FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-time dispatching in manufacturing using machine-learning models

Maria Luísa Fontanete Loureiro Salvador

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Jorge Manuel Gomes Barbosa

Second Supervisor: Nuno Fernandes - Critical Manufacturing

July 29, 2026

Real-time dispatching in manufacturing using machine-learning models

Maria Luísa Fontanete Loureiro Salvador

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Gil Gonçalves
Examiner: Prof. Nuno Lau

July 29, 2026

Resumo

O escalonamento em tempo real numa unidade de fabrico de semicondutores é um problema de elevada complexidade: cada lote percorre centenas de passos reentrantes, sujeito a tempos de *setup* dependentes da sequência, qualificação restrita de equipamentos e perturbações estocásticas como avarias e manutenções. Na prática, estas decisões são tomadas por regras de prioridade estáticas que, embora eficientes e interpretáveis, não se adaptam ao estado corrente da fábrica, e nenhuma regra domina em todas as condições de operação.

Esta dissertação propõe um *meta-dispatcher* baseado em aprendizagem por reforço que, a cada evento de despacho, seleciona dinamicamente qual das regras clássicas (FIFO, SPT, EDD, LST, CR, ATCS) o sistema de execução deve aplicar, operando como uma camada acima da interface de despacho do MES em vez de a substituir. O agente é treinado com *Proximal Policy Optimization* sobre um ambiente de simulação de eventos discretos construído a partir de cinco meses de registos operacionais reais de uma fábrica (622 máquinas, 281 fluxos de produto), com um espaço de estado de 35 dimensões e uma função de recompensa multi-termo com modelação de arrependimento.

Avaliado contra cinco regras estáticas em dados de teste retidos, e embora a diferença por episódio não seja estatisticamente significativa devido à elevada assimetria das distribuições, o agente atinge consistentemente a menor tardiness média de todas as políticas (214,2 h, uma redução de 25,7% face ao melhor baseline), mantendo uma taxa de entrega a tempo competitiva. De forma mais relevante, as preferências de regra aprendidas são consistentes, em termos direcionais, com a teoria clássica de escalonamento: o agente recorre à única regra do seu repertório sensível a custos de *setup* precisamente nos cenários de maior pressão de *setup*. Um estudo complementar com quatro réplicas do agente treinadas de forma independente revela ainda *sensibilidade da escolha de regra à condição de avaliação*: sob condições de dados reais congestionadas, a maioria das réplicas converge para a mesma regra (FIFO ou CR), ao passo que, sob condições pouco congestionadas, cada réplica diverge para uma regra diferente. O contributo central é a validação credível e interpretável de despacho por aprendizagem por reforço sobre dados industriais reais à escala de fábrica.

Palavras-chave: Aprendizagem por Reforço, Despacho em Tempo Real, Fabrico de Semicondutores, Regras de Prioridade, Proximal Policy Optimization, Sistemas de Execução de Fabrico.

Abstract

Real-time dispatching in a semiconductor fabrication facility is a hard decision problem: each lot traverses hundreds of reentrant process steps subject to sequence-dependent setup times, sparse equipment qualification, and stochastic disturbances such as breakdowns and maintenance. In practice these decisions are made by static priority rules which, although efficient and interpretable, do not adapt to the current fab state, and no single rule dominates across all operating conditions.

This dissertation proposes a reinforcement learning *meta-dispatcher* that, at each dispatching event, dynamically selects which of the classical rules (FIFO, SPT, EDD, LST, CR, ATCS) the execution system should apply, acting as a layer above the MES dispatching interface rather than replacing it. The agent is trained with Proximal Policy Optimization on a discrete-event simulation environment built from five months of real operational records of a fab (622 machines, 281 product flows), using a 35-dimensional state space and a multi-term reward with bounded regret shaping.

Evaluated against five static baselines on held-out test data, and although the per-episode difference is not statistically significant owing to the heavily skewed tardiness distributions, the agent consistently achieves the lowest average tardiness of any policy (214.2 h, a 25.7% reduction over the best baseline) while maintaining a competitive on-time delivery rate. More importantly, the learned rule preferences are directionally consistent with classical scheduling theory: the agent relies on the only setup-aware rule in its repertoire specifically under the most setup-pressured conditions. A complementary study across four independently-trained replicas of the agent further reveals *evaluation-condition-dependent rule sensitivity*: under congested real data conditions most replicas converge on the same rule (FIFO or CR), whereas under uncongested conditions each replica diverges to a different one. The central contribution is a credible and interpretable validation of reinforcement-learning dispatching on real, fab-scale industrial data.

Keywords: Reinforcement Learning, Real-Time Dispatching, Semiconductor Manufacturing, Priority Dispatching Rules, Proximal Policy Optimization, Manufacturing Execution Systems.

Acknowledgments

Nunca é tarde demais para agradecer a todos aqueles que, de alguma forma, contribuíram para a realização deste trabalho. Em primeiro lugar, gostaria de expressar a minha sincera gratidão ao meu orientador, Prof. Jorge Barbosa, pela orientação e feedback prestados ao longo deste percurso. Quero também agradecer ao meu co-orientador, Nuno Fernandes, cuja paciência, disponibilidade e conhecimento foram indispensáveis para o sucesso deste projeto.

Aos meus pais, agradeço do fundo do coração por estarem presentes, mesmo longe, e por nunca desistirem de mim. O vosso apoio incondicional, nos bons e maus momentos, foi fundamental para que eu chegasse ao fim deste percurso. Levo comigo todos os bons conselhos e procuro aplicá-los com a responsabilidade e a dedicação que vos vi ter estes anos todos. A vitória é nossa! Agradeço também à minha irmã, que sempre me apoiou e me incentivou a seguir em frente, mesmo quando as coisas pareciam difíceis.

À minha Tia Irene e aos meus avós, obrigada pelas refeições feitas com amor, pelas orações e por acreditarem sempre em mim, mesmo quando eu duvidava.

Aos meus amigos e colegas de curso, que tornaram esta jornada muito mais divertida, Bruna, Choca, Malva, Phranklin, Afonso, Ávila, Macedo, Vasquinho, Gonçalo e Dinis, sou grata por todas as memórias que criámos juntos, entre projetos intermináveis e boas risadas, e por termos aprendido a ultrapassar os desafios e a celebrar as pequenas conquistas em conjunto. Um agradecimento também aos amigos de Viseu que fiz no Porto , por tantos momentos felizes.

À Xica, um abraço apertado e especial. Partilhámos viagens, saídas e conversas sem fim, mas o que ficou mesmo foi a confiança e a proximidade que construímos ao longo dos anos, algo que sei que vou levar para sempre. Obrigada por estares sempre presente, por me ouvires, e me celebrares com tanto carinho. A tua amizade foi um grande presente e espero que continue a ser assim por muitos anos.

Ao Diogo, por ter transformado a nossa casa num lugar seguro onde podemos ser nós mesmos, foi uma verdadeira aventura!

Às minhas amigas de Viseu, last but not least, Maria, Mariana, Ana e Matilde. Obrigada por serem a minha lufada de ar fresco e ao mesmo tempo, o ombro incansável de que precisei sempre que me senti mais em baixo. Ensinaram-me que a vida vai muito além disto. E que, afinal, está só a começar!

"What makes the desert beautiful
is that somewhere it hides a well."

Antoine de Saint-Exupéry, *The Little Prince*

Contents

List of Figures

List of Tables

Abbreviations and Symbols

Reinforcement learning

A3C	Asynchronous Advantage Actor-Critic
CNN	Convolutional Neural Network
DQN	Deep Q-Network
GAE	Generalized Advantage Estimation
GNN	Graph Neural Network
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
TRPO	Trust Region Policy Optimization

Scheduling and dispatching

ATCS	Apparent Tardiness Cost with Setups
CR	Critical Ratio
DES	Discrete-Event Simulation
EDD	Earliest Due Date
FIFO	First In First Out
FJSP	Flexible Job Shop Problem
JSSP	Job Shop Scheduling Problem
LST	Least Slack Time
QTC	Queue-Time Constraint
SPT	Shortest Processing Time

Manufacturing and metrics

CMP	Chemical-Mechanical Polishing
CoV	Coefficient of Variation
FOUP	Front Opening Unified Pod
KPI	Key Performance Indicator
MES	Manufacturing Execution System
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
OTR	On-Time Rate
TSMC	Taiwan Semiconductor Manufacturing Company
WIP	Work-in-Process

General

PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
SDG	Sustainable Development Goal

Chapter 1

Introduction

1.1 Context

Over the past decades, semiconductor technology has become a central enabler of modern digital systems, supporting the widespread adoption of computers, smartphones, sensors, data centers, automotive systems and, more recently, artificial intelligence applications. As a result, semiconductor devices are now embedded in many sectors of the global economy and everyday life, making semiconductor manufacturing a strategically important industrial domain worldwide. This market continues to expand rapidly, with global sales increasing from 630.5$ billion in 2024 to 791.7$ billion in 2025, representing an annual growth rate of 25.6%. This growth is largely driven by the increasing demand for AI-related technologies, which have become a major source of revenue for the industry [**?**].

Sustaining this growth trajectory requires semiconductor manufacturers to continuously expand production capacity, which in turn demands massive and ongoing capital investment. Modern fabrication facilities rely on highly specialized equipment that is both technically sophisticated and prohibitively expensive to acquire or replace. TSMC alone reported capital expenditures of 40.9 billion in 2025, with planned investments ranging from 52 billion to 56 billion for 2026 [**?**], while individual advanced lithography systems can exceed 150 million per unit [**?**]. At this scale of investment, sustained equipment productivity is not an operational preference but an economic imperative.

This imperative gives rise to a fundamental challenge in manufacturing management: how to effectively coordinate the flow of work through the facility such that equipment utilization is maximized, order completion times are minimized, and resources are allocated without waste. As highlighted by Wiendahl et al. [**?**], these objectives are inherently interrelated and frequently in conflict, such that improving performance along one dimension may come at the expense of another. Navigating these trade-offs systematically, under demanding production targets and continuously evolving shop-floor conditions, requires structured and informed decision-making throughout the production process, a function fulfilled, in practice, by production scheduling and dispatching.

The growing need to manage these challenges coincided with the emergence of Industry 4.0,

a broad transformation of manufacturing driven by the integration of digital technologies that enhance efficiency, adaptability, and data-driven decision-making. One of the most significant enablers of this shift is the Manufacturing Execution System (MES), which has become a key component of modern production environments. Acting as the central operational layer of the facility, these platforms are designed to improve productivity, monitoring, and synchronization of production resources while providing real-time visibility into shop-floor operations [**?**, **?**]. Within such systems, routing and scheduling modules play a particularly vital role, as they define the dispatching rules that guide how materials are allocated to equipment across the different stages of the production process. In practice, operators can configure process types, optimization targets, and scheduling rules according to their production goals, while the system itself accounts for processing time variability across machines and supports the simultaneous processing of products on multiple pieces of equipment. Beyond execution, these platforms also enable the simulation and evaluation of alternative scheduling and dispatching strategies under dynamic shop-floor conditions, allowing manufacturers to assess their impact on overall production performance [**?**].

1.2 Motivation

While MES platforms provide the infrastructure to configure and execute dispatching rules, the effectiveness of those rules depends heavily on the complexity of the production environment they operate in. In semiconductor facilities, a single production lot may traverse more than seven hundred interdependent process steps throughout its manufacturing route, often revisiting the same equipment groups multiple times across different processing stages [**?**]. Each dispatching decision is therefore shaped by a complex set of interdependencies involving reentrant flows, sequence-dependent setups, equipment qualification constraints, and strict timing requirements between consecutive steps [**?**]. To manage this complexity in real time, production decisions are driven by priority dispatching rules (DRs), which determine the sequencing and assignment of competing lots to available equipment.

DRs remain widely adopted in industrial practice due to their computational efficiency, interpretability, and straightforward integration with MES dispatching interfaces [**?**, **?**]. Common examples include Shortest Processing Time (SPT), which prioritizes lots with the lowest remaining processing time, Earliest Due Date (EDD), which sequences lots by their delivery deadline, and First In First Out (FIFO), which processes lots in the order they arrive. These platforms support highly configurable production environments where predefined DRs offer a practical way to incorporate operational priorities into real-time scheduling decisions, without requiring a complex optimization model at each decision point.

However, static DRs are inherently context-insensitive once the decision logic is fixed regardless of the current system state. When demand fluctuates, processing and setup times vary, machines fail, or other disruptions occur, their performance often deteriorates, leading to increased waiting times, higher work-in-process levels, and missed due dates. In one such setting, Gui et al. report that this rigidity can result in productivity losses exceeding 15%, mainly due to inefficient

resource allocation and poor responsiveness to disturbances [**?**]. More broadly, this limitation is consistent with the *No Free Lunch* principle, which states that no single dispatching rule can consistently achieve optimal performance across all manufacturing scenarios [**?**].

These limitations have motivated the adoption of learning-based approaches capable of responding to the stochastic and dynamic conditions of semiconductor manufacturing environments in real time [**?**]. Machine learning techniques, and reinforcement learning (RL) in particular, have emerged as a particularly promising direction, with algorithms such as Deep Q-Network, Advantage Actor-Critic, and Proximal Policy Optimization demonstrating substantial scheduling improvements in simulated semiconductor environments. In some cases total tardiness was reduced from values close to 55% under conventional DRs such as EDD and FIFO to nearly zero in simulated environments [**?**]. These results highlight the potential of RL-based approaches to deliver more adaptive and responsive dispatching policies in highly dynamic semiconductor environments.

Beyond simulation results, there is growing industrial evidence of the benefits of intelligent dispatching in semiconductor manufacturing. Applied Smart Factory reports that customers deploying its dispatching solutions in real wafer fabs improve throughput by 5–10%, with a further 3–5% productivity gain when dispatching is combined with scheduling optimisation [**?**]. More broadly, recent literature reviews confirm that RL-based dispatching approaches have been successfully validated in industrial semiconductor settings, demonstrating consistent improvements over conventional static rules in cycle time, WIP, and on-time delivery [**?**, **?**].

Building on these developments, this dissertation adopts a different perspective, instead of replacing existing dispatching logic, the proposed approach leverages RL techniques to dynamically select, at each dispatching event, the most suitable rule from the set already exposed by the MES platform [**?**]. Critically, the proposed agent operates as a meta-layer *above* the MES dispatching interface: it does not replace rule execution but determines *which* rule the MES should apply at each decision epoch, inheriting the interpretability, constraint-compliance, and operational guarantees of each underlying heuristic while gaining the ability to switch strategies as shop-floor conditions evolve. It is important to note that the improvements cited above, such as total tardiness reductions to near zero, are reported in simplified simulation environments with synthetic parameters. In real industrial settings, complex routing structures, heterogeneous machine qualifications, and irregular demand patterns substantially constrain achievable gains; the objective of this dissertation is to demonstrate a consistent, measurable improvement over static baselines on real production data. Because that data carries these constraints already, rather than abstracting them away, such an improvement arguably transfers more readily to deployment than a performance ceiling established under simplified simulation.

1.3 Objectives

- To develop a discrete-event simulation environment of a semiconductor fabrication facility from real MES event data, providing a realistic training and evaluation platform for the

proposed dispatching agent;

- To design and train a reinforcement learning agent capable of learning adaptive dispatching policies, investigating the influence of reward formulation and observation design on learning stability, convergence behavior, and scheduling performance relative to a baseline configuration;
- To perform a comparative analysis between the RL-based meta-dispatching agent and traditional priority dispatching rules, evaluating performance in terms of average tardiness, makespan, and on-time delivery rate under diverse operating conditions.

