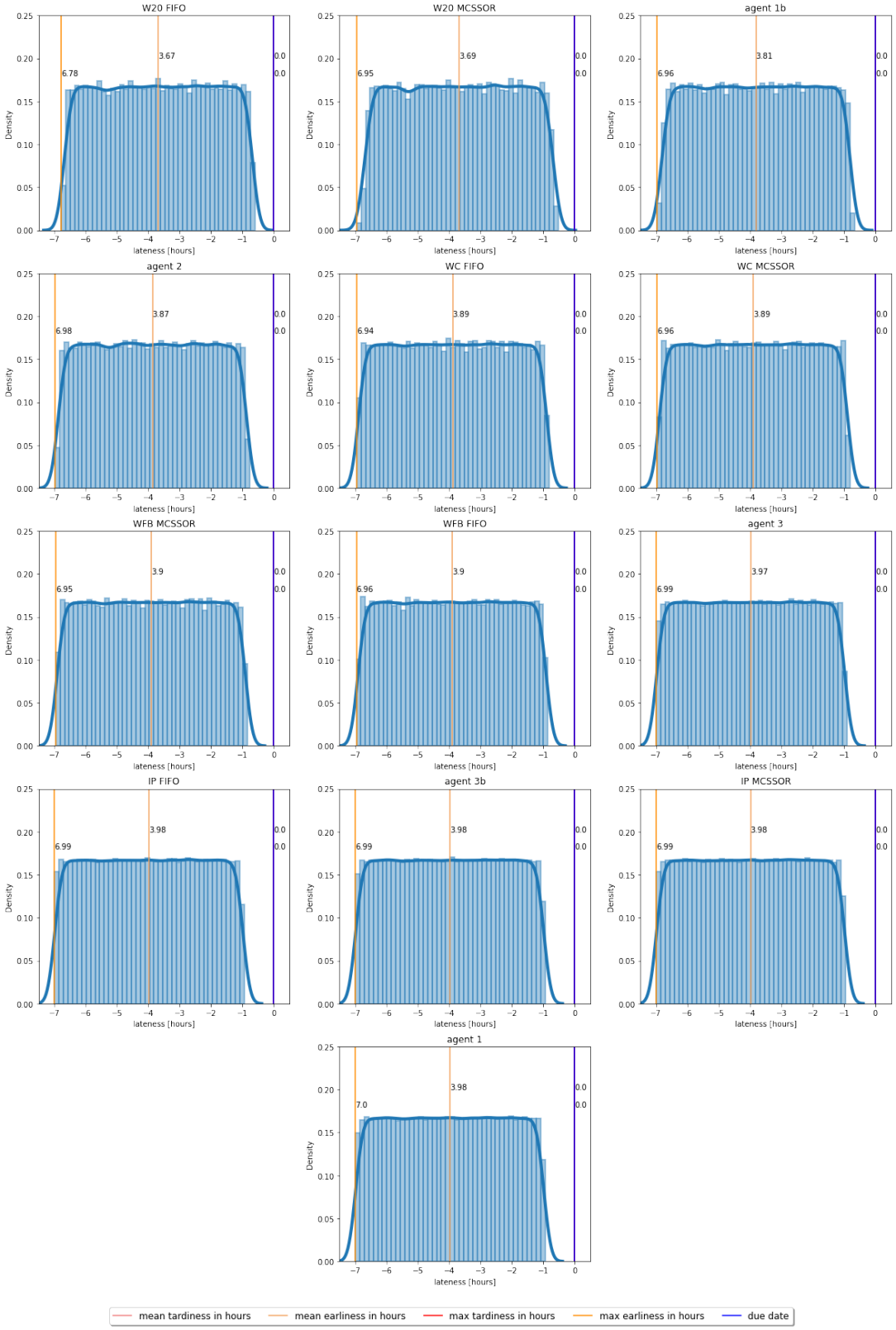


Figure C.5: Distribution of picking completion time of orders along the due-date for all decision-makers in *normal* demand.

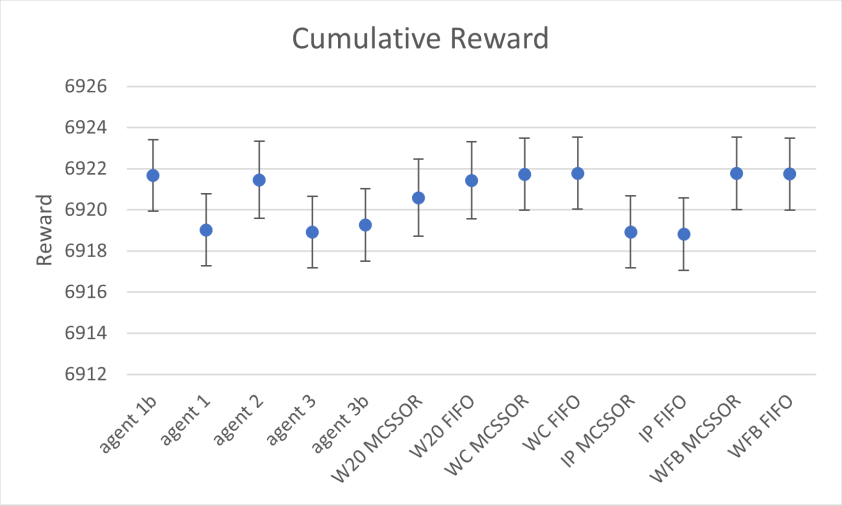


Figure C.6: Mean cumulative reward per decision-maker for *normal* demand with episodes of 5 days