1.4 Research Questions

This dissertation is guided by the following research questions:

- **RQ1:** To what extent can a reinforcement learning meta-dispatching agent outperform conventional static dispatching rules in terms of average tardiness, makespan, and on-time delivery rate under diverse operating conditions?
- **RQ2:** How does reward function design influence learning stability, convergence behaviour, and the quality of the dispatching policies learned by the agent?
- **RQ3:** How does the performance of the RL-based meta-dispatching agent compare to static priority dispatching rules under realistic shop-floor disturbances such as equipment failures, elevated maintenance density, and demand variability?

1.5 Sustainable Development Goals

The SDGs established by the United Nations represent a global commitment to addressing economic, social, and environmental challenges, promoting balanced and sustainable growth [**?**]. In the manufacturing sector, the adoption of innovative technological solutions plays a crucial role in responding to these goals, enabling not only improvements in productive efficiency but also the minimization of environmental impacts and the boosting of economic development. In this context, this project directly aligns with four fundamental SDGs for modern industry:

The connection between this work and the four goals is summarised in Table **??** and operates through a single mechanism. Semiconductor fabrication is among the most energy-intensive of all industrial activities, with a modern wafer fab drawing a continuous 50–100 MW [**?**, **?**], so any reduction in machine idle time has a direct energy footprint. By keeping high-power equipment productively occupied through adaptive rule switching, the meta-dispatcher improves utilisation and lowers per-unit energy intensity (Target 7.3), while the resulting gains in throughput and on-time delivery support more resource-efficient, less wasteful production (Target 12.2). Because the agent selects among the heuristics already exposed by the MES rather than replacing them,

it advances data-driven, AI-supported decision-making on the shop floor while preserving the interpretability of each rule, contributing to the digitalisation and innovation goals of Industry 4.0 (Targets 8.2 and 9.4) [**?**, **?**, **?**, **?**]. In combining technological innovation with operational efficiency, the project aligns with a manufacturing model that is more resilient, resource-conscious, and sustainable.

1.6 Chapter Roadmap

The remainder of this dissertation is structured as follows. Chapter **??** establishes the theoretical background of the thesis, covering the semiconductor manufacturing environment, production scheduling foundations and reinforcement learning foundations. Chapter **??** reviews recent literature on the application of reinforcement learning to production dispatching and semiconductor scheduling, identifying open challenges and motivating the proposed approach. Chapter **??** details the reinforcement learning methodology, simulation environment, and experimental setup. Chapter **??** presents experimental results and comparative evaluations. Finally, Chapter **??** reflects on the practical implications of the proposed approach and outlines conclusions and future research directions.

Table 1.1: Project contribution to the Sustainable Development Goals

SDG	**Target**	**Contribution**	**Performance indicators and metrics**
SDG 7 Affordable and Clean Energy	7.3	Semiconductor fabrication facilities operate at continuous power loads of 50–100 MW. By reducing machine idle time through adaptive rule switching, the RL meta-dispatcher improves machine utilisation and decreases standby energy consumption, contributing to energy-efficient production operations.	Machine utilisation (%); Machine idle time (%); Cycle time (h).
SDG 8 Decent Work and Economic Growth	8.2	The project employs a Reinforcement Learning meta-dispatcher to adaptively select dispatching rules at each decision epoch, optimising lot sequencing, increasing throughput, and promoting AI-driven decision-making in semiconductor manufacturing.	On-Time Ratio (OTR) (%); Mean tardiness (hours/lot); Cycle time (h).
SDG 9 Industry, Innovation and Infrastructure	9.4	The RL-based dispatching agent enables adaptive, real-time production control, reducing machine idle time and improving resource utilisation across the fabrication line through intelligent, data-driven scheduling.	Machine utilisation (%); Cycle time (h); Rule selection diversity across scenarios (%).
SDG 12 Responsible Consumption and Production	12.2	By minimising tardiness and reducing cycle times, the meta-dispatcher contributes to leaner production flows, improving the overall efficiency of resource consumption throughout the manufacturing process.	Mean tardiness (hours/lot); Cycle time (h); Machine utilisation (%).

Chapter 2

Background Knowledge

Building on the motivation established in Chapter **??**, this chapter develops the theoretical foundations required to study adaptive dispatching in semiconductor manufacturing systems. The purpose of the chapter is to characterize the operational structure and constraints of wafer fabrication and to formalize the scheduling and learning concepts, establishing the conceptual design space within which the methodological choices of the thesis are later made.

The chapter is organized as follows. Section **??** describes the semiconductor manufacturing environment and highlights the structural sources of scheduling complexity, including reentrant flows, qualification constraints, batching and queue-time restrictions. Section **??** introduces the scheduling formulations most relevant to this work, with emphasis on the job shop and flexible job shop settings, classical dispatching rules and the motivation for dynamic rule selection. Section **??** briefly situates machine learning within this context and distinguishes its main paradigms. Section **??** then presents the reinforcement learning framework for sequential decision making.

2.1 Industrial Context: Semiconductor Wafer Fabrication

2.1.1 Front-End Process and Equipment

Semiconductor manufacturing is a complex, multi-stage process in which raw silicon wafers are transformed into integrated circuits through a sequence of highly specialized physical and chemical operations [**?**, **?**].

The overall manufacturing flow of the wafers is broadly divided into two main stages. In the first stage, the front-end, the electronic structures are built directly on the wafer, layer by layer. Then, the wafers are cut into individual chips and assembled into final products in the back-end stage. From a scheduling point of view, the front-end stage is the most important, since it contains most of the processing steps, including resource limitations and operational complexity [**?**].

Figure **??** shows the complete manufacturing flow, from wafer fabrication to final assembly, while Figure **??** focuses on the major stages of front-end processing. Throughout this stage, wafers undergo deposition, lithography, etch, ion implantation, and chemical-mechanical polishing, with cleaning and metrology operations interspersed between the process steps [**?**]. In practice, this

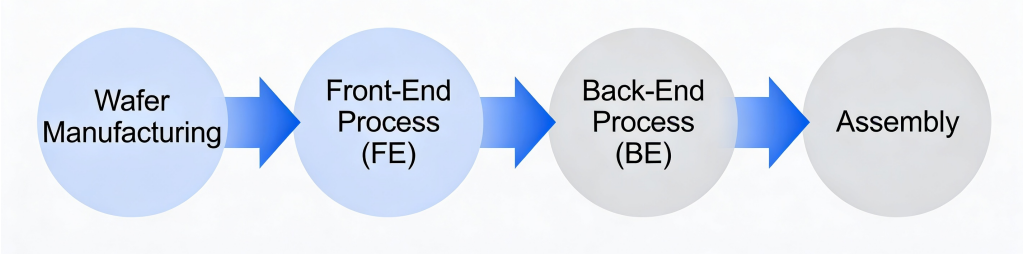

Figure 2.1: Overall semiconductor manufacturing flow

sequence is not executed just once. With each additional device layer, the same set of operations must be repeated, sending lots back through the same family of tools tens or hundreds of times over a single product flow. This pattern is known as reentrant routing, and it is the structural property that most sharply distinguishes semiconductor scheduling from the classical job-shop [**?**], formally introduced in Section **??**.

2.1.2 Basic Operational Model

As the number of layers and processing steps increases, the interactions between different operations become increasingly intricate. Decisions taken at one stage may propagate downstream and repeated visits to the same machines often cause bottlenecks to emerge over time. For this reason, semiconductor scheduling cannot be understood purely as a collection of isolated local decisions, but must instead be viewed as a tightly coupled system of interdependent resource allocations.

In semiconductor facilities, machines are typically organized into tool groups that represent the main units for scheduling decisions. Each tool group contains machines with similar capabilities that can process the same family of recipes. However, not all machines can process every product or recipe because the qualification matrix is sparse and evolves dynamically as machines gain or lose authorization to run specific recipes [**?**]. Therefore, machine assignment decisions must account for not only technological routing constraints, but also equipment qualification restrictions and machine-specific processing characteristics. Even within the same tool group, processing times may differ due to equipment condition, calibration drift, or recipe-specific effects [**?**].

A further defining characteristic of wafer fabrication is that wafers are not scheduled individually. Instead, they are grouped into lots and transported in Front Opening Unified Pods (FOUPs), each of which can hold up to 25 wafers. This grouping reduces handling effort and protects material from contamination, but it also means that scheduling decisions are made at the lot level rather than at the wafer level. Each lot can be processed by only one machine at a time, and once processing has started, it is assumed to be non-preemptive.

2.1.3 Equipment Processing Modes

Although the concepts of lots, steps, and scheduling constraints apply throughout the factory, the operational behavior of equipment depends on the physical architecture of the tools involved. In the wafer-preparation area studied in this work, two processing modes are distinguished.

Figure 2.2: Main stages of front-end semiconductor manufacturing, showing the process sequence from deposition to wafer sorting.

- **Serial Single-Wafer Machines:** The dominant equipment type, representing 620 of the 622 machines in the dataset. These tools process wafers sequentially, one at a time. Consequently, the total processing duration for a lot is directly proportional to the number of wafers it contains. Dispatching decisions are made at the lot level, with the entire lot modeled as a single continuous operation with an aggregated processing time. Loading, unloading, and intra-lot transitions are implicitly included in this duration. Sequence-dependent setup times apply to approximately 16% of operations, arising when consecutive lots require different recipe families on the same machine.
- **Parallel-Chamber Machines:** Two machines in the dataset (BONDER-1 and WETSTRIP-1, anonymized machine identifiers; see Appendix **??**) have multiple internal processing chambers in addition to dedicated load ports. Physically, these machines can overlap loading and processing across chambers. However, in the simulation developed for this thesis, these machines are modelled as single-slot resources, consistent with the conservative assumption that chamber-level scheduling is managed at a lower layer of the MES. Load port availability is tracked and enforced. Sequence-dependent setups do not apply to these tools.

In both cases, lots are non-preemptive once started, and each lot occupies exactly one machine at a time. The absence of batch processing in this dataset simplifies capacity constraints but places the full scheduling burden on sequencing decisions, making dispatching rule selection the primary level for performance optimization.

2.1.4 Priorities and Hot Lots

While temporal constraints and machine availability dictate which lots *can* be processed, they do not indicate which lots *should* be processed first. Not all lots share the same operational importance. To ensure critical orders are met, lots are assigned a *priority* (a positive natural number, where larger values indicate greater importance). Certain urgent orders, known as hot lots, may jump ahead of the standard queue [**?**]. Furthermore, priority is not static so, if a lot approaches its maximum queue-time deadline without starting its subsequent step, the dispatching system invokes a priority escalation mechanism, overriding standard heuristics to protect wafer integrity. This priority signal is one of the operational levers the dispatching policy must reconcile against throughput and due-date pressure at each decision epoch.

2.1.5 From Predictive Scheduling to Real-Time Dispatching

Managing these competing priorities, targets, and constraints over a continuous flow of incoming lots naturally limits the applicability of static planning approaches. While traditional job-shop models often focus on predictive scheduling generating a full future timeline for operations and flexibly routing lots across parallel machines, semiconductor dispatching relies heavily on a real-time, state-driven approach.

The primary reason for this is the strictness of the qualification matrix and the stochastic nature of the shop floor. In this facility, the set of eligible machines for a given lot at a specific step is highly restricted. This restriction is formally represented as an *eligibility mask*, which is often so sparse that it reduces the choice to a single qualified machine. Consequently, the primary lever available to the dispatching system is *sequencing* (deciding the order of lots on a uniquely designated machine) rather than assignment routing.

Furthermore, predictive schedules are frequently invalidated by the operational realities detailed in the preceding sections: unexpected machine downtime, statistical process control holds, and rework excursions. In response, decisions must be made locally and dynamically whenever a machine becomes idle or a new lot arrives.

Because these local dispatching decisions cascade through the reentrant routing of the fab, they shape the evolution of cycle times, downstream resource utilization, and overall due-date performance. The quality of a dispatching policy therefore depends on how it navigates the inherent trade-offs between these different operational goals.

2.1.6 Performance Metrics and Downtime Decomposition

To evaluate these trade-offs, several metrics must be tracked simultaneously (Table **??**), and they often pull in competing directions. Under steady-state conditions, three of the most critical operational metrics such as cycle time, throughput, and work-in-process (WIP) are mathematically coupled through Little's Law [**?**]:

$$\mathrm{WIP} = \mathrm{TH} \cdot \mathrm{CT}.$$

This relationship dictates that cycle time and WIP cannot both be reduced simultaneously for a given throughput level. In practice, manufacturing systems trace a non-linear CT–WIP operating curve: at low WIP, machines frequently sit idle and cycle time is dominated purely by processing times. However, as WIP builds, machine utilization rises until the bottleneck saturates, beyond which any additional WIP merely inflates queueing delays without improving throughput [**?**]. The real-time sequencing choices made by the dispatcher accumulate over time, collectively determining where on that curve the system operates.

Beyond lot-level metrics, a second relationship central to evaluating dispatching performance is the decomposition of machine downtime. From the facility perspective, any hour a machine spends not processing is lost capacity, but the operational cause of that loss matters. Idle time caused by an empty queue is a fundamentally different problem from setup time caused by a

Table 2.1: Performance metrics used to evaluate dispatching policies. ↓: minimise; ↑: maximise.

Metric	**Definition**	
Makespan ($C_{\max}$)	Time from first lot release to last lot completion.	↓
Cycle time (CT)	Mean time a production lot spends in the system.	↓
Work-in-process (WIP)	Mean number of lots simultaneously present in the system.	↓
Production tardiness	$\sum_j \max(C_j - d_j, 0)$ over production lots.	↓
On-time delivery rate (OTR)	Fraction of production lots completed before d_j.	↑
Machine utilisation	Fraction of time machines are actively processing.	↑
Queue-time violations	Number of lots exceeding an inter-step time-lag bound.	↓

recipe change, and both differ from a forced hold due to a quality alert. The primary components of non-processing time include:

- **Idle:** The machine is available and qualified, but no lot is dispatched to it.
- **Setup:** The machine is being configured for a different recipe. This sequence-dependent setup time is the direct cost of switching product families between consecutive lots, and is heavily influenced by the dispatcher's sequencing choices.
- **Hold:** The MES has blocked the lot from processing, typically because a queue-time constraint is at risk or a quality alert has been raised.
- **Abort:** A processing attempt has failed; the lot returns to the queue and the machine chamber must be prepared again.
- **Off-flow:** The lot has been temporarily routed to a side flow (e.g., for sampling or metrology verification) and is unavailable for normal dispatching.
- **Rework:** The current step must be repeated due to a process defect, inducing a time penalty and backward routing.
- **Maintenance:** The machine is unavailable due to planned preventive maintenance or unplanned breakdowns.

In aggregate, the total non-productive time of a machine can be expressed as:

$$T_{\text{down}} = T_{\text{idle}} + T_{\text{setup}} + T_{\text{hold}} + T_{\text{abort}} + T_{\text{offload}} + T_{\text{rework}} + T_{\text{maint}}.$$

Evaluating a dispatching policy requires understanding which of these components it aggravates or alleviates. For instance, a high-WIP scenario driven by an elevated abort rate requires a different dispatching response than one driven by poor setup optimization. A robust scheduling system must account for these distinct sources of delay alongside standard queue waiting times.

These metrics and decomposition principles form the practical foundation upon which scheduling policies are evaluated in semiconductor manufacturing. To solve these optimization challenges algorithmically, the industrial realities detailed throughout this section must first be mapped into formal scheduling theory, as presented in the following section.

2.2 Manufacturing Scheduling Foundations

The operational characteristics described in the previous section can be interpreted through the lens of manufacturing scheduling theory. Scheduling concerns the allocation of limited manufacturing resources to competing jobs over time under precedence, capacity, and performance constraints. Among classical formulations, the Job Shop Scheduling Problem (JSSP) has become one of the main reference models for reasoning about such decisions [**?**, **?**]. It is also well established that the JSSP is NP-hard, which helps explain why exact optimization rapidly becomes impractical as problem size and operational constraints increase [**?**].

Although classical job shop models are intentionally simplified, they provide a useful starting point for formalizing the structure of semiconductor production control problems. They make it possible to describe, in a common language, how operational decisions arise from the interaction between routing constraints, machine availability, and optimization objectives. At the same time, the gap between these classical formulations and real wafer-fab environments is substantial, which is why industrial settings are more naturally understood through enriched variants such as the Flexible Job Shop Problem (FJSP) and related dynamic extensions [**?**, **?**].

To describe such problems systematically, Graham et al. introduced the $\alpha|\beta|\gamma$ notation, in which the machine environment (α), side constraints (β), and optimization objective (γ) jointly characterize a scheduling instance [**?**]. This framework is particularly useful here because it clarifies how the industrial features discussed earlier, such as machine flexibility, batching, setup dependence, and time-lag constraints, modify the structure of the underlying scheduling problem. The remainder of this section adopts this perspective to situate semiconductor dispatching within the broader scheduling literature while highlighting the additional complexities of the decision setting addressed in this dissertation.

2.2.1 The Classical Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) is defined over a set of jobs $J = \{1,\dots,n\}$ and a set of machines $M = \{1,\dots,m\}$. Each job $j \in J$ comprises an ordered sequence of operations $O_{j1}, O_{j2}, \dots, O_{jn_j}$. Each operation O_{ji} must be processed on a specific machine $\mu(j,i) \in M$ for a fixed and uninterruptible processing time $p_{ji} > 0$. A feasible schedule assigns a start time $s_{ji} \geq 0$ to each operation such that two fundamental conditions are satisfied. First, operations within the same job must respect their technological order:

$$s_{j,i+1} \geq s_{ji} + p_{ji} \tag{2.1}$$

Second, machine capacity constraints ensure that no machine processes more than one operation at any given time.

The most common objective is makespan minimization [**?**]:

$$C_{\max} = \max_{j \in J} C_j, \quad \text{where } C_j = s_{jn_j} + p_{jn_j} \tag{2.2}$$

where $C_j = s_{jn_j} + p_{jn_j}$ is the completion time of job j. Other common objectives include mean flow time, total weighted tardiness, and number of late jobs.

This representation highlights an important aspect of the problem: scheduling is fundamentally about resolving ordering decisions under constraints, rather than simply assigning start times.

The combinatorial difficulty of the JSSP is captured by its disjunctive graph in which operations are nodes, conjunctive arcs encode job precedence, and undirected disjunctive arcs connect operations sharing the same machine. A feasible schedule corresponds to an acyclic orientation of all disjunctive arcs; the makespan equals the length of the longest path in the resulting directed acyclic graph. This representation underpins both classical branch-and-bound algorithms and, more recently, graph-based learning approaches [**?**].

2.2.2 Taxonomy of JSSP Extensions

Although theoretically foundational, the classical JSSP abstracts away several features that are essential in real manufacturing systems. Table **??** summarizes the most industrially relevant extensions, mapping the physical and operational constraints described in Section **??** into formal scheduling taxonomy.

Table 2.2: Extensions to the classical JSSP and their relevance in semiconductor wafer fabrication. Based on Dauzère-Pérès et al. [**?**] and Mönch et al. [**?**].

Extension	Modification to classical JSSP	Semiconductor instance
Flexible JSSP (FJSP)	Each operation can be processed on any machine from an eligible subset $M_{ij} \subseteq M$, with potentially different processing times p_{ijm}	Parallel tool groups; equipment qualification constraints
Sequence-dependent setup times	Setup time between consecutive operations depends on the recipe or product type of the preceding job	Recipe changeovers between product families
Batch processing	A machine can process multiple jobs simultaneously up to a capacity limit	Single-chamber and pipeline operations; compatible job families
Reentrant flows	Jobs revisit the same machine groups multiple times along their route	Wafer fabrication: steps cycling through lithography, etch, and deposition
Dynamic arrivals	New jobs arrive continuously during the planning horizon	Continuous lot release; hot lot expediting
Machine unavailability	Machines are temporarily unavailable due to unplanned breakdowns or scheduled preventive maintenance windows	Random equipment failures; maintenance cycles
Queue time constraints	Maximum allowable waiting time between specific pairs of process steps; violations require rework or scrap	Material degradation windows and contamination prevention

As shown in Table **??**, the simultaneous presence of reentrant flows, sequence-dependent setups, batching, and queue-time constraints places semiconductor fabrication among the most complex extensions of the JSSP [**?**]. The difficulty does not arise from each feature in isolation, but from their highly coupled interaction.

For instance, reentrant routing amplifies the impact of machine unavailability, as disruptions propagate across multiple revisits to the same tool groups. At the same time, flexible assignment decisions made early in the process influence resource availability at downstream stages. Furthermore, because each lot follows a fixed product-specific route with many ordered steps that cannot be resequenced, routing rigidity itself becomes a binding constraint. Because the theoretical formulation of these combined extensions yields a dynamic, strongly NP-hard problem, offline predictive schedules become computationally intractable and operationally fragile, reinforcing the need for the real-time dispatching approaches motivated in Section **??**.

2.2.3 The Flexible Job Shop Problem

The Flexible Job Shop Problem (FJSP) generalises the JSSP by allowing each operation O_{ij} to be processed on any machine from a non-empty eligible set $M_{ij} \subseteq M$, with processing times p_{ijm} that may vary across machines [**?**].

A feasible solution therefore requires two interdependent decisions for every operation:

1. **Machine assignment**: selecting $m_{ij} \in M_{ij}$;
2. **Operation sequencing**: determining the processing order of operations on each machine.

These decisions are tightly coupled: assigning operations to machines directly affects the resulting sequencing decisions, and vice versa.

The FJSP strictly generalizes the classical JSSP, which corresponds to the special case where $|M_{ij}| = 1$ for all operations. As a result, the problem remains strongly NP-hard [**?**], with machine flexibility significantly increasing the size of the solution space.

Due to this complexity, exact optimization methods quickly become computationally infeasible for realistic industrial instances. In practice, solution approaches rely on heuristics, metaheuristics, or decomposition-based strategies that trade optimality guarantees for scalability and responsiveness.

In semiconductor manufacturing, the FJSP provides a natural modeling framework, as parallel tool groups with qualification constraints directly define the eligible machine sets $M_{ij} \subseteq M$. However, the limitations of classical optimization methods in stochastic and dynamic environments motivate the learning-based approach developed in Chapter **??**.

2.2.4 Solution Approaches

Decades of work on the JSSP and FJSP have produced a well-understood spectrum of solution methods, each occupying a different position along the trade-off between solution quality, computational cost, and responsiveness to dynamic events. Table **??** summarizes these methods.

Exact methods such as mixed-integer linear programming (MILP) and constraint programming (CP) can mathematically guarantee optimal solutions. However, in practice, they become computationally intractable beyond instances of roughly twenty jobs. More critically, they operate entirely offline, requiring the full problem parameters to be specified before execution begins. Any

Table 2.3: Comparison of solution approaches for the scheduling problem. *Online*: decisions are made without pre-computing a full schedule. *Adaptive*: behaviour adjusts to the current system state.

Method	**Optimality**	**Online**	**Adaptive**	**Scalable to fab**
Exact (MILP, CP)	Guaranteed	No	No	No
Metaheuristics	Near-optimal	No	No	Limited
Static dispatching rules	Heuristic	Yes	No	Yes
RL — direct lot selection	Learned	Yes	Yes	Limited
RL — rule selector (this work)	Learned	Yes	Yes	Yes

machine failure or new lot arrival invalidates the current plan and forces a complete re-solve, an option that is computationally prohibitive in a fab with hundreds of active lots [**?**].

Metaheuristics such as genetic algorithms and tabu search [**?**] produce near-optimal solutions on larger instances within a reasonable time. Nevertheless, they still operate in batch mode. In a semiconductor facility where disruptions arrive continuously, repeatedly re-solving a full schedule introduces severe computational overhead and scheduling instability, since lots already in motion may be rerouted mid-process [**?**].

Static dispatching rules avoid both problems by resolving queue conflicts locally in real time. Because they impose negligible computational cost ($\mathcal{O}(n)$ per decision) and integrate naturally into MES interfaces, they remain the dominant industrial mechanism. Their fundamental limitation, however, is context-insensitivity: a fixed priority function cannot alter its logic when the operating regime shifts (e.g., from an idle state to a heavy backlog), and no single rule consistently dominates across all operating conditions [**?**].

This gap motivates the **learning-based approach** developed in this thesis. Applying Reinforcement Learning (RL) directly to lot selection, where the agent chooses the specific lot to process from all eligible candidates, suffers from severe scalability issues because the action space grows dynamically with the number of waiting lots. By keeping the action space fixed to a set of predefined dispatching heuristics, the RL agent acts as a *rule selector*. This preserves the scalability and MES compatibility of classical dispatching while enabling context-sensitive adaptation, allowing the system to shift toward urgency-aware rules when due dates are tight, and toward throughput-maximizing rules when the line is underloaded [**?**].

Formally, a dispatching rule is a priority function $\phi : \mathcal{Q}(t) \to \mathbb{R}$ that scores each lot waiting in queue $\mathcal{Q}$ at time t. The lowest-scoring lot is selected whenever a machine becomes free. Computationally cheap and transparent, these rules are the building blocks of real-time shop-floor control [**?**].

Different rules draw on different sources of information. Some score lots using only the processing time of the next operation; others factor in due-date urgency, remaining workload, or setup costs. While rules that integrate more information can respond better to global fab conditions, they are also more sensitive to estimation errors. Six fundamental rules serve as both the benchmark baselines and the action space for the RL agent in this thesis. They are listed in Table **??**, following

the standard notation of Pinedo [**?**].

Table 2.4: Dispatching rules used in this thesis. Notation: t_j^{arr} arrival time; p_j^{next} processing time of next operation; d_j due date; t current time; w_j remaining processing time; s_{ij} setup time; $\bar{p}$ mean processing time; $\bar{s}$ mean setup time; k_1, k_2 scaling parameters. All six scores share a single convention, the lowest-scoring eligible lot is dispatched first; ATCS is the only rule whose natural index runs the opposite way, so it carries an explicit leading minus sign to fit the same convention.

Rule	**Score (lower dispatched first)**	**Strength**	**Weakness**
FIFO	t_j^{arr}	Fairness; prevents starvation	Ignores operational and timing info.
SPT	p_j^{next}	Minimises WIP and cycle time	Long lots may starve.
EDD	d_j	Minimises maximum tardiness	Ignores remaining workload.
LST	$d_j - t - w_j$	Reactive to slack time	Highly sensitive to errors in w_j.
CR	$(d_j - t)\,/\,w_j$	Balances cycle time and tardiness	Unstable as $w_j \to 0$.
ATCS	$-\frac{1}{p_j^{\text{next}}} \exp\left(\frac{-\max(d_j - p_j^{\text{next}} - t, 0)}{k_1 \bar{p}}\right) \exp\left(\frac{-s_{ij}}{k_2 \bar{s}}\right)$	Integrates throughput, urgency, and setups	Requires calibration of k_1, k_2.

Empirical studies confirm that the performance of these rules depends strictly on the current operating regime, such as line load, due-date tightness, and breakdown frequency [**?**, **?**]. Because no fixed rule consistently dominates, measurable gains can be achieved by dynamically switching between rules as shop-floor conditions change.

2.2.4.1 Hyper-Heuristic Rule Selection

This sensitivity is the central motivation for hyper-heuristics, procedures that search over the space of heuristics rather than directly over the solution space [**?**]. Rather than replacing or redesigning the rules themselves, the approach rests on a clean separation of concerns: a high-level selector observes the current system state and determines which low-level rule to activate at each dispatching event, while the individual rules remain unchanged. Precisely how this selector is instantiated in the present work, including the choice of action space, policy architecture, and rule set, is motivated and detailed in Chapter **??**.

Recent work demonstrates that policy-gradient algorithms are highly effective hyper-heuristic selectors in job-shop environments, outperforming both static rules and metaheuristic-based selectors by learning complex state-action mappings [**?**]. The meta-dispatching agent proposed in this dissertation adopts exactly this architecture: using the six classical rules from Table **??** as its discrete action space to adapt to stochastic fab conditions in real time.

However, training such an agent directly on a live factory floor is operationally infeasible, as the trial-and-error nature of the learning process would cause unacceptable production disruptions. Consequently, the agent must be trained within a discrete-event simulation (DES) environment that

accurately replicates the physical constraints and reentrant routing of the real fab. To understand how a computational agent can actually learn to make these dispatching decisions inside a simulator, it is necessary to first introduce the underlying mathematical principles of these algorithms.

2.3 Machine Learning Fundamentals

This section introduces the machine learning concepts needed to understand the approach developed in this thesis, focusing only on ideas directly relevant to scheduling. It distinguishes the three principal ML paradigms before Section **??** develops, in full, the reinforcement learning framework central to this research, covering Markov Decision Processes, value functions, and Proximal Policy Optimization. The limitations of exact scheduling methods and static heuristics discussed in the previous sections highlight a fundamental challenge: it is exceptionally difficult to explicitly program the correct decision logic for every possible shop-floor state. Machine learning (ML) addresses this by automatically extracting patterns and improving task performance from data or experience, rather than relying on hand-coded rules [**?**]. ML is traditionally divided into three paradigms: *supervised learning*, which learns an input–output mapping from labelled examples; *unsupervised learning*, which discovers latent structure in unlabelled data; and *reinforcement learning*, in which an agent learns through trial-and-error interaction guided by scalar reward signals. The first two have been applied to scheduling for processing-time prediction and imitation learning from expert schedules, but fall outside this dissertation's scope: the dispatching policy is learned through direct environmental interaction rather than from labelled demonstrations. Reinforcement learning, developed in full in Section **??**, is therefore the sole learning paradigm employed.

2.4 Reinforcement Learning Foundations

Having distinguished reinforcement learning from the other two paradigms above, this section develops it in full, following the standard formalisation of Sutton and Barto [**?**]: how the agent-environment interaction is structured as a Markov Decision Process, how policies and value functions are defined over it, and how modern policy-gradient algorithms, PPO in particular, learn from that interaction. This framework is particularly suited for dynamic decision-making problems where optimal strategies must be discovered through experience, such as real-time dispatching in semiconductor manufacturing.

2.4.1 The Markov Decision Process Framework

The formal foundation of reinforcement learning is the **Markov Decision Process (MDP)**, a mathematical framework that models sequential decision-making under uncertainty [**?**, **?**]. An MDP is defined by the tuple $\langle \mathscr{S}, \mathscr{A}, T, R, \gamma \rangle$, where:

- $\mathscr{S}$ is the **state space**, representing all possible configurations of the environment (e.g., machine status, task queue contents, current workload).
- $\mathscr{A}$ is the **action space**, containing all decisions available to the agent (e.g., selection of dispatching rules, task assignments).
- $T(s'|s,a)$ is the **transition probability function**, where $T(s'|s,a)$ denotes the probability of transitioning to state s' after taking action a in state s. The transition dynamics may be stochastic, capturing uncertainty in processing times, machine failures, or job arrivals.
- $R(s,a)$ is the **reward function**, providing immediate feedback $r_t = R(s_t, a_t)$ that evaluates the quality of the action taken (e.g., penalizing tardiness, idle time, or excessive WIP).
- $\gamma \in [0,1]$ is the **discount factor**, determining the present value of future rewards. A discount factor close to 1 emphasizes long-term consequences, while values near 0 prioritize immediate rewards.