C.1.4 Increased Demand

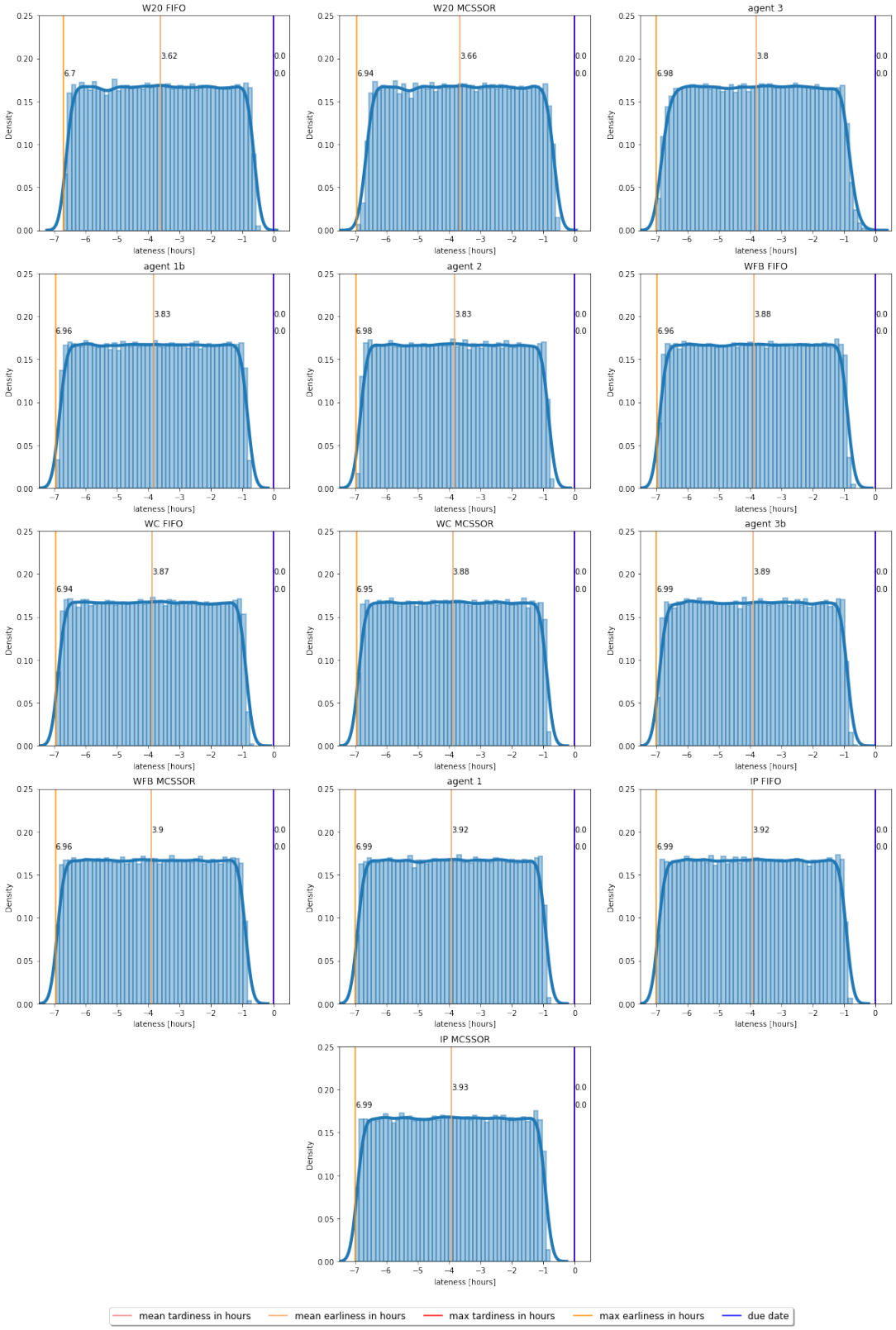


Figure C.7: Distribution of picking completion time of orders along the due-date for all decision-makers in *increased* demand.

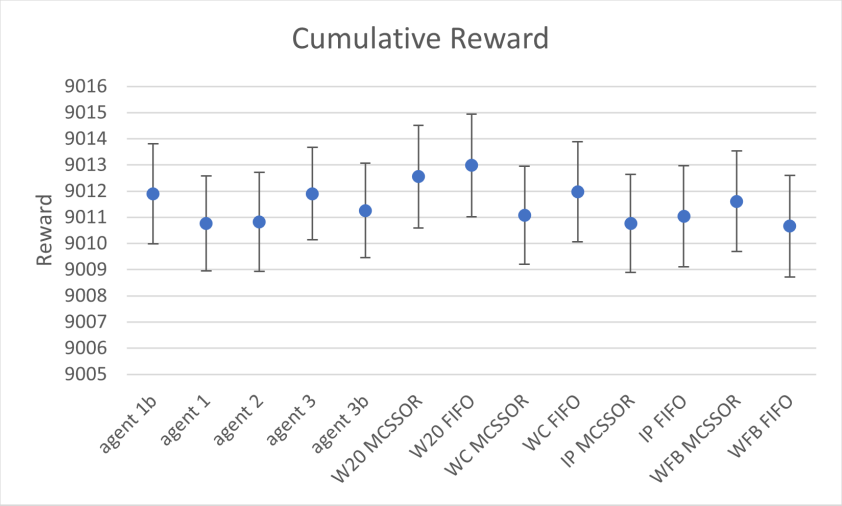


Figure C.8: Mean cumulative reward per decision-maker for *increased* demand with episodes of 5 days