The Markov Property The **Markov property** ensures that the future evolution of the system depends only on the current state and action, independent of the history [**?**]:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t)$$

This memoryless property enables tractable optimization: the agent need not maintain or reason about the entire history of states and actions, only the current state. While real-world manufacturing systems may exhibit non-Markovian characteristics (e.g., equipment degradation over time), carefully designed state representations can often approximate Markovian behavior sufficiently for RL to be effective.

Agent-Environment Interaction Figure **??** illustrates the fundamental agent-environment interaction cycle. At each discrete time step t, the agent observes the current state $s_t \in \mathscr{S}$, selects an action $a_t \in \mathscr{A}$ according to its policy, receives a scalar reward $r_t = R(s_t, a_t)$, and transitions to a new state $s_{t+1} \sim T(\cdot|s_t, a_t)$. This cyclical process continues until a terminal state is reached or a predefined time horizon is exceeded. Figure **??** itself follows Sutton and Barto's original indexing, in which the reward and next state produced by the environment are labelled R_{t+1} and S_{t+1} before becoming the agent's R_t and S_t at the following step; this thesis instead indexes the reward at the same time step as the state and action that produced it ($r_t = R(s_t, a_t)$, consistent with Chapter **??**), which is only a difference in labelling convention, not a different causal sequence.

2.4.2 Policies and Value Functions

2.4.2.1 Policies

A **policy** π defines the agent's behaviour by mapping states to actions. It may be *deterministic*, selecting a single action $a = \pi(s)$ for each state, or *stochastic*, defining a probability distribution

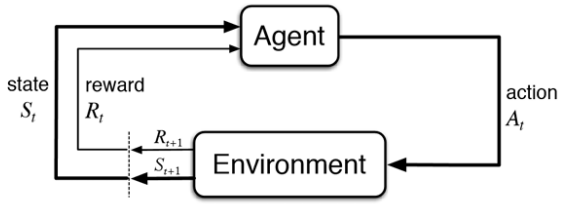

Figure 2.3: Agent-environment interaction loop in reinforcement learning. At each time step t, the agent observes state S_t, executes action A_t, receives reward R_{t+1}, and transitions to state S_{t+1}. Adapted from Sutton and Barto [**?**].

$\pi(a|s)$ over actions. The stochastic form is the more useful one here, since it builds in the exploration needed to cope with the uncertainty of a dynamic manufacturing system. The objective of reinforcement learning is to find an **optimal policy** π^* that maximises the expected cumulative discounted reward.

2.4.2.2 Value Functions

Value functions quantify the expected return from a given state, or state–action pair, under a specific policy. The **state-value function** $V^\pi(s)$ is the expected return when starting from state s and following π thereafter,

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s], \qquad G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

where the **return** G_t is the cumulative discounted reward from time t onward. The closely related **action-value function** (or Q-function) $Q^\pi(s,a)$ measures the same expected return, but conditions in addition on taking action a in state s before following π: $Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]$.

Bellman Equations Value functions satisfy recursive relationships known as the **Bellman equations**:

$$V^\pi(s) = \sum_{a \in \mathscr{A}} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathscr{S}} T(s'|s,a) V^\pi(s') \right] \tag{2.3}$$

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathscr{S}} T(s'|s,a) \sum_{a' \in \mathscr{A}} \pi(a'|s') Q^\pi(s',a') \tag{2.4}$$

The **optimal value functions** $V^*(s)$ and $Q^*(s,a)$ correspond to the maximum achievable expected return and satisfy the **Bellman optimality equations**:

$$V^*(s) = \max_{a \in \mathscr{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathscr{S}} T(s'|s,a) V^*(s') \right]$$

An optimal policy π^* can be extracted from the optimal Q-function via $\pi^*(s) = \arg\max_a Q^*(s,a)$.

2.4.3 Returns and the Objective Function

The agent's goal is to maximize the **expected return**, formally defined as:

$$J(\pi) = \mathbb{E}_{s\sim\rho^{\pi},a\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}R(s_t,a_t)\right]$$

where ρ^{π} is the state distribution induced by policy π. The discount factor γ balances immediate versus long-term rewards: for $\gamma = 0$, the agent is myopic, considering only immediate rewards; for $\gamma \to 1$, future rewards receive nearly equal weight to immediate ones, encouraging foresight and long-term planning.

In episodic tasks (e.g., scheduling a fixed batch of jobs), the return is finite: $G_t = \sum_{k=0}^{T-t}\gamma^{k}r_{t+k}$, where T is the terminal time step. In continuing tasks (e.g., ongoing production control), discounting ensures the return remains bounded even with infinite horizons.

2.4.4 Taxonomy of RL Algorithms

Reinforcement learning algorithms are categorized based on their approach to learning optimal policies. Figure **??** presents a comprehensive taxonomy adapted from OpenAI Spinning Up [**?**], distinguishing between model-based and model-free methods, and further subdividing model-free approaches into value-based, policy-based, and actor-critic methods.

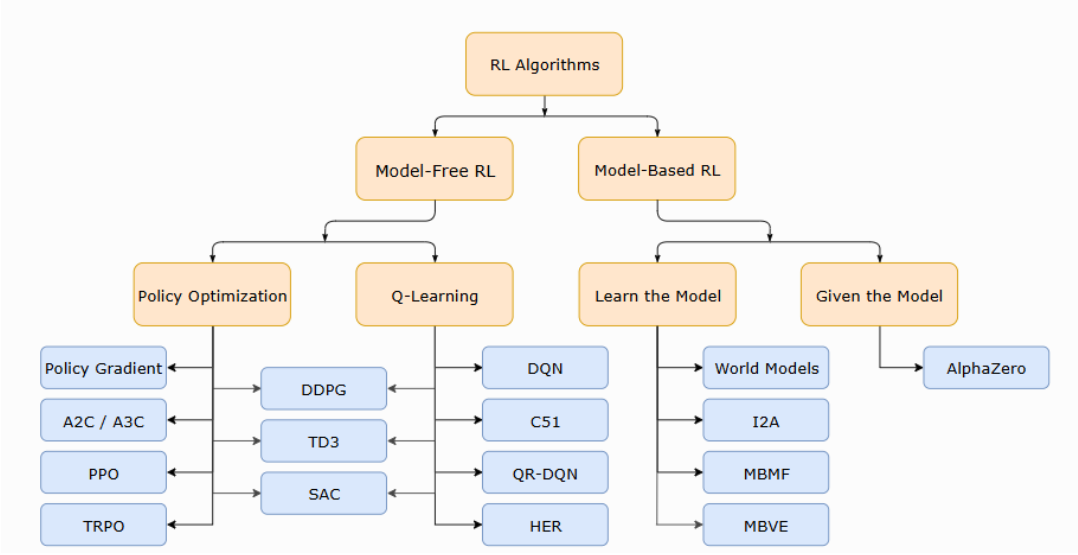

Figure 2.4: Taxonomy of modern reinforcement learning algorithms. Model-free methods dominate industrial applications due to their robustness to modeling errors. Adapted from OpenAI Spinning Up [**?**].

2.4.4.1 Model-Based vs Model-Free Methods

Model-based RL algorithms learn explicit models of the environment's transition dynamics $T(s'|s,a)$ and reward function $R(s,a)$, enabling planning through simulation (e.g., Monte Carlo Tree Search, Dyna-Q) [**?**]. While sample-efficient, these methods suffer from model errors that compound during multi-step predictions, limiting their applicability in complex manufacturing environments where accurate models are difficult to obtain.

Model-free methods directly learn value functions or policies from experience without constructing environment models, trading sample efficiency for robustness and simplicity. Both the value-based and policy-gradient families introduced below belong to this category. This thesis develops the policy-gradient branch in detail, since PPO (Section **??**) is the sole algorithm adopted for the experiments reported in Chapter **??**; value-based alternatives such as DQN are discussed only as points of comparison in the literature review of Chapter **??**.

2.4.4.2 Value-Based Methods

Value-based approaches learn action-value functions $Q(s,a)$ and derive policies by selecting actions that maximize these values: $\pi(s) = \arg\max_a Q(s,a)$.

2.4.4.3 Policy Gradient Methods

Policy gradient methods directly parameterize the policy as $\pi_\theta(a|s)$, where θ represents learnable parameters (e.g., neural network weights), and optimize θ to maximize expected return [**?**, **?**].

The **policy gradient theorem** provides the gradient of the performance objective [**?**]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

2.4.4.4 Actor-Critic Methods

While pure value-based methods learn $Q(s,a)$ directly, and pure policy-based methods learn $\pi(a|s)$ directly, *actor-critic* architectures combine both approaches. They maintain both a parameterized policy (the actor) $\pi_\theta(a|s)$ and a parameterized value function (the critic) $V_\phi(s)$ [**?**]. The critic estimates the value function to evaluate the quality of the actions, while the actor updates the policy probabilities based on this evaluation. This synergy significantly reduces the variance in policy gradient estimates and improves sample efficiency compared to pure policy methods.

2.4.5 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a state-of-the-art actor-critic algorithm that addresses the instability and sample inefficiency of traditional policy gradient methods through constrained policy updates [**?**].

PPO starts from the policy gradient theorem already introduced above,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)],$$

but replaces the raw action-value $Q^{\pi_\theta}(s,a)$ with the advantage function $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$. Subtracting the state-value baseline $V^{\pi_\theta}(s)$ does not change the expected gradient, since any baseline that does not depend on the action has zero expected contribution to the gradient, but it substantially reduces the gradient estimate's variance, which is why actor-critic methods such as PPO use A^{π_θ} rather than Q^{π_θ} in practice:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s,a)].$$

Even with this variance reduction, naive estimates of this gradient still suffer from high variance, and excessively large updates can cause catastrophic performance degradation due to the non-convex nature of policy optimization.

Trust Region Policy Optimization (TRPO) [**?**] addresses this instability by mathematically constraining policy updates to remain within a strict "trust region," bounding the Kullback-Leibler (KL) divergence between successive policies. While effective, TRPO's conjugate gradient-based constraint enforcement is computationally expensive.

PPO simplifies TRPO by replacing the strict KL constraint with a *clipped surrogate objective* that penalizes excessive policy deviations. Defining the probability ratio between the new and old policy as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)},$$

the PPO-Clip objective is formulated as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t,\ \text{clip}\big(r_t(\theta),\, 1-\varepsilon,\, 1+\varepsilon\big)A_t\right)\right],$$

where ε defines the clipping range (set to 0.2 throughout this thesis). This objective prevents large policy updates by clipping the ratio $r_t(\theta)$ when it deviates beyond $[1-\varepsilon, 1+\varepsilon]$, taking the minimum of the clipped and unclipped terms to form a pessimistic (lower-bound) estimate of the true objective. Unlike TRPO's trust region, this does not come with a formal monotonic-improvement guarantee; it is a cheaper heuristic approximation that empirically limits destructively large policy changes in most cases, without the computational cost of TRPO's constrained optimisation.

2.4.5.1 Generalized Advantage Estimation (GAE)

Accurate advantage estimates A_t are critical for reducing variance in these updates. PPO computes advantages using Generalized Advantage Estimation (GAE) [**?**], which builds upon temporal-difference (TD) errors:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Here $r_t = R(s_t, a_t)$ denotes the scalar reward received at time t, unrelated to the probability ratio $r_t(\theta)$ defined above for the PPO-Clip objective; the two are conventionally given the same letter in the RL literature but always carry different arguments in this text. The GAE advantage estimator is defined as an exponentially weighted average of these TD errors:

$$A_t^{\mathrm{GAE}}(\gamma, \lambda) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\lambda \in [0, 1]$ controls the bias-variance trade-off. Setting $\lambda = 0$ reduces the estimator to the one-step TD error (low variance, high bias); setting $\lambda = 1$ recovers the full Monte Carlo return (high variance, low bias). Throughout this thesis, $\lambda = 0.95$ is used to balance these extremes effectively, consistent with standard implementations [**?**, **?**].

Finally, PPO incorporates an *entropy bonus* $\beta\,\mathscr{H}[\pi_\theta]$ into the training objective to discourage premature convergence to a deterministic policy. Because the meta-selector action space in this work consists of only six possible rules, this regularization is critical: without it, the policy quickly locks onto a single dispatching rule. Empirical calibration shows that an entropy coefficient of $\beta = 0.03$ provides the necessary exploration incentives [**?**].

This chapter has established the semiconductor manufacturing context that motivates the dispatching problem, the classical scheduling foundations that static rules build on, and the reinforcement learning vocabulary, MDPs, value functions, policy gradients, and PPO specifically, needed to formalise dispatching as a learning problem. Chapter **??** now turns to how these tools have actually been applied to scheduling in the literature, surveying which industrial constraints and algorithmic choices prior RL-based dispatchers have addressed and which remain open, before Chapter **??** details this thesis's own instantiation of the framework introduced here.

Chapter 3

Reinforcement Learning for Semiconductor Scheduling: State of the Art

Building on the semiconductor manufacturing context and scheduling foundations established in Chapter **??**, this chapter reviews the scientific literature on the application of reinforcement learning to dynamic production scheduling, with particular emphasis on how prior work addresses the specific operational constraints identified in Section **??**. Rather than surveying RL theory in isolation, this chapter maps the literature directly onto the scheduling extensions defined in Table **??** and examines how different algorithmic choices (state representation, action abstraction, reward design) handle the industrial realities outlined in Sections **??** through **??**.

The chapter is organized as follows. Section **??** describes the systematic review methodology. Section **??** maps the literature to the JSSP extensions from Chapter **??**, identifying which studies explicitly address reentrant flows, sequence-dependent setups, queue-time constraints, and dynamic disturbances. Section **??** compares rule-selection and direct-action policies, highlighting the scalability and interpretability trade-offs. Section **??** surveys how prior work represents manufacturing state, comparing tabular, vector-based, and graph-based approaches. Section **??** synthesizes reward function design, with attention to how different studies encode the temporal penalties and downtime categories from Section **??**. Section **??** reviews the RL algorithms most frequently applied to dispatching. Finally, Section **??** identifies gaps and positions this thesis relative to the existing body of work.

3.1 Systematic Review Methodology

To ensure rigour, reproducibility, and comprehensive coverage of the relevant literature, the review presented in this chapter was conducted following the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines [**?**]. The process comprises five stages:

definition of keywords, formulation of search queries, application of inclusion and exclusion criteria, screening and selection of articles, and quantitative analysis of the results.

3.1.1 Keyword Definition

The selection of keywords targets the application of machine learning to real-time dispatching in the context of Manufacturing Execution Systems (MES), with particular emphasis on dynamic resource allocation, dispatching rule selection, and processing time prediction. Four thematic categories were established to represent the central axes of the investigation:

- **Manufacturing Execution Systems:** *"manufacturing execution systems"*, *"MES"*, *"smart manufacturing"*.
- **Scheduling and dispatching:** *"production scheduling"*, *"dynamic scheduling"*, *"resource allocation"*, *"dispatching rules"*, *"real-time dispatching"*.
- **Machine learning models:** *"machine learning"*, *"artificial intelligence"*, *"reinforcement learning"*, *"predictive analysis"*, *"optimization algorithms"*.
- **Semiconductor manufacturing:** *"semiconductor scheduling"*, *"wafer fab"*, *"reentrant routing"*, *"job shop"*.

3.1.2 Search Queries

Two search queries were formulated using Boolean operators to accommodate linguistic and structural variations. An iterative refinement process was applied to balance specificity (ensuring relevance) with recall (avoiding the exclusion of essential studies).