C.1.5 High Demand

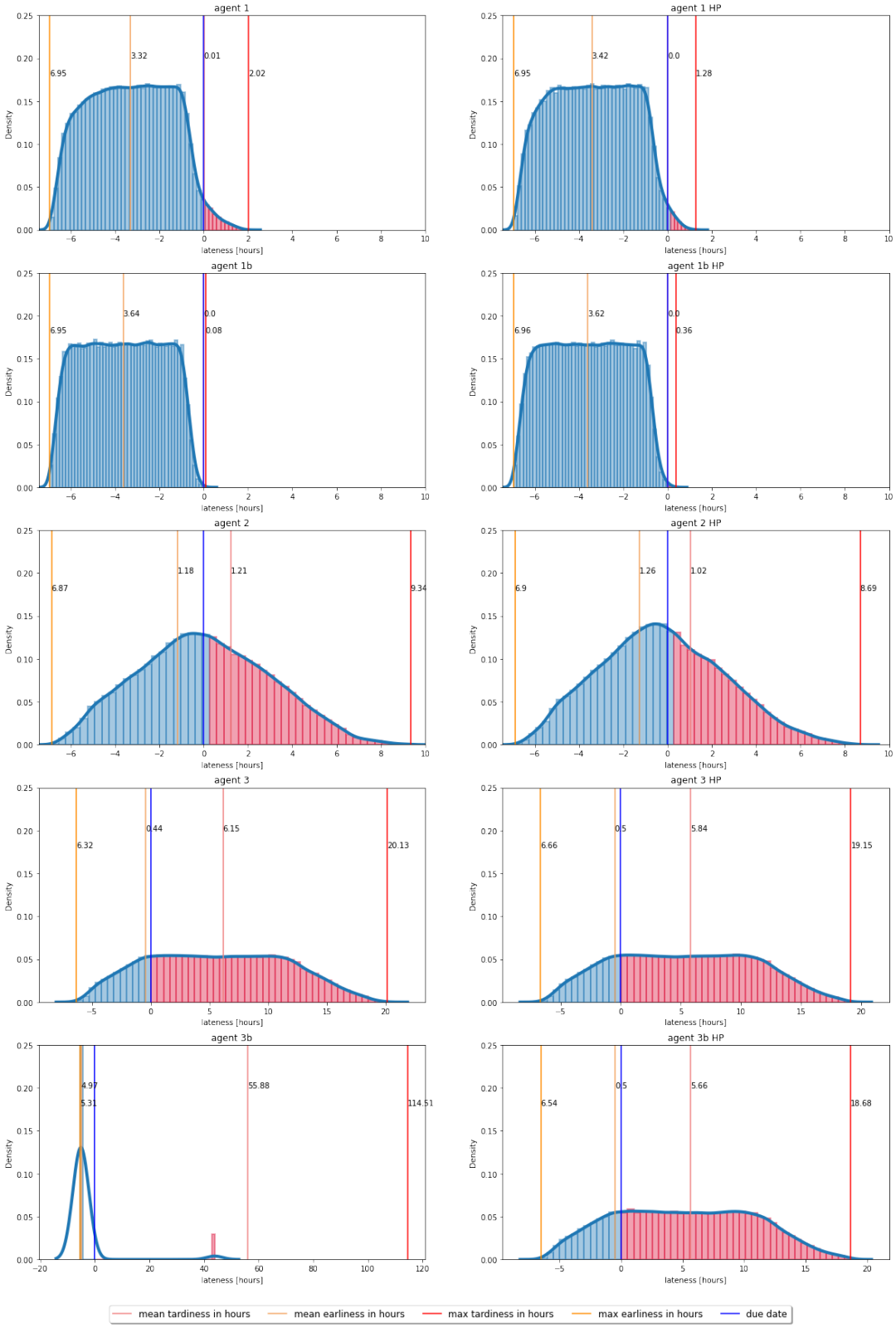


Figure C.9: Distribution of picking completion time of orders along the due-date for agents before and after hyperparameter tuning, in *high* demand.

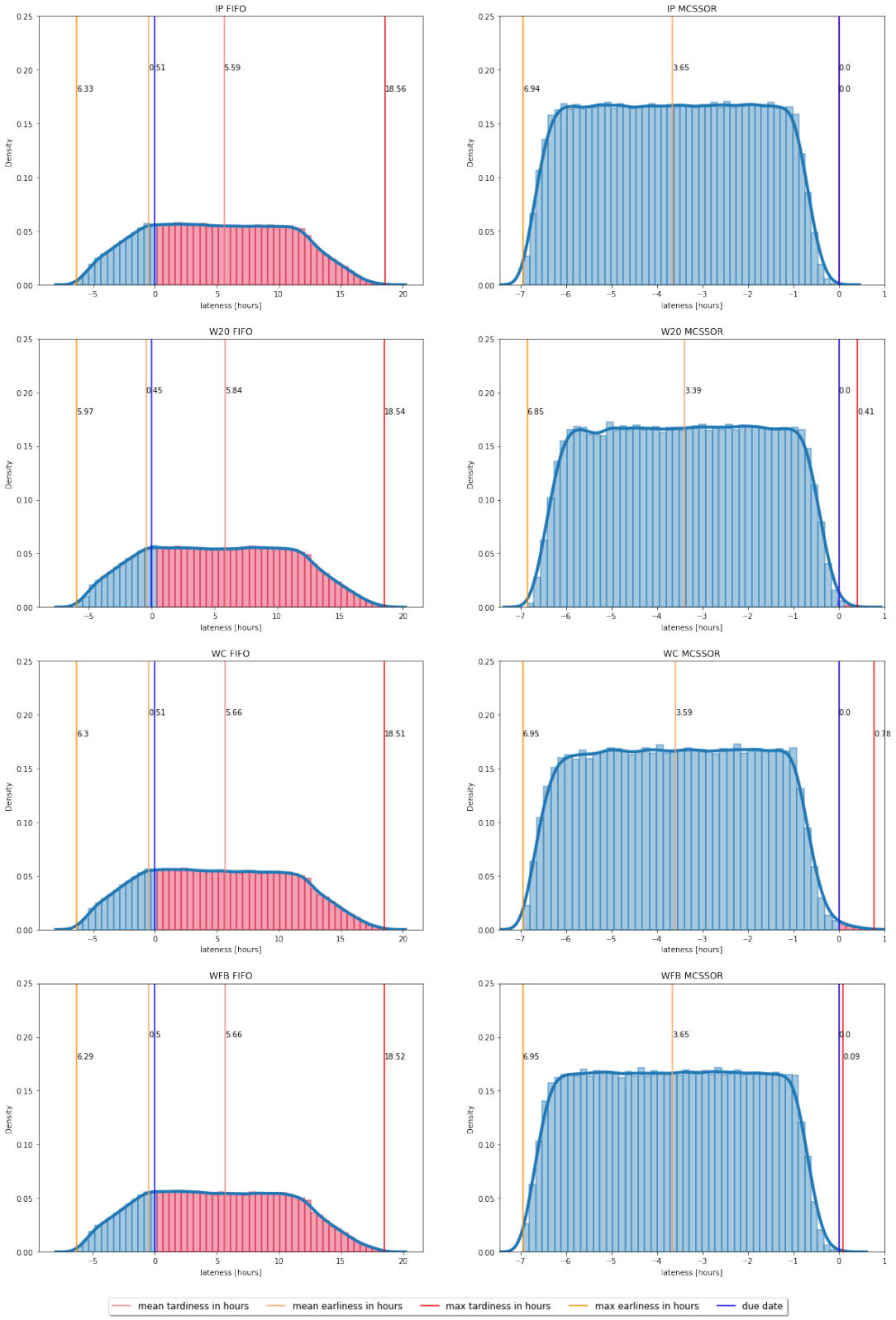


Figure C.10: Distribution of picking completion time of orders along the due-date for heuristics in *high* demand.

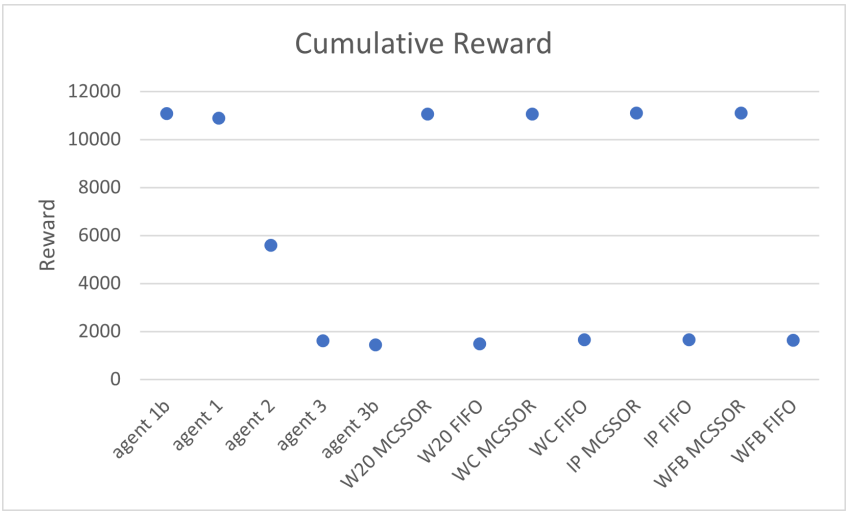


Figure C.11: Mean cumulative reward per decision-maker for *high* demand with episodes of 5 days

C.2 Real Demand

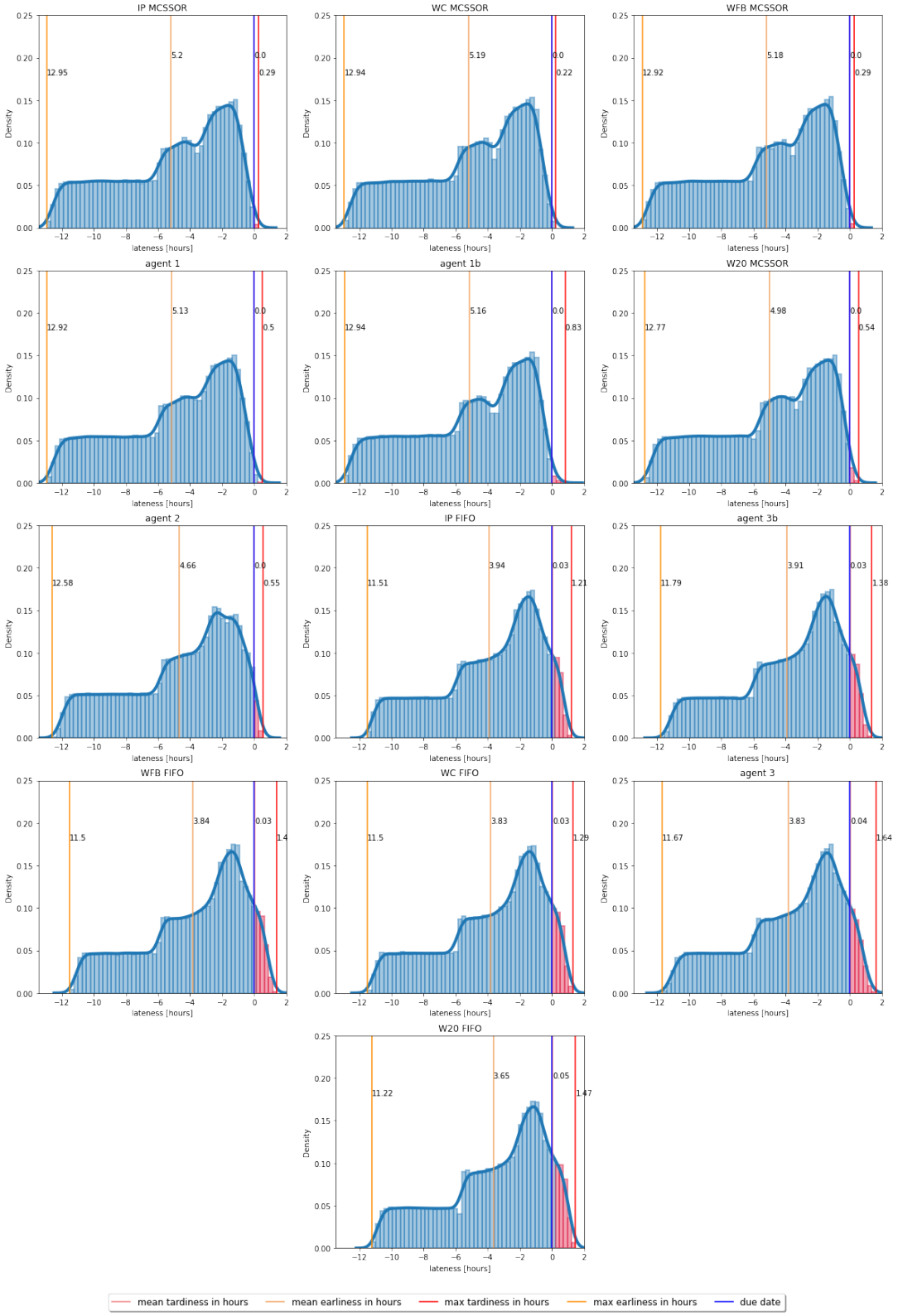


Figure C.12: Distribution of picking completion time of orders along the due-date for all decision-makers in *real* demand.