Query SQ1 (broad).

```
("machine learning" OR "ML") AND "dispatching rules"
AND "dynamic scheduling"
```

Query SQ2 (targeted).

```
("machine learning" OR "ML" OR "artificial intelligence"
OR "AI" OR "predictive model" OR "reinforcement learning")
AND ("dispatching rules" OR "real-time dispatching" OR "RTD")
AND ("dynamic scheduling" OR "adaptive scheduling"
OR "production scheduling" OR "resource allocation"
OR "scheduling optimization")
AND ("manufacturing systems"
OR "manufacturing execution systems" OR "MES"
OR "semiconductor" OR "wafer fab")
```

Both queries were executed independently across three reference databases, IEEE Xplore, ACM Digital Library, and Scopus, selected for their high-impact coverage of intelligent manufacturing, production optimisation, and predictive algorithms.

3.1.3 Inclusion and Exclusion Criteria

Inclusion criteria. Articles published in peer-reviewed journals or conference proceedings from 2018 onward; studies focused on the application of machine learning to production scheduling, dispatching, or MES contexts; publications in English with full-text access.

Exclusion criteria. Duplicate articles; publications not available in English; tutorials, workshops, or conference abstracts without scientific validation; studies without practical application to manufacturing scheduling or dispatching.

3.1.4 Quantitative Results

Table **??** summarizes the results of both search queries. In total, 139 documents were retrieved, of which 132 were unique and 7 were identified as duplicates. Of the unique documents, 59 were deemed relevant to the research objectives, while 73 were classified as non-relevant.

Table 3.1: Results of search queries SQ1 and SQ2.

	Total	**Unique**	**Duplicates**	**Relevant**	**Non-relevant**
SQ1	38	34	4	19	15
SQ2	101	98	3	40	58
Total	139	132	7	59	73

Table **??** records the contribution of each database to the retrieval of relevant documents. Scopus contributed the largest share of relevant documents (30), followed by ACM (16) and IEEE (13). The inclusion of all three databases ensured a comprehensive and well-rounded review.

Table 3.2: Distribution of relevant documents by database.

	Relevant	**Non-relevant**	**ACM**	**IEEE**	**Scopus**
SQ1	19	15	2	6	11
SQ2	40	58	14	7	19
Total	59	73	16	13	30

3.2 The Scheduling Problem in the Literature: From JSSP to Semiconductor Extensions

The classical Job Shop Scheduling Problem (JSSP) provides a foundational abstraction for reasoning about production control, but real semiconductor manufacturing deviates from this model

in multiple dimensions. Table **??** of Chapter **??** enumerated these deviations formally. The literature on RL for scheduling exhibits highly uneven coverage of these extensions: some features are addressed in many papers, while others remain largely unexplored.

This section maps the reviewed literature onto each JSSP extension to identify which studies explicitly tackle the specific operational constraints of wafer fabrication; Table **??** consolidates the comparison.

3.2.1 Reentrant Flows

Reentrant routing, where lots revisit the same machines or machine groups multiple times over the course of processing, is among the most distinctive features of semiconductor manufacturing (Section **??**). Classical JSSP formulations assume that each job visits each machine at most once, an assumption that breaks down entirely on the fab floor: once it is violated, machine contention and queue dynamics begin to interact nonlinearly across successive revisits, and the resulting scheduling problem becomes considerably harder to reason about.

Given how central reentrancy is to the domain, it is striking how few RL scheduling papers actually engage with it. The earliest attempt came from Waschneck et al. [**?**], who applied DQN to a simulated wafer fab with reentrant routing and showed that RL-based rule selection beats any static rule on its own, although their tool-group structure was kept deliberately simple. Kuhnle et al. [**?**] went further still, validating PPO-based rule selection on a realistic fab model with multi-layer reentrancy and, notably, real semiconductor process data rather than a synthetic instance. Pushing the idea furthest, Ghaleb et al. [**?**] deployed their RL dispatcher in an actual industrial facility with full reentrant routing, where it held up under machine breakdowns and rush orders. Outside these three, however, the rest of the RL scheduling literature still leans on simplified job-shop models that never revisit a machine, which inevitably limits how much of their reported success can be expected to carry over to a real fab. This thesis follows the more demanding path: re-entrant visits are permitted and common by design (Constraint C3, Section **??** of Chapter **??**), and the 622-machine, 281-flow fab modelled in Chapter **??** exhibits reentrancy at a scale well beyond any of the three studies above.

3.2.2 Sequence-Dependent Setup Times

A second complication is that setup times often depend on what was processed immediately before: switching between recipes can introduce its own cost function on top of ordinary processing time. This is not a marginal effect in the dataset studied in this thesis, where roughly 16% of operations exhibit sequence-dependent setups (Section **??**), yet it receives comparatively little attention in the literature. Among the exceptions, Kuhnle et al. [**?**, **?**] fold setup times directly into both the state representation and the reward function, so that the agent can learn when a setup delay is worth tolerating and when it is not. Freitag and Hildebrandt [**?**] take a different route, evolving setup-aware dispatching rules through genetic programming and showing that the resulting rules can outperform hand-designed heuristics such as ATCS. A third case arrives at something similar

almost by accident: Sun et al.'s [**?**] image-based state representation ends up implicitly encoding setup costs through machine-occupancy patterns, even though setup handling was never their explicit focus. Beyond this small handful of studies, most RL scheduling work simply ignores setup times or folds them into a fixed overhead term, a simplification that quietly erodes how applicable these methods are to a real fab. This thesis instead derives a full sequence-dependent setup-time matrix from historical recipe-family transitions and feeds it into both the dispatcher's machine-selection stage and the reward function (Section **??** of Chapter **??**), so that the setup awareness Kuhnle et al. [**?**, **?**] pioneer at a small scale is here grounded in empirically observed changeover times across 622 machines.

3.2.3 Queue-Time Constraints

Queue-time constraints, the maximum inter-step waiting windows described in Section **??**, are not a modelling convenience but a hard physical limit: if a lot waits too long between two steps, the material itself can degrade or require rework, with direct consequences for cycle time and yield. Despite this, the literature treats the constraint almost as an afterthought. Kuhnle et al. [**?**] acknowledge the constraint, but only as a soft reward penalty, letting the agent trade violations off against other objectives rather than treating them as non-negotiable. The same tension surfaces in Mönch et al.'s [**?**] semiconductor scheduling survey, which flags queue-time constraints as an important industrial feature while noting in the same breath that most optimisation methods drop them for the sake of tractability; even classical approaches such as MILP and constraint programming, capable of enforcing queue-time windows explicitly, do not scale to dynamic, online settings. None of the RL studies reviewed for this thesis, in fact, enforces queue-time constraints as a hard constraint within the MDP formulation itself, leaving a fairly clear gap between what the RL scheduling literature models and what industrial practice actually demands. This thesis does not close that gap either, in the strict sense: the priority-escalation logic for queue-time-critical lots is implemented in full (Constraint C4, Section **??** of Chapter **??**), but the constraint dictionary it operates on is never populated, since explicit window specifications were not available in the source data; activating it is identified as the highest-priority extension in Chapter **??**.

3.2.4 Dynamic Arrivals and Stochastic Disturbances

Real fab environments never stand still: lots keep arriving, processing times vary, and equipment fails without warning. Many RL papers claim to address this kind of "dynamic" scheduling, but a closer look reveals considerable variation in how seriously that claim is taken. It is common, for instance, to see an episodic formulation built around a fixed set of jobs, when only a genuinely continuous-production simulation actually reflects how a fab behaves; likewise, studies that introduce stochastic processing times or failures often do so by adding noise to an otherwise fixed-size instance rather than modelling a real arrival process. Kuhnle et al. [**?**] and Ghaleb et al. [**?**] stand out as exceptions on this front, training their agents on continuous production with randomised arrivals and failures, and showing that policies learned under this kind of stochastic simulation

transfer reasonably well to held-out scenarios with different arrival rates and failure distributions. This thesis adopts the same continuous-production philosophy and extends it along five distinct axes of randomness, processing-time noise, machine downtime, MES exception injection, hot-lot selection, and scenario sampling, all drawn from empirical rates rather than hand-tuned distributions (Section **??** of Chapter **??**).

Summary: Coverage of JSSP Extensions in the Literature. Table **??** summarizes how studies in the reviewed literature address each JSSP extension. The table makes clear that: (1) most studies address one to two extensions in isolation; (2) no prior study combines all four extensions simultaneously; (3) studies that address multiple extensions typically use simplified simulations rather than real fab data.

Table 3.3: Coverage of JSSP extensions in reviewed RL scheduling literature. ✓ indicates explicit treatment; ∼ indicates implicit or partial treatment.

Study	Reentrant	Setup Times	Queue-Time	Dynamic Arr.	Stochastic
Waschneck et al. [**?**]	✓			∼	∼
Kuhnle et al. [**?**]	✓	✓	∼	✓	✓
Ghaleb et al. [**?**]	✓			✓	✓
Sun et al. [**?**]		∼		✓	✓
Marchesano et al. [**?**]				✓	✓
Zhang et al. [**?**]					
This Thesis	✓	✓	∼	✓	✓

3.3 Action Abstraction: Rule Selection vs. Direct Action

A fundamental design choice in RL-based scheduling is what the agent's action space represents. This choice has profound implications for scalability, interpretability, and compatibility with industrial MES systems.

3.3.1 Direct-Action Policies

In the direct-action approach, the agent learns to pick the specific next lot or operation to dispatch out of all eligible candidates, so its action space is simply the set of lots currently waiting at a given decision point. The appeal is flexibility: unconstrained by predefined heuristics, the agent can in principle discover nontrivial sequencing strategies that no human-designed rule would ever capture. In practice, however, this flexibility comes at a cost that matters a great deal in an industrial setting. Because the number of waiting lots fluctuates over time, the action space itself changes shape, and a network trained under one WIP level often fails to generalise to another; as WIP grows into the hundreds of concurrent lots typical of a real fab, the action space balloons and both learning and inference slow down accordingly. The resulting policy is also opaque, difficult to validate against domain knowledge, and offers no intermediate explanation for why a particular

lot was chosen, which sits uneasily with the traceability and human-approval requirements of real manufacturing systems.

A handful of studies have nonetheless pursued this route. Zhang et al. [**?**], for instance, proposed a GNN-based policy operating on a disjunctive graph, with actions corresponding to individual operations; the approach generalises well to larger instances but remains subject to the variable action space problem described above. Working in an offline setting where the full job set is known in advance, Park et al. [**?**] developed an attention-based architecture for JSSP that achieves strong generalisation results, though that assumption does not hold in real-time dispatching. Closer to the continuous-arrival setting studied here, Marchesano et al. [**?**] applied DQN to single-machine scheduling using action masking to cope with variable queue sizes, but their action space remains tied to specific lots rather than to dispatching rules.

3.3.2 Rule-Selection Policies (Hyper-Heuristics)

The rule-selection approach takes a different view of the problem: rather than choosing individual lots, the agent selects which classical dispatching rule to apply at each decision point, giving a fixed and small action space $\mathscr{A} = \{\text{FIFO}, \text{SPT}, \text{EDD}, \text{LST}, \text{CR}, \ldots\}$. This single design choice resolves most of the practical concerns raised above. Because the agent always picks from the same k rules, training is stable and inference runs in $O(k)$ regardless of WIP; because the rules themselves are interpretable, a policy that selects SPT when queue time is slack and EDD when deadlines tighten can be read and validated by domain experts rather than treated as a black box. Deployment is also simpler in practice, since the classical rules being selected are already implemented in most industrial MES systems, and each one carries well-understood theoretical guarantees, such as SPT's property of minimising mean flow time. The trade-off is that the agent cannot invent dispatching logic beyond what the predefined rules encode, so its performance ceiling is ultimately bounded by the quality of the rule set it is given.

Several studies have adopted this paradigm with encouraging results. In a real industrial fab, Ghaleb et al. [**?**] deployed a DQN-based rule selector and found that even with only three rules available (FIFO, SPT, EDD), the learned selector consistently outperformed any one of them used alone. Zhao et al. [**?**] extended the action set to six rules using PPO on a complex job shop, observing that the agent learns interpretable behaviour such as favouring urgency-based rules under high load. Waschneck et al. [**?**], meanwhile, trained a DQN agent to switch between FIFO and SPT on a simulated wafer fab with reentrant routing. The idea traces back further still to Aydin and Öztemel [**?**], who used tabular Q-learning to choose between SPT, EDD, and FIFO; foundational as that work is, it lacks the function approximation needed to scale to a real fab.

Table **??** lays out the trade-offs between the two paradigms side by side. Given the scalability, real-time inference, and explainability that real wafer fabs demand, rule selection emerges as the more practical choice, and it is the one this thesis adopts, selecting among six dispatching rules defined in Section **??**.

Table 3.4: Comparison of action abstraction approaches for RL-based dispatching. This thesis adopts rule selection with six classical rules.

Dimension	Direct Action	Rule Selection	This Thesis
Action space size	Variable ($\sim n$ lots)	Fixed (k rules)	Fixed (6)
Generalization across scales	Limited	High	High
Inference speed	Slow	Very fast	Very fast
Interpretability	Low	High	High
MES compatibility	Limited	High	High
Maximum flexibility	High	Moderate	Moderate

3.4 State Representations: Encoding Manufacturing State

The effectiveness of an RL agent depends critically on how the system state, whether represented as a vector, an image, or a graph, is passed to the neural network, and manufacturing state is inherently multidimensional: machine availability, queue lengths, tardiness levels, and setup constraints all bear on what counts as a good dispatching decision. How the literature has chosen to encode this state has evolved considerably as the field matured.

3.4.1 Tabular Representations

The earliest RL work in scheduling represented state as a discrete, enumerated space, where each entry corresponds to a specific configuration of the job shop and the value function or policy is simply stored in a lookup table. This approach does not survive contact with realistically sized problems: the state space explodes combinatorially even for small job shops, and nothing learned for one problem size transfers to another. Tabular methods are consequently of largely historical interest today [**?**].

3.4.2 Vector-Based Aggregations

Modern RL scheduling typically uses continuous state vectors encoding aggregate shop-floor statistics. Instead of tracking individual job identities, the state encodes global summary statistics: total WIP, mean queue length, average tardiness, machine utilization, etc.

Table 3.5: State feature dimensionality and categories in reviewed RL scheduling studies.

Study	Dim.	Feature Categories	Approach
Waschneck et al. [**?**]	3–4	Queue length, mean slack	Minimal
Ghaleb et al. [**?**]	8–10	WIP, tardiness, machine load	Aggregated
Kuhnle et al. [**?**]	12–15	WIP, utilization, setup cost proxy	Aggregated
Zhao et al. [**?**]	10–12	Queue stats, urgency distribution	Aggregated
This Thesis	**35**	10 semantic categories	Rich

Vector-based aggregations offer three practical advantages: the state size does not grow with WIP or machine count, features can be named and their influence analyzed, and standard feedforward architectures suffice. The main drawback is that aggregation discards inter-job relationships:

a queue length of 10 could represent ten identical short jobs or two long jobs with very different urgency profiles.

This thesis uses a 35-dimensional feature vector categorized into ten semantic domains (Appendix **??**): (1) lot counts; (2) slack time distribution; (3) waiting time metrics; (4) tardiness and completion metrics; (5) WIP structure; (6) urgency signals; (7) trend features; (8) fab disturbance indicators; (9) setup pressure; and (10) heuristic cost proxies. Two further positions in the vector (indices 9–10) are reserved for calendar features (hour of day, day of week) but are deliberately zeroed at runtime and excluded from this count, since a SHAP importance check found them to contribute negligibly to the policy (Section **??** of Chapter **??**).