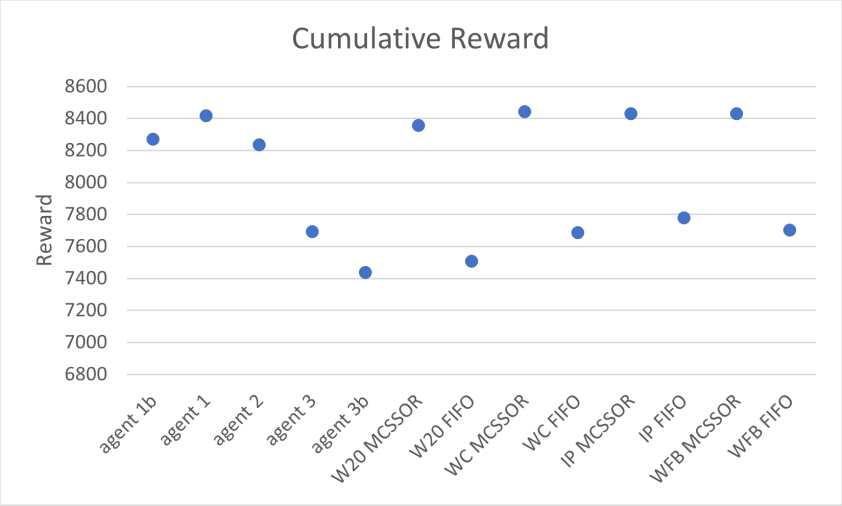
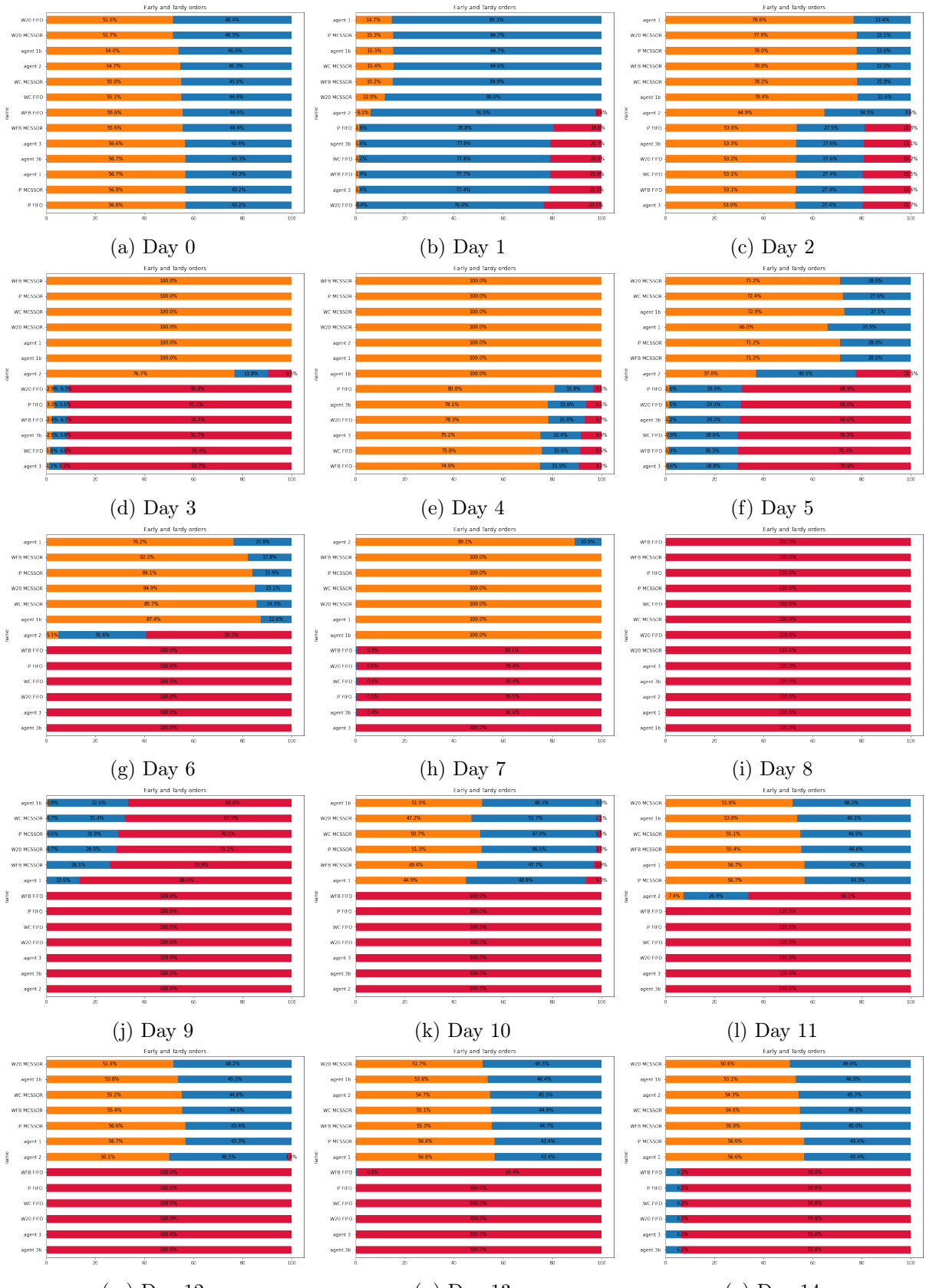


Figure C.13: Mean cumulative reward per decision-maker for *real* demand with episodes of 5 days

C.2.1 Peak Demand

Figure C.14: Distribution of picking completion time of orders along the due-date for peak demand.



(a) Day 0 (b) Day 1 (c) Day 2

(d) Day 3 (e) Day 4 (f) Day 5

(g) Day 6 (h) Day 7 (i) Day 8

(j) Day 9 (k) Day 10 (l) Day 11

(m) Day 12 (n) Day 13 (o) Day 14

Figure C.15: Performance comparison of agents and heuristics under peak demand arrival.

C.3 Full Factorial Analyses

demand	**early/tardy**	**decision**	**mean effect**	**confidence interval**
low	early	A	-0.054	[-0.055, -0.053]
		B	-0.168	[-0.169 ,-0.167]
	tardy	A	0.0	[0.0, 0.0]
		B	0.0	[0.0, 0.0]
decreased	early	A	-0.070	[-0.070, -0.069]
		B	-0.181	[-0.181, -0.180]
	tardy	A	0.0	[0.0, 0.0]
		B	0.0	[0.0, 0.0]
normal	early	A	-0.092	[-0.096, -0.089]
		B	-0.168	[-0.171, -0.165]
	tardy	A	0.0	[0.0, 0.0]
		B	0.0	[0.0, 0.0]
increased	early	A	1,013	[0.912, 1.113]
		B	1,067	[0.955, 1.179]
	tardy	A	-0.327	[-0.414, -0.241]
		B	-0.327	[-0.413, -0.241]
high	early	A	1.445	[1.406, 1.485]
		B	1.657	[1.621, 1.693]
	tardy	A	-7.142	[-7.420, -6.864]
		B	-2.520	[-2.757, -2.284]