This representation is substantially larger than most prior work (Table **??**) and explicitly encodes the multidimensional trade-offs inherent to semiconductor scheduling.

3.4.3 Graph-Based Representations

Recent work encodes the manufacturing system state as a graph, where nodes represent jobs or operations and edges encode precedence, resource, or temporal constraints. Graph Neural Networks (GNNs) then learn policies that propagate information across these relationships.

Zhang et al. [**?**] encode the job shop as a disjunctive graph and train a GNN-based policy that generalizes across problem sizes, while Schulz et al. [**?**] propose a generic graph model for semiconductor fabs with nodes for equipment, operations, and products. Building on this line of work, Liu et al. [**?**] extend GNNs with auxiliary strategies for high-uncertainty settings.

Graph representations offer structural fidelity and enable global reasoning via message passing. Their practical limitations are higher computational cost, important for real-time MES, and the overhead of reconstructing the graph at every decision point. Besides that, no production deployments in real facilities are known.

3.4.4 Visual/Image-Based Representations

A small number of studies encode the shop-floor state as a 2D image, with rows representing machines, columns representing time, and pixel intensity encoding occupancy or queue depth. Convolutional Neural Networks (CNNs) then process these images.

Sun et al. [**?**], for example, used a CNN combined with Dueling DQN and spatial pyramid pooling, whereas Liu et al. [**?**] instead paired an Actor-Critic architecture with a CNN state encoder. While these approaches borrow powerful image-processing tools, manufacturing state is not naturally a 2D grid: encoding it as an image loses semantic meaning, the resulting policies are difficult to inspect, and no industrial deployments are known.

Summary: State Representation Positioning. This thesis uses vector-based aggregation (35 features) rather than tabular, graph-based, or visual approaches. This choice balances two requirements: (1) **Industrial compatibility** (vector inputs integrate easily with standard MES architectures); (2) **Scalability** (state space does not grow with WIP or number of machines).

3.5 Reward Function Design for Dispatching

The reward function directly shapes what the agent learns to optimize. In scheduling, this choice is consequential: a reward that penalizes only tardiness may incur excessive WIP; a reward that penalizes only WIP may miss deadlines.

3.5.1 Single-Objective Rewards

Early RL scheduling studies kept the reward simple, targeting a single metric at a time: makespan minimisation, $r_t = -(\text{max completion time})$, common in offline scheduling but of limited relevance to real-time dispatching; tardiness minimisation, $r_t = -\sum_j \max(C_j - d_j, 0)$; cycle-time minimisation, $r_t = -(\text{mean cycle time})$; or throughput maximisation, $r_t = (\text{number of completions in the period})$. The appeal of this simplicity is short-lived, however, because these objectives conflict with one another in practice. Little's Law alone guarantees that pushing tardiness down by accelerating urgent lots tends to inflate WIP elsewhere in the system, and a purely immediate reward signal gives the agent no reason to anticipate consequences that only materialise several decisions later.

3.5.2 Multi-Objective and Composite Rewards

To address conflicting objectives, modern RL scheduling uses weighted combinations:

$$r_t = w_1 r_{\text{tardiness}} + w_2 r_{\text{cycle time}} + w_3 r_{\text{WIP}} + \dots \tag{3.1}$$

Kuhnle et al. [**?**] combined tardiness, WIP, and machine utilization in a single reward, whereas Zhao et al. [**?**] instead used separate penalties per production target group. Liu et al. [**?**] took a different structural approach, proposing a tree-based reward that combines waiting time and utilization.

3.5.3 Temporal Penalties and Downtime Categories

Section **??** defined a downtime decomposition (idle, setup, hold, abort, offload, rework, maintenance). A well-designed reward should penalize these categories in proportion to their operational impact.

In practice, the literature treats these categories quite unevenly. Setup time is rarely penalised explicitly at all; Kuhnle et al. [**?**] is one of the few exceptions, including a dedicated term $r_{\text{setup}} = -\gamma_{\text{setup}} \times (\text{setup time})$ in the reward. Holds and queue-time violations fare worse still: most studies make no distinction between a lot waiting voluntarily in queue and one forcibly held because a constraint was violated, even though the two have very different operational meanings. Aborts and rework receive a similar level of neglect, with Kuhnle et al. [**?**] again the only study to penalise rework events directly, and maintenance downtime is typically modelled as a random exogenous event rather than something a dispatching decision could exacerbate or mitigate.

3.5.4 Potential-Based Reward Shaping

Potential-based reward shaping [**?**] modifies the reward function by adding a shaping term $F(s,a,s') = \gamma\Phi(s') - \Phi(s)$, where Φ is a potential function. Ng et al. proved that this transformation does not change the optimal policy under the original MDP formalism of Section **??** in Chapter **??**, making shaping provably safe [**?**]. Wirth et al. [**?**] extend the theoretical framework to preference-based RL settings.

This thesis incorporates potential-based shaping in the B4 reward variant (Chapter **??**), adding a term that penalizes decisions whose immediate heuristic cost exceeds that of the greedy-optimal rule in the current state, as described in Section **??**.

Summary: Reward Design in This Work. Table **??** positions this thesis's reward design relative to the literature, using the base B0 formulation as the reference point: among the B0–B5 variants developed in Chapter **??**, B0 is the most comprehensive in the RL scheduling literature, explicitly addressing setup, hold, abort, and rework categories identified in the downtime decomposition of Section **??**. The variants actually carried forward into training and evaluation, including the production B4($w = 0.3$) configuration used throughout Chapter **??**, progressively simplify this base formulation and do not retain all of these terms explicitly (Section **??**); the comparison below should therefore be read as characterising the design space this thesis explored, not the specific reward used to produce the reported results.

Table 3.6: Comparison of reward function components across studies. "E" = explicit term; "I" = implicit/proxy.

Component	**Kuhnle [?]**	**Ghaleb [?]**	**Liu [?]**	**Zhao [?]**	**This**
Tardiness	E	E	E	E	E
WIP / Cycle time	E	I	E	I	E
Setup time	E				E
Hold / Queue-time	I		I		E
Abort / Rework	E				E
Reward shaping					E

3.6 RL Algorithms for Discrete Dispatching

The choice of RL algorithm has a direct bearing on learning stability, sample efficiency, and final performance, and for discrete action spaces such as rule selection, three main families have come to dominate the literature.

3.6.1 Value-Based Methods

Value-based methods learn the action-value function $Q(s,a)$ introduced in Section **??** of Chapter **??** and derive the policy greedily as $\pi(s) = \arg\max_a Q(s,a)$. In scheduling, this family was

applied first by Waschneck et al. [**?**], who used DQN on fab-like environments with reentrant routing, and then by Ghaleb et al. [**?**], who went a step further by deploying DQN in a real industrial fab; both report stable convergence under the standard DQN framework. The family has since accumulated a set of well-known refinements, Double DQN to reduce overestimation bias, Dueling DQN to separate the advantage and value streams, and prioritised experience replay to focus learning on informative transitions, but a persistent weakness remains common to all of them: ε-greedy exploration is a blunt instrument, and the early overestimation of $\max_a Q(s,a)$ tends to slow convergence regardless of which refinement is used.

3.6.2 Policy Gradient and Actor-Critic Methods

Policy-based methods directly parameterize $\pi_\theta(a|s)$ and optimize it via the policy gradient theorem introduced in Section **??** of Chapter **??**. The simplest, highest-variance instance of that theorem replaces the action-value term $Q^{\pi_\theta}(s,a)$ with the Monte-Carlo return G_t sampled from a complete episode, since $\mathbb{E}[G_t \mid s_t, a_t] = Q^{\pi_\theta}(s_t, a_t)$:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s)\, G_t].$$

Actor-Critic methods combine a policy (actor) with a value function (critic), where the critic estimates the advantage $A(s,a) = Q(s,a) - V(s)$, the same variance-reduction quantity derived in Section **??** of Chapter **??**. Kuhnle et al. [**?**] applied A3C to semiconductor scheduling and reported faster convergence than DQN; Liu et al. [**?**] paired A3C with a CNN state encoder.

3.6.3 Proximal Policy Optimization (PPO)

PPO [**?**] is a modern policy gradient algorithm that addresses instability through a clipped surrogate objective (detailed in Section **??** of Chapter **??**):

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)]\,,$$

where $r_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$.

Clipping prevents catastrophic performance degradation, multiple gradient epochs per data batch improve sample efficiency, and the algorithm is relatively robust to hyperparameter choices, making it the modern standard for discrete scheduling.

In a comprehensive comparison of PPO, DQN, and A3C for semiconductor scheduling, Kuhnle et al. [**?**, **?**] found PPO the most robust of the three. Zhao et al. [**?**] likewise used PPO for rule selection, obtaining interpretable policies that adapt to different load levels. This thesis follows the same choice, combining PPO with GAE and entropy regularization (Section **??** of Chapter **??**).

Algorithm Selection. Table **??** compares the main RL algorithms used in scheduling literature. PPO emerges as the preferred algorithm for this application due to its stability, sample efficiency, and proven track record in industrial scheduling settings.

Table 3.7: Comparison of RL algorithms for discrete dispatching action spaces, with representative studies applying each algorithm to scheduling or dispatching problems.

Algorithm	**Convergence**	**Sample Eff.**	**Stability**	**Implementation**	**Representative Studies**
DQN	Slow	Medium	Medium	Standard	Waschneck et al. [**?**]; Ghaleb et al. [**?**]; Aydin and Öztemel [**?**]
Double DQN	Medium	Medium	Good	Standard	Improvement reported within Waschneck et al. [**?**] and Ghaleb et al. [**?**] as a standard DQN extension
A3C	Fast	Good	Medium	Parallel	Kuhnle et al. [**?**]; Liu et al. [**?**]
PPO	Fast	Good	Excellent	Standard	Schulman et al. [**?**]; Kuhnle et al. [**?**, **?**]; Zhao et al. [**?**]; **this thesis**

3.7 Research Gaps and Positioning of This Thesis

The foregoing sections have surveyed the literature on RL-based scheduling, mapping empirical studies onto the industrial JSSP extensions and methodological choices. This section synthesizes these findings to identify gaps and position the thesis relative to the state of the art.

A recurring limitation across this literature is the reliance on synthetic or simplified simulation environments: most evaluated fabs contain fewer than 50 machines, use homogeneous product mixes, and rely on Poisson arrival processes calibrated to demonstrate algorithmic properties rather than represent operational complexity. Under such conditions, RL agents can learn near-optimal policies that do not transfer to real fabs where routing heterogeneity, sparse machine qualification matrices, and irregular demand create distributions that synthetic training environments cannot capture. This systematic gap between simulation-reported performance and real-world deployability motivates the use of real MES event data in the present work.

3.7.1 Identified Research Gaps

1. **Limited treatment of JSSP extensions in combination.** Most RL studies address 1–2 industrial constraints in isolation. No prior work systematically combines reentrant flows, sequence-dependent setup times, queue-time constraints, and dynamic stochastic arrivals in a single RL framework. Table **??** documents this gap.

2. **State representations under-exploit manufacturing structure.** Studies reviewed use 3–15 dimensional feature vectors (Table **??**). The 35-dimensional state used in this thesis is substantially richer and explicitly encodes WIP structure, urgency distributions, and heuristic cost proxies that prior work compresses or omits.

3. **Reward design does not account for downtime decomposition.** Prior RL studies either ignore the downtime decomposition or address it implicitly. This thesis explicitly penalizes setup, hold, abort, and rework as distinct reward terms, aligned with the industrial reality of fab downtime (Section **??**, Chapter **??**).

4. **Queue-time constraints treated as soft penalties, not hard constraints.** No reviewed RL study enforces queue-time windows as explicit constraints. This thesis implements the queue-time tracking and priority escalation mechanism in the simulator — specifically, the `queue_time_constraints` dictionary and critical-lot dispatch logic (Assumption 6, Chapter **??**) — but the constraints are not populated in the current experiments, as explicit queue-time window specifications were not available in the source data. Hold event penalties ($-\gamma_{\text{hold}}\,\Delta H_t$) in the reward function partially capture hold costs that arise from constraint-related interventions in the real fab at empirically calibrated rates. Full QTC enforcement remains the highest-priority future extension (Section **??**, Chapter **??**).

5. **Lack of real fab parameterization.** Most RL scheduling studies use simplified simulations or academic benchmarks. This thesis is parameterized from 11.4 million MES events from a real semiconductor facility, with realistic processing times, routing structures, equipment failures, and qualification constraints.

6. **No systematic evaluation of generalization across operating regimes.** None of the reviewed studies systematically evaluates how policies generalize across different operating regimes. This thesis explicitly addresses regime-dependent behaviour through scenario-variant training and evaluation (Section **??**).

7. **Limited work on policy interpretability and extraction.** RL policies are typically opaque. This thesis addresses interpretability through scenario-conditional rule-selection analysis: the learned policy is characterised by which dispatching rules it selects under each operating condition, producing a human-readable preference profile that aligns with classical scheduling theory (Chapter **??**). Policy distillation into interpretable decision trees is identified as a further extension (Section **??**).

3.7.2 Positioning of This Thesis

Table **??** summarizes how this thesis positions relative to the principal categories of prior work.

Queue-time constraint tracking is implemented in the simulation engine but is inactive in the current experiments due to the absence of explicit window specifications in the available dataset (see Chapter **??**, Assumption 6).

Table 3.8: Positioning of this thesis relative to the state of the art in RL-based dispatching.

Dimension	**Static Rules**	**Hybrid ML**	**RL Direct**	**RL Rule-Sel.**	**This**
Adaptive to state	No	Partial	Yes	Yes	Yes
Real-time capable	Yes	No	Limited	Yes	Yes
Interpretable	Yes	Partial	No	Partial	Yes
MES compatible	Yes	Limited	Limited	Yes	Yes
Reentrant routing	N/A	Limited	Some	Some	Explicit
Setup times	Yes	Limited	Implicit	Some	Explicit
Queue-time const.	Yes	Limited	No	No	Partial
Fab scale (600+ m.)	Yes	No	No	No	Yes
Rich state	N/A	$\sim$5D	$\sim$15D	$\sim$10D	35D
Multi-scenario gen.	No	No	Limited	No	Explicit
Real MES data	N/A	Limited	Limited	Limited	11.4M

This thesis distinguishes itself on six dimensions: (1) comprehensive treatment of all four industrial constraints simultaneously; (2) parameterization from 11.4 million real MES events from a semiconductor facility; (3) a 35-dimensional feature vector substantially more detailed than prior work; (4) principled multi-term reward design explicitly addressing the full downtime decomposition; (5) a multi-scenario training framework combining parameter scaling (Catalogue A) and disturbance injection (Catalogue B); and (6) a structured evaluation of regime-dependent behaviour across ten scenario variants.

It is worth being precise about where the novelty of this work actually lies. The rule-selection paradigm itself is not new: PPO-based hyper-heuristic dispatchers have been demonstrated before (Section **??**), and the present work does not claim an algorithmic advance over them. Its genuine contribution is twofold. The first is the parameterisation of the entire pipeline from 11.4 million real MES events of an operating facility (dimension 2 above), at a fab scale, 622 machines and 281 flows, well beyond the simplified testbeds that dominate the literature; this is what makes the reported gains credible rather than artefacts of a synthetic environment. The second is the interpretability of the resulting policy: rather than reporting performance alone, the analysis characterises *which* rule the agent selects under each operating condition and shows that these choices recover the predictions of classical scheduling theory (Chapter **??**). The remaining dimensions, a richer state vector, a fuller reward decomposition, and a multi-scenario evaluation protocol, are best understood as careful engineering that supports these two contributions rather than as independent novelties. Framed this way, the thesis sits at the intersection of academic rigour and industrial applicability, advancing the credibility and transparency of RL-based scheduling rather than its underlying algorithms.

Chapter 4

Methodological Framework

Building on the research gaps identified in Section **??** of Chapter **??**, most notably the limited treatment of JSSP extensions in combination, under-exploited state representations, and reward designs that ignore downtime decomposition, this chapter presents the complete methodological framework for applying deep reinforcement learning to dispatching in semiconductor wafer fabrication. The chapter opens with an architectural overview that motivates the design choices adopted throughout (Section **??**) and states the modelling assumptions and operational constraints that bound the simulation's fidelity (Section **??**). It then systematically describes each major component: the data processing pipeline that transforms raw production records into simulation-ready artefacts (Section **??**), the discrete-event simulation engine that replicates the fab's operating dynamics (Section **??**), the two-stage dispatching logic that translates RL decisions into concrete lot–machine assignments (Section **??**), the Partially Observable Markov Decision Process formulation that frames the learning problem (Section **??**), the multi-term reward function that shapes the learning objective (Section **??**), the RL training architecture that discovers context-dependent rule selection policies (Section **??**), and finally the six classical dispatching rules that serve as both the agent's discrete action repertoire and its fixed-policy baselines (Section **??**). Throughout, design decisions are grounded in the operational constraints and data characteristics of a real semiconductor fab, with key components rigorously validated against historical production records.

4.1 Architectural Overview

The reinforcement learning meta-dispatcher is built from four tightly integrated layers, each serving a distinct purpose in progressively transforming raw production data into a trained policy capable of real-time rule selection. Figure **??** illustrates this layered architecture. The remainder of this section discusses each layer and the design philosophy underpinning the whole system.

Layered design rationale Learning a dispatching policy for semiconductor manufacturing is fundamentally a data-to-policy problem. Raw Manufacturing Execution System (MES) records,

the event logs the fab's own tracking software generates for every lot, machine, and recipe interaction, millions of events recorded across five months of operation, consistently contain the system's operating signatures, showing which rules work when, which machines matter most, and which disturbances occur most frequently. These records cannot be fed directly to an RL agent, however, since the agent requires a clean, stochastic environment that replicates the real fab's dynamics while remaining computationally tractable. This is why three intermediate layers are systematically interposed between raw data and the trained policy: data cleaning and artifact production, a simulation engine parameterized by those artifacts, and an RL training loop that learns from simulated experience. This design philosophy prioritizes *interpretability*, *reproducibility*, and *fidelity* over raw end-to-end optimization.

What crosses each layer boundary. The layers in Figure **??** are not independent stages that merely happen in sequence: each boundary between them is a specific, narrow interface that determines what one layer can and cannot know about the ones before it. Between Data and Pipeline, the interface is the four raw tables themselves (Section **??**); nothing about the simulator or the agent leaks upstream into how those tables are cleaned. Between Pipeline and Simulation, the interface is a single `components` dictionary, built once by `build_simulator_components()` and passed to `create_simulator_instance()` on every `reset()`, a function that deep-copies the lot and machine objects it contains into a fresh `FlowSimulator`, so that one episode's state can never leak into the next (Appendix **??**); the simulator never re-reads the raw MES tables directly; it only ever sees this dictionary. Between Simulation and Learning, the interface is the pair (o_t, r_t) that `MetaSelectorEnv` extracts from the simulator's internal state at each decision epoch (Section **??**); the agent has no access to the simulator's true state s_t, only to this fixed-width projection. Finally, the loop closes through the `Dispatcher`: the agent's discrete action a_t is translated into a named rule, the rule commits a concrete lot–machine assignment, and that assignment's consequences propagate forward as changes to the simulator's state that the next observation will reflect (Section **??**). Treating these four handoffs as the object of study, rather than the four boxes themselves, is what makes each layer replaceable in isolation, for instance swapping the simulator for a higher-fidelity one, without touching the other three.

Scope and out-of-scope. This work addresses lot-level dispatching within a single frontend fab, using a Proximal Policy Optimization (PPO) agent, the reinforcement learning algorithm detailed in Section **??**, that selects among classical dispatching heuristics at each decision epoch rather than replacing the existing MES rule infrastructure. Production data spanning five months (January–May 2025) are used for training, validation, and testing, with full statistics reported in Chapter **??**. The work covers the following aspects:

- **Dispatching granularity**: Rule selection at each machine-availability event. The agent controls the priority criterion, not individual lot–machine assignments.

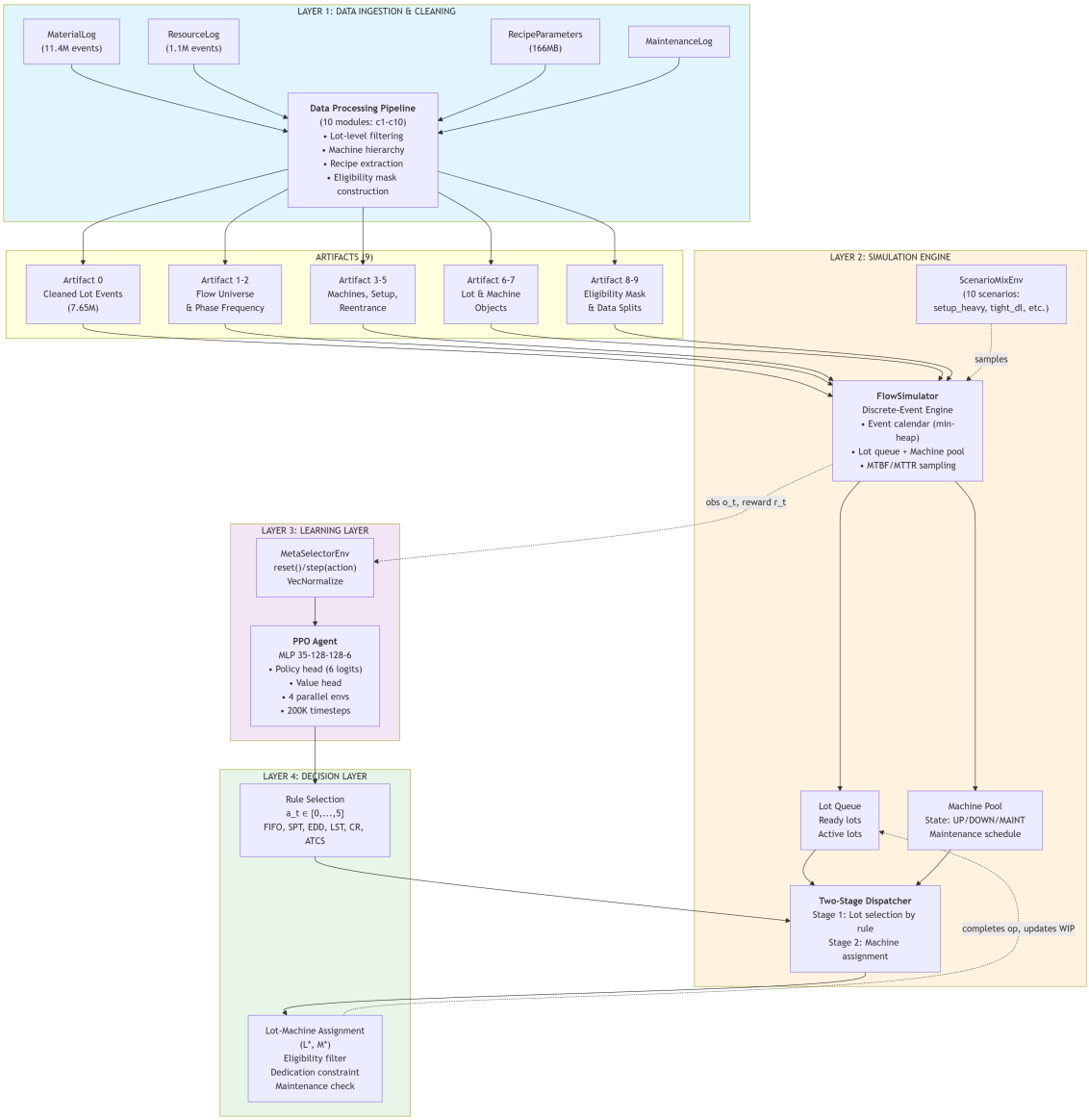

Figure 4.1: Four-layer architecture of the RL meta-dispatcher. The Data layer's four raw MES tables are consumed by the nine-stage pipeline, which writes its artefacts to disk under their real filenames (Section **??**). Those artefacts are loaded into a single `components` dictionary that parameterises a fresh `FlowSimulator` instance at the start of every episode. `MetaSelectorEnv` wraps that simulator and exchanges an observation o_t and reward r_t with the PPO agent at every decision epoch; the agent's action a_t closes the loop by naming the dispatching rule the `Dispatcher` applies to commit the next lot–machine assignment, whose outcome (completed operations, updated WIP) shapes the next observation. Each interface is detailed where the corresponding layer is discussed.

- **Operational constraints**: Machine eligibility masks, sequence-dependent setup times, equipment dedication for critical lithographic layers, and preventive and corrective maintenance.
- **Objective**: Composite minimization of cycle time, work-in-progress (WIP, the count of lots simultaneously in process across the fab) deviation, on-time delivery shortfall, and MES exception penalties, namely aborts, reworks, holds, and off-flows.
- **Evaluation protocol**: Temporal train, validation, and test split. The test month is never observed during training.
- **Out of scope**: Inter-fab coordination, operator scheduling, lot release planning, batch scheduling, and multi-site optimization.

4.2 Modelling Assumptions and Constraints

4.2.1 Simplifying Modelling Assumptions

The simulation model incorporates structural simplifications that bound its fidelity relative to the physical production system. These assumptions are stated explicitly so that their impact on the experimental conclusions can be assessed.

1. **No inter-bay transport.** Lots advance instantaneously from one machine queue to the next upon step completion. Transport times are typically one to two orders of magnitude smaller than processing times in the facility and do not materially affect dispatching-level analysis.
2. **No operator constraints.** Operator availability, shift schedules, and manual handling delays are not represented. All machine operations proceed autonomously once a lot is assigned.
3. **Single-lot processing.** Each machine processes one lot at a time. Batch multi-lot processing is not modelled, and the machines identified as batch-capable are treated as single-server resources.
4. **No lot splitting or merging.** Each lot is an indivisible entity throughout its flow.
5. **Atomic processing steps.** Once a lot begins an operation, it runs to completion without preemption. Stochastic failures occur between steps, not during them.
6. **Queue-time constraint mechanism implemented but inactive.** The simulation engine tracks queue-time constraints and implements priority escalation logic (Section **??**). Constraints are not populated from the current dataset, however, since explicit queue-time window specifications were not available in the source data.

These simplifications consistently share a common direction, each removing a source of waiting or blocking from the model, so the simulator systematically under-represents delay rather than inventing it. Instantaneous transport and the absence of operator and queue-time holds are the dominant contributors, and their combined effect is bounded empirically. These two omissions are distinct from the MES exception holds described in Section **??**, which remain fully active in the simulation and are injected stochastically from empirical (flow, step)-level rates. Operator and queue-time holds are absent from the model, while historical MES exception holds are present and enforced identically across every policy evaluated. The fidelity comparison in Section **??** (Table **??**) shows simulated cycle time and utilisation within 2% of the historical values, with tardiness under-estimated by roughly 13%. Crucially, because every dispatching policy is evaluated under the same assumptions, this bias is shared across the agent and all baselines and therefore does not affect the relative comparisons that the evaluation is built on. It bounds only the absolute realism of the reported figures. Single-lot processing and the absence of splitting leave a small amount of throughput on the table, since batching is shallow in this work area, while atomicity is a standard non-preemption assumption with negligible effect at the decision granularity studied here.

4.2.2 Operational Constraints

The following constraints govern the dispatching system and are formally referenced in the Partially Observable Markov Decision Process (POMDP) formulation introduced in Section **??**.

1. **C1 (Single-server).** Each machine processes at most one lot at a time. Process chambers are present in the machine object model for two tools but are not activated in the current simulation, so the dispatcher treats all machines as single-slot resources.

2. **C2 (Eligibility).** A lot may only be processed on machines in $\mathscr{E}(\text{flow}_j, \text{step}_j)$, the eligibility mask derived from historical machine assignments in MaterialLog (Section **??**).

3. **C3 (Sequential flow with re-entrance).** Steps within a lot must be executed in the order defined by its product flow. Re-entrant visits to the same workstation at different step indices are permitted and common.

4. **C4 (Queue-time constraints).** Consecutive steps subject to a queue-time constraint must be processed within the specified window, or a hold or rework is triggered. This constraint is modelled but inactive in the current experiments, per assumption 6 above.

5. **C5 (Machine availability).** Machines may be unavailable due to preventive maintenance or corrective repair, both represented by the same `DOWN` state value. A machine currently `DOWN` is excluded from all dispatching candidates.

6. **C6 (Equipment dedication).** Critical lithographic layers must be processed on the same machine across all visits, to preserve overlay alignment. The dispatcher restricts the eligible

set to the historically dedicated machine, identified by backward search through the lot's dispatch history.

4.3 Data Processing Pipeline: From Raw Logs to Simulation Artifacts

Before the RL agent can learn a dispatching policy, the production system must be faithfully represented in simulation. This begins with data: millions of raw MES event records that capture every decision, delay, and disruption over five months of real operation. However, raw data is typically messy, redundant, and unstructured. The data processing pipeline, implemented in `src/pipeline/` and orchestrated by a single function, `run_all_in_one_pipeline()`, systematically cleans, validates, and transforms these raw records into the artefacts the simulator consumes. This section explains the pipeline's design, separates it from earlier exploratory analysis that shaped some design choices but is not part of the reproducible pipeline, and walks through the nine stages the orchestrator actually executes and the artefacts each one writes to disk. Appendix **??** complements this section with material not repeated here: column-level descriptions of each raw source table, the exact row-level cleaning filters applied at each step, per-stage row-count and validation statistics, and the full tables underlying the exploratory flow-universe, phase-frequency, and reentrance analysis introduced in Section **??**.

4.3.1 Pipeline Design

The raw MES system records over 11 million events for 173,227 distinct lots flowing through 1,114 product flows. However, not all of these events are directly relevant to the dispatching problem, and not all data is clean. Wafers are sorted, split, and recombined. Container-level logistics events, marked by flow names starting with square brackets such as `[Sort]` or `[Merge]`, must be excluded. Equipment failures, maintenance windows, and recipe changes leave traces scattered across multiple tables (MaterialLog, ResourceLog, RecipeParameters, MaintenanceLog) that must be reconstructed and cross-validated. A single MES termination event is sometimes replicated across multiple area codes for bookkeeping purposes, creating duplicates that must be collapsed.

A note on `CandidateResources.csv`. These four tables are the only genuinely independent, externally sourced inputs to the pipeline. Two further artefacts are occasionally described elsewhere as auxiliary source tables but are not independent MES exports: `CandidateResources.csv`, produced by Stage 8 (Section **??**), is a reformatted view of the machine-assignment history already present in MaterialLog, not a second, cross-validating source, and `ResourceSubResources.csv` is not consumed anywhere in the pipeline that produces the reported results; the machine hierarchy is instead built directly from the `ParentResourcesCount` column of ResourceLog (Stage 2, Section **??**). Both distinctions

matter for anyone trying to verify the eligibility mask or the machine hierarchy against an independent MES source, since no such independent cross-check exists for either.