Table C.1: Effect of each decision with 95% certainty interval

D Strategy Analyses

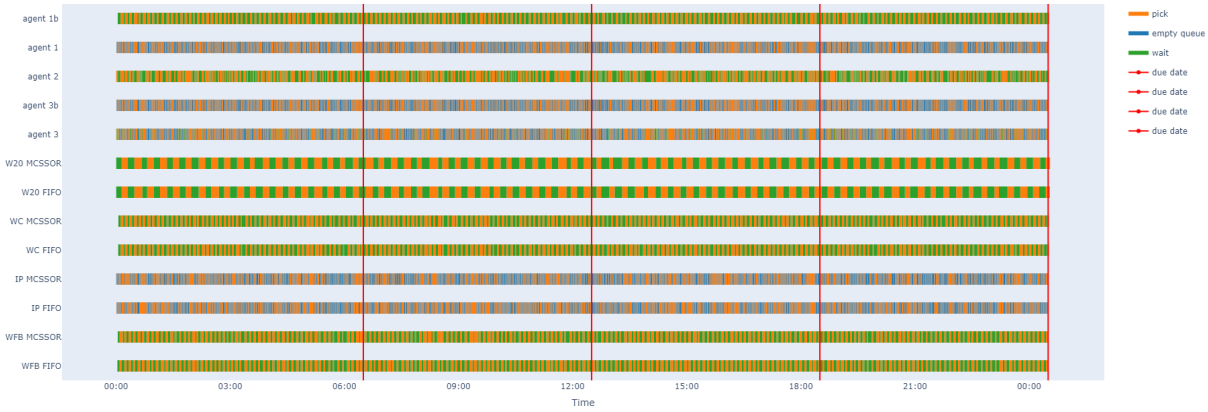


Figure D.1: Robot's activity over time in *decreased* demanding environment

Figure D.2: Robot's activity over time in *increased* demanding environment

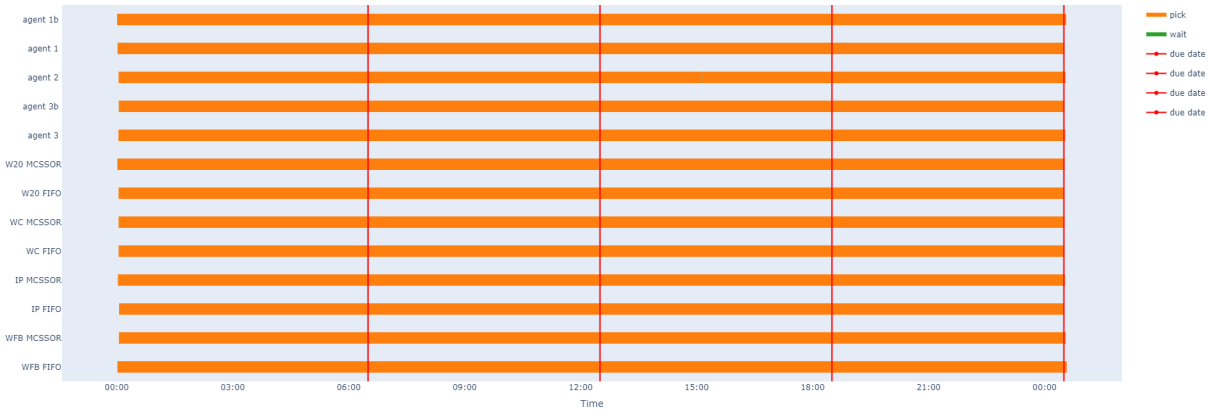
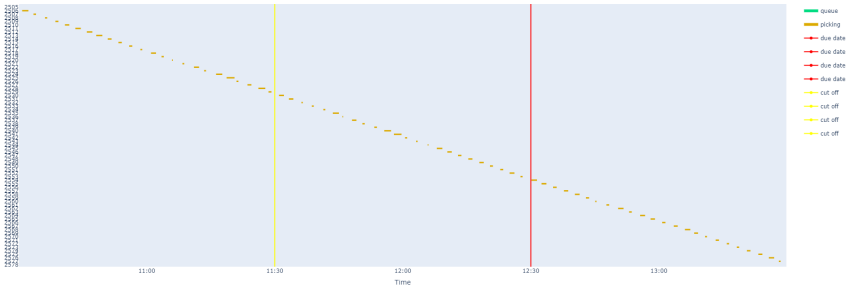
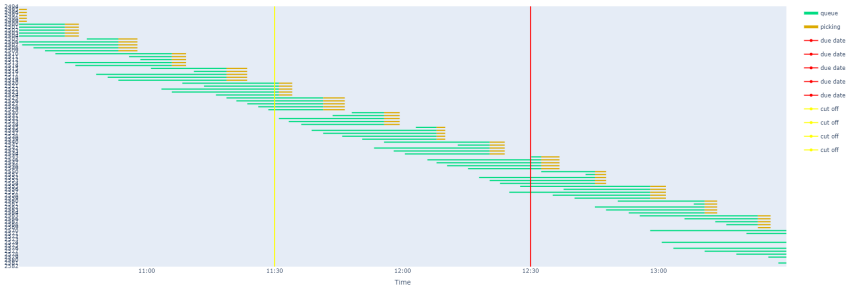


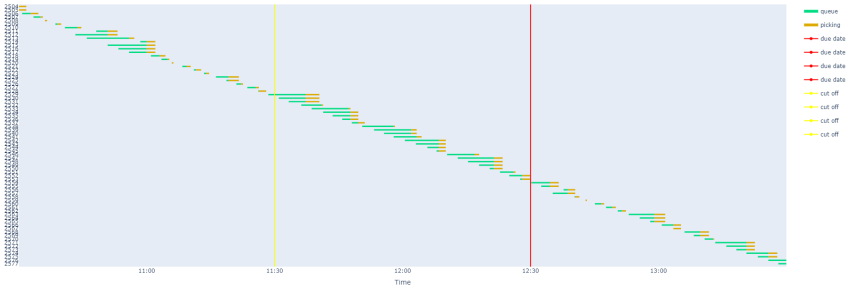
Figure D.3: Robot's activity over time in *high* demanding environment



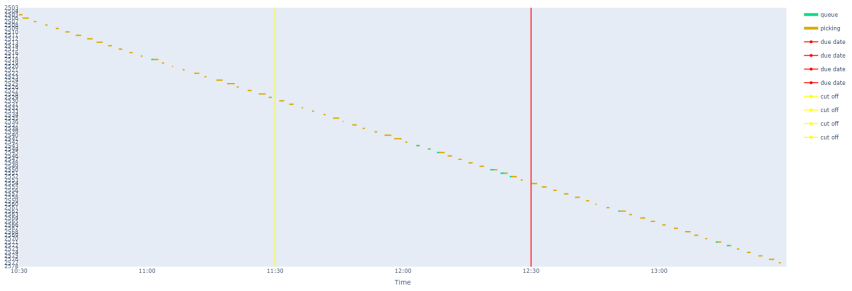
(a) Agent 1



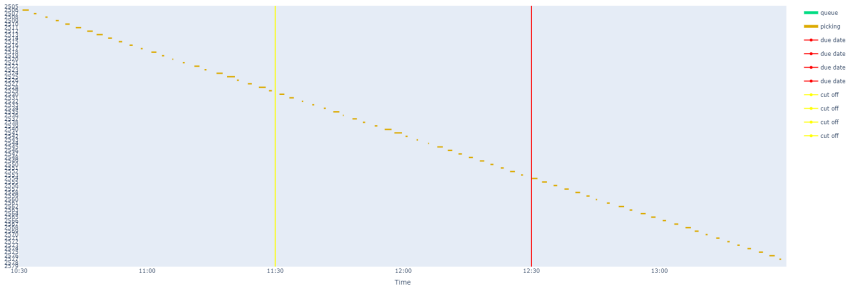
(b) Agent 1b



(c) Agent 2

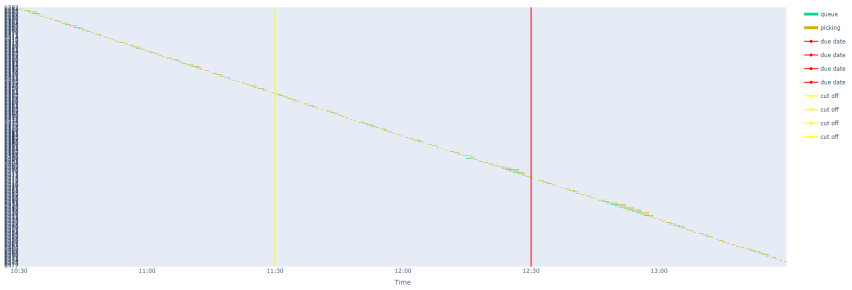


(d) Agent 3

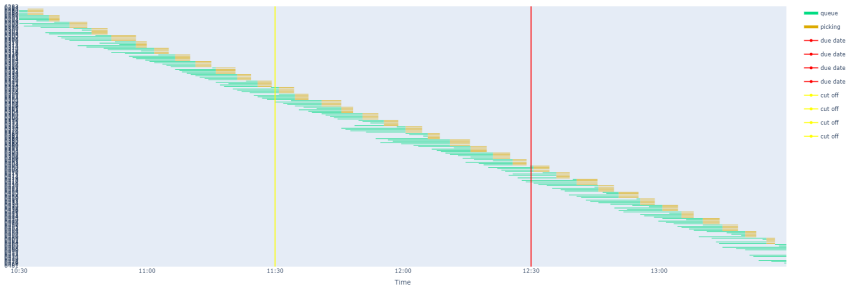


(e) Agent 3b

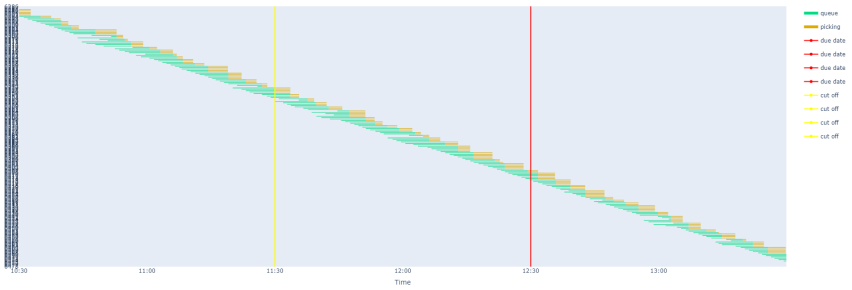
Figure D.4: The arrival and processing of orders over time in *low* demand



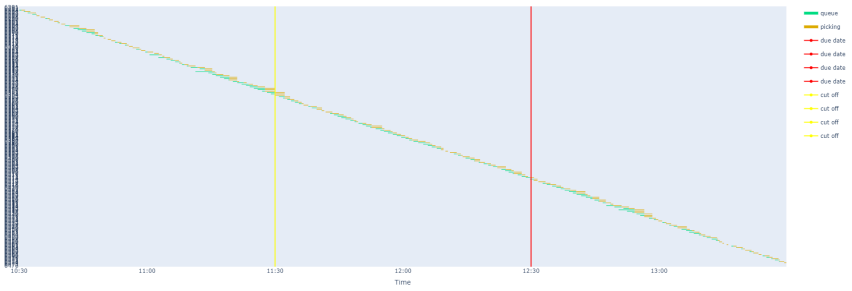
(a) Agent 1



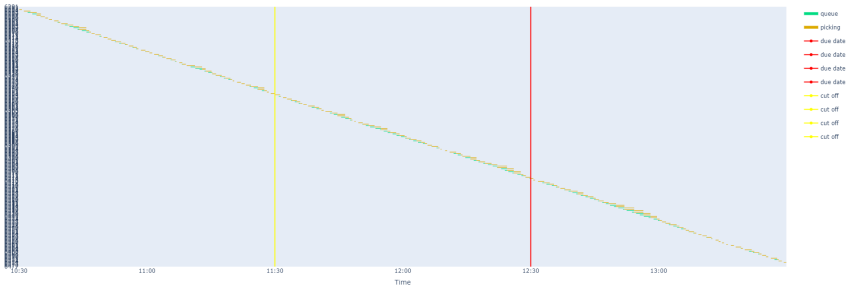
(b) Agent 1b



(c) Agent 2

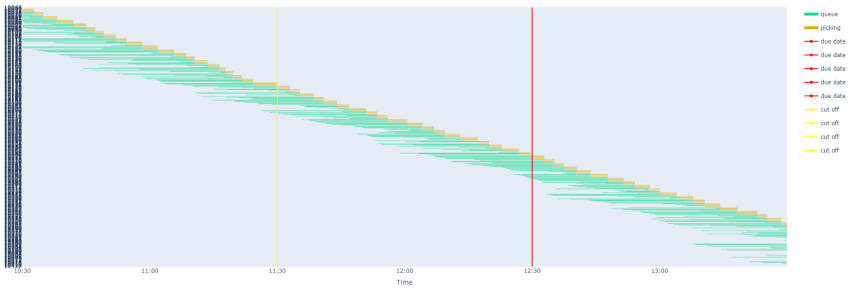


(d) Agent 3

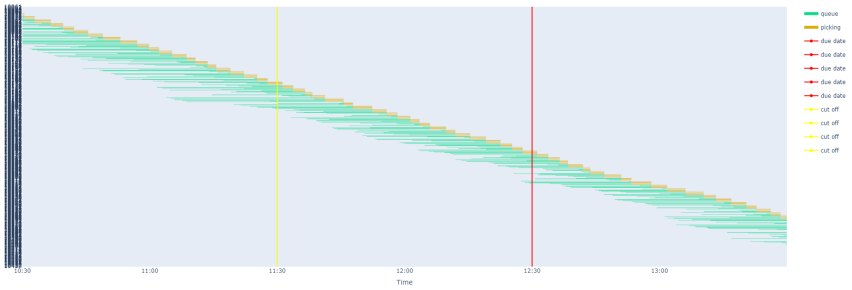


(e) Agent 3b

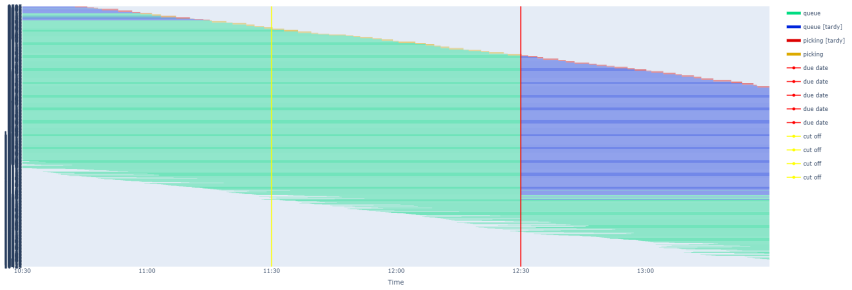
Figure D.5: The arrival and processing of orders over time in *normal* demand



(a) Agent 1

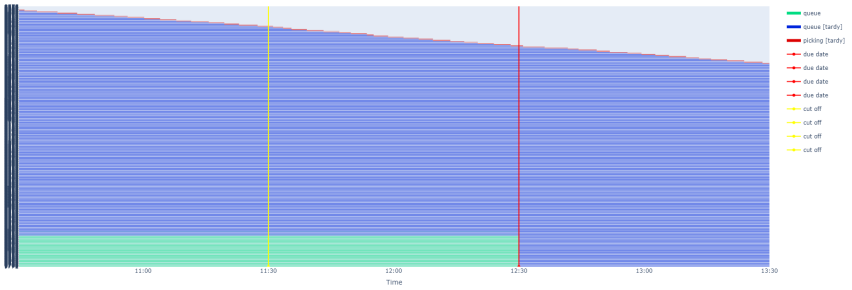


(b) Agent 1b



(c) Agent 2

(d) Agent 3



(e) Agent 3b

Figure D.6: The arrival and processing of orders over time in *high* demand

E Results Overview

Below, we show a summary of most results shown discussed in Section 6.1. Each colour gradient scale is set separately for each demand, with green being the best scoring.

	Low	Decreased	Nomral	Increased	High	Real
agent 1	0.9668	0.9668	0.9668	0.9673	0.9535	0.9522
agent 1b	0.9700	0.9687	0.9683	0.9681	0.9686	0.9343
agent 2	0.9674	0.9676	0.9678	0.9681	0.5192	0.9400
agent 3	0.9668	0.9668	0.9669	0.9683	0.1488	0.8617
agent 3b	0.9668	0.9668	0.9668	0.9676	0.1643	0.8905
IP-FIFO	0.9668	0.9668	0.9668	0.9673	0.1699	0.9002
IP-MCSSOR	0.9668	0.9668	0.9668	0.9673	0.9696	0.9532
WFB-FIFO	0.9679	0.9676	0.9675	0.9677	0.1669	0.8924
WFB-MCSSOR	0.9679	0.9676	0.9675	0.9675	0.9695	0.9531
W20-FIFO	0.9687	0.9691	0.9694	0.9699	0.1526	0.8712
W20-MCSSOR	0.9687	0.9690	0.9693	0.9695	0.9690	0.9463
WC-FIFO	0.9683	0.9678	0.9676	0.9677	0.1695	0.8910
WC-MCSSOR	0.9683	0.9678	0.9676	0.9676	0.9668	0.9534

Figure E.1: Overview of average reward per picked order.

	Low	Decreased	Normal	Increased	High	Real
agent 1	3.983362	3.98627826	3.980364	3.9192705	3.417589	5.1319304
agent 1b	3.605447	3.751496241	3.807703	3.8298425	3.618308	5.1574911
agent 2	3.906388	3.891431329	3.866496	3.8299374	1.264512	4.6581048
agent 3	3.981408	3.983363894	3.974712	3.7994147	0.499298	3.8297549
agent 3b	3.983365	3.986281806	3.980465	3.8884105	0.502741	3.9059666
IP-FIFO	3.983403	3.986259547	3.980457	3.9219172	0.509055	3.9359277
IP-MCSSOR	3.983335	3.986356923	3.980417	3.9267964	3.653196	5.1965812
WFB-FIFO	3.849848	3.890819312	3.904028	3.8769522	0.496428	3.8429895
WFB-MCSSOR	3.849781	3.890126759	3.903568	3.8959092	3.654777	5.1771161
W20-FIFO	3.750214	3.712702389	3.673391	3.6158034	0.447549	3.6540688
W20-MCSSOR	3.751575	3.719404643	3.685983	3.6576763	3.394305	4.9812028
WC-FIFO	3.807963	3.866121841	3.88717	3.8725961	0.508868	3.8315369
WC-MCSSOR	3.80807	3.86677509	3.887881	3.884171	3.59396	5.1875278

Figure E.2: Overview of mean earliness per order.

	Low	Decreased	Normal	Increased	High	Real
agent 1	0	0	0	0	0.003821	0.0002123
agent 1b	0	0	0	0	6.30E-05	0.0006185
agent 2	0	0	0	0	1.022903	0.0022342
agent 3	0	0	0	0	5.83697	0.0363537
agent 3b	0	0	0	0	5.659314	0.0287511
IP-FIFO	0	0	0	0	5.585107	0.0258666
IP-MCSSOR	0	0	0	0	0	0.000116
WFB-FIFO	0	0	0	0	5.656683	0.032579
WFB-MCSSOR	0	0	0	0	4.35E-06	0.0001444
W20-FIFO	0	0	0	0	5.842363	0.051342
W20-MCSSOR	0	0	0	0	0.000101	0.000752
WC-FIFO	0	0	0	0	5.65682	0.0334371
WC-MCSSOR	0	0	0	0	0.000489	0.0001012

Figure E.3: Overview of mean tardiness per order.

	Low	Decreased	Normal	Increased	High	Real
agent 1	11	11	13	11	6	4
agent 1b	1	3	3	4	3	5
agent 2	8	8	4	5	7	7
agent 3	9	9	9	3	12	12
agent 3b	10	10	11	9	10	9
IP-FIFO	13	12	10	12	8	8
IP-MCSSOR	12	11	12	13	1	1
WFB-FIFO	6	6	8	6	11	10
WFB-MCSSOR	7	7	7	10	2	3
W20-FIFO	2	1	1	1	13	13
W20-MCSSOR	3	2	2	2	4	6
WC-FIFO	4	5	5	7	9	11
WC-MCSSOR	5	4	6	8	5	2

Figure E.4: Overview of rating based on their order in the performance comparison plots.