Rather than hard-coding these cleanup steps into the simulator, which would couple data quality to simulation logic, the pipeline decomposes the process into a sequence of independent, testable stages that progressively filter, validate, and enrich the raw data, producing a set of serialized artefacts that the simulator consumes. This design offers three advantages. Each stage can be validated independently against known historical ground truth, which supports *testability*. The pipeline is also deterministic given fixed random seeds, so policy checkpoints can be loaded and evaluated against bitwise-identical lot populations used during training, giving *reproducibility*. Finally, the separation of concerns keeps data logic out of the simulation engine, so changes to data processing do not require re-tuning RL hyperparameters, giving *clarity* to the overall design.

4.3.2 Exploratory Analysis Behind Early Design Choices

Several empirical facts referenced elsewhere in this chapter were established through standalone exploratory analysis conducted early in the project, not by the reproducible pipeline described in Section **??** below. These include: the flow universe of 1,114 distinct product flows and the identification of a smaller set of flows active and stable enough to warrant automated scheduling; the phase-frequency breakdown of step activity (metrology accounting for 20.6% of events, deposition and lithography together for 25.6%, and sort or container handling for 33.6%) that motivated the fab-disturbance features of Section **??**; and the reentrance statistics (71.2% of lots revisit at least one step root, but only 4.7% of (flow, step-root) pairs show median revisits above two) that justified aggregating reentry into fab-level features (machine load coefficient of variation, machines down, hot-lot fraction) rather than encoding per-step reentry counters.

Unlike the nine stages in Section **??**, this analysis is not re-executed by `run_all_in_one_pipeline()`, and its numeric outputs are not written to any artefact the simulator reads at runtime. One of its conclusions, a whitelist of flows considered stable enough for simulation, is instead carried forward as a fixed constant (`HEALTHY_FLOWS`) inside the simulator-building code rather than being recomputed from a dedicated pipeline stage. The percentages above should therefore be read as the empirical basis for design decisions made once, early in the project, rather than as figures independently reproducible by re-running the pipeline described below.

4.3.3 The Nine Pipeline Stages and Their Artefacts

The nine stages below are exactly the calls made, in order, by `run_all_in_one_pipeline()`. As in the exploratory analysis above, execution order and data dependency are not the same thing: a stage runs where it does because its output is needed by a later stage, not necessarily because it depends on the stage immediately before it. Figure **??** shows the nine stages in execution order, each paired with the artefact it writes to disk.

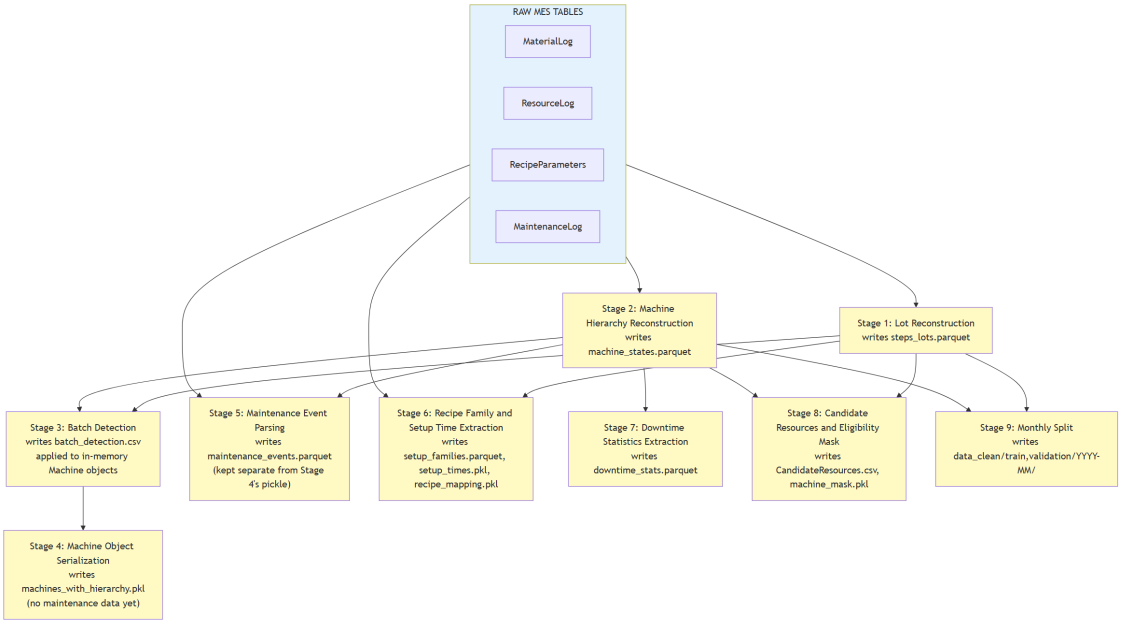

Figure 4.2: Data processing pipeline: execution order of the nine stages orchestrated by `run_all_in_one_pipeline()`, each paired with the artefact it writes to disk. The vertical order is the order of execution, not necessarily the order of data dependency, as detailed in the text.

4.3.3.1 Stage 1: Lot Reconstruction

The function `extract_material_events()` ingests the raw MaterialLog, containing 11.4 million events across 39 columns, and filters it to the lot-level events relevant to dispatching. The function `build_lots_and_wafers()` then reconstructs each lot's complete processing history by iterating over events in chronological order per lot, matching `TrackIn` events to `Terminate` events to compute per-step processing durations. Cycle time is decomposed into processing time, queue time, and overhead (holds, reworks, off-flows, aborts), and processing-time outliers are clipped to their per-(flow, step) median. The function `remove_short_wafers_and_empty_lots()` then discards degenerate lots with fewer than three steps, and `generate_parquet_from_lots()` flattens the surviving `Lot` objects into a single tabular artefact, **`steps_lots.parquet`**, spanning 7,653,932 cleaned lot-level events across 173,227 distinct lots and 1,114 flows. A further processing step produces `steps_lots_clean.parquet`, the file `build_simulator_components()` actually loads at runtime (Appendix **??**). Either way, this flat table, not a serialized collection of `Lot` objects, is what every later stage means by "the lot data." The `Lot` objects themselves (with their exception counters, `detected_aborts`, `detected_reworks`, `detected_temp_offflows`, defined in `src/envs/lot.py`) are held only in memory for the remainder of the pipeline run and are never serialized in full; the sole on-disk trace of the object collection is a ten-lot text sample written by `save_lot_paths_txt()` at the very end of the run, for spot-checking rather than reproduction.

4.3.3.2 Stage 2: Machine Hierarchy Reconstruction

The function `extract_resource_events()` ingests the raw ResourceLog, and `build_machines_with_hierarchy()` reconstructs the three-level equipment hierarchy it encodes: 622 main machines (`ParentResourcesCount = 0`) and 1,301 subresources, including 7 process chambers and 463 components. This stage writes a tabular summary, `machine_states.parquet` (via `generate_parquet_from_machines()`), and returns the live `Machine` objects in memory for the two stages that follow.

4.3.3.3 Stage 3: Batch Detection

The function `detect_batch_processing_from_logs()` reverse-engineers batch-processing behaviour from co-temporal `TrackIn` events in `steps_lots.parquet`, writing **`batch_detection.csv`**. Its output is applied directly onto the in-memory `Machine` objects from Stage 2 (`batch_capacity`, `processing_mode`, `allow_mixed_recipes`), before those objects are serialized in Stage 4; there is no separate reentrance-detection step, despite the name sometimes used for this stage elsewhere.

4.3.3.4 Stage 4: Machine Object Serialization

The enriched `Machine` objects from Stages 2 and 3, hardware layer, recipe and setup layer, policy layer, state layer (`UP`/`DOWN`, with scheduled maintenance and unscheduled failures both represented by `DOWN`), are pickled directly to **`machines_with_hierarchy.pkl`**. Maintenance schedules are *not* part of this pickle, despite encapsulating scheduling-relevant machine state elsewhere in this chapter: they are parsed and saved separately in Stage 5 and never merged back into these objects.

4.3.3.5 Stage 5: Maintenance Event Parsing

The function `parse_maintenance_log()` reads the raw MaintenanceLog and classifies events into predictable (scheduled, 71.6%) and adhoc (corrective, 28.4%), with a median downtime duration of 1.79 hours, writing **`maintenance_events.parquet`**. This stage is skipped entirely if no maintenance log path is supplied.

4.3.3.6 Stage 6: Recipe Family and Setup Time Extraction

This stage is implemented by three functions with different inputs, not one. The functions `parse_recipe_parameters()` and `build_setup_families()` extract a recipe-family taxonomy from the raw RecipeParameters table (`ParameterType == 'RecipeAttr'`), yielding 8,232 distinct families and writing **`setup_families.parquet`**. Separately, `calculate_setup_times_from_materiallog()` performs the actual gap-scan that produces usable setup durations: it scans `steps_lots.parquet` (Stage 1's artefact, not RecipeParameters) for consecutive

operations on the same machine, identifying inter-operation gaps in the range $(0, 4\text{h}]$ as plausible setup intervals, and writes **`setup_times.pkl`**, containing 1,526 non-trivial `(machine, from-family, to-family)` triplets with a median setup time of 1.01 hours. A third function, `build_recipe_mapping()`, combines both inputs to write **`recipe_mapping.pkl`**. Only `setup_times.pkl` feeds the dispatcher's setup-time cascade (Section **??**); the recipe-family taxonomy from RecipeParameters plays a supporting, not primary, role.

4.3.3.7 Stage 7: Downtime Statistics Extraction

The function `extract_downtime_from_resourcelog()` derives per-machine failure and repair statistics from the raw ResourceLog events already loaded in Stage 2, specifically the Mean Time Between Failures (MTBF, how long a machine typically runs before an unscheduled breakdown) and the Mean Time To Repair (MTTR, how long that breakdown typically takes to fix), writing **`downtime_stats.parquet`**. This stage is wrapped in exception handling and silently skipped if it fails, since it is not required for the pipeline's other outputs.

4.3.3.8 Stage 8: Candidate Resources and Eligibility Mask

The function `build_candidate_resources()` constructs an eligibility set of machines for each (flow, step) pair directly from `steps_lots.parquet` (Stage 1), taking the union, per lot, of the initial and secondary machine fields recorded at each step, and writes **`CandidateResources.csv`**. This eligibility set is derived entirely from MaterialLog machine-assignment history and is not cross-validated against an independent MES qualification table; `CandidateResources.csv` is a reformatted view of that same history, not a second data source. The function `build_and_save_machine_mask()` then combines it with the machine hierarchy from Stage 2 to write the mask the simulator actually uses, **`machine_mask.pkl`** (and a parquet mirror). After filtering to the 281 flows in the simulation routing dataset, the active mask reduces to 1,539 entries. As in the full union (72.0% single-eligible-machine share, Appendix **??**), the large majority of (flow, step) pairs in this filtered mask have exactly one eligible machine, meaning dispatching rule selection has no effect on machine assignment for most steps. Critically, the mask's coverage against historical assignments is verified at build time, though by construction this check is closer to an internal consistency test than an external validation, since both sides derive from the same source.

4.3.3.9 Stage 9: Monthly Split

The function `split_data_by_month()` partitions `steps_lots.parquet` chronologically into per-month directories under `data_clean/`. The function itself produces only two named buckets, `train` and `validation`, assigning the first 80% of available months to `train` and the remainder to `validation` by index, not three explicitly named pickles keyed to specific months. Where this chapter and Chapter **??** report a three-way split (train: January 2025, validation: April 2025, test: May 2025, held out because it combines the highest flow diversity and machine count

of the five months), that assignment is applied at the point the month and split are selected for a given run, not produced as a standing artefact by this stage.

4.3.4 Summary: The Pipeline's Artefacts

Table **??** lists the artefacts written to disk by the nine stages above, by their actual filenames. The pipeline is deterministic given fixed random seeds, enabling exact reproduction of these artefacts from the raw source tables.

Table 4.1: Artefacts written to disk by the data processing pipeline, by real filename.

Artefact (file)	Description	Stage
`steps_lots.parquet`	7.65M events, 173K lots, flattened lot-step table	1
`machine_states.parquet`	622 machines, 1,301 subresources, tabular summary	2
`batch_detection.csv`	Reverse-engineered batch capacity per machine	3
`machines_with_hierarchy.pkl`	Machine objects, hierarchy and batch flags, no maintenance	4
`maintenance_events.parquet`	Predictable vs. adhoc maintenance events, kept separate	5
`setup_families.parquet`	Recipe-family taxonomy from RecipeParameters	6
`setup_times.pkl`	1,526 setup-time triplets, from lot-data gaps	6
`recipe_mapping.pkl`	(material, step) $\rightarrow$ setup-family mapping	6
`downtime_stats.parquet`	Per-machine MTBF/MTTR statistics	7
`CandidateResources.csv`	Reformatted lot-history view, not an independent source	8
`machine_mask.pkl`	1,539 (flow, step, machine) eligibility entries	8
`data_clean/\{train, validation\}/YYYY-MM/`	Per-month partitions, two buckets only	9

4.4 Simulation Engine: Discrete-Event Simulation with Entity Models

With the pipeline's artefacts in hand, this section describes the FlowSimulator discrete-event simulation engine that consumes these artifacts and replicates the fab's operating dynamics. Unlike simulators parameterized by synthetic instances (e.g., Mason instances of [**?**] or Mönch instances of [**?**]), FlowSimulator is built from real production data yet remains computationally efficient for RL training. This section explains the simulation architecture, the two core entity models (Lot and Machine), the event loop, and validation against historical ground truth. Appendix **??** complements this section with material not repeated here: the full MES operation taxonomy with historical event frequencies (Table **??**), the LightGBM processing-time model's cross-validated accuracy, and the episode initialisation and warm-up policy.

4.4.1 Simulation Architecture

FlowSimulator was implemented entirely from scratch in Python rather than built on a generic simulation library such as SimPy, for two reasons. First, it integrates directly with Gymnasium, the standard Python API for reinforcement learning environments (a reset() method that starts a new episode and returns the first observation, and a step(action) method that applies one action and returns the next observation, reward, and termination flag), allowing a natural reset and step loop where the agent observes fab state, selects a dispatching rule, and receives a reward signal. Second, custom code makes it considerably easier to represent facility-specific behaviour, such as MES exception replay, queue-time constraint tracking, and equipment dedication for overlay-critical layers. Machine sub-resources such as process chambers notably exist in the object model but are not activated in the current simulation, since the dispatcher treats every machine as a single-slot resource (Constraint C1, Section **??**). Besides that, the implementation favours clarity over raw computational speed. For the RL training budgets used here, close to 1,800 episodes of about 1,700–2,000 steps each, a CPU-based simulator is sufficient.

Discrete-event simulation fits this problem naturally. This specific type of simulation, data-driven and event-triggered, models a production system that only changes state at discrete instants, for instance when a lot completes, a maintenance window opens or closes, or a machine fails. Advancing the clock directly from event to event, rather than stepping through fixed timesteps, lets the simulator skip idle intervals entirely and stay computationally efficient. It also captures the fab's naturally stochastic behaviour, since empirical failure distributions drive equipment downtime rather than a predetermined schedule. Fixed-timestep simulation would instead waste computation stepping through idle intervals, and analytical methods struggle to capture the fab's nonlinear dynamics.

4.4.2 Core Entity: Lot

A Lot is the fundamental unit of dispatching. It corresponds to a physical lot of wafers that must complete an ordered sequence of manufacturing steps. Each lot object, reconstructed during the data pipeline (Stage 1, Section **??**), stores its complete processing history and tracks progress through a step index. At simulation time, a lot is considered *dispatchable* when its ready timestamp does not exceed the simulation clock and it passes the dispatchability check:

$$\texttt{can\_dispatch}(L_j) = \neg L_j.\texttt{on\_hold} \wedge \neg L_j.\texttt{is\_terminated} \tag{4.1}$$

Lots that are on hold (pending quality review or blocked by a prior abort) or that have been permanently terminated are filtered from the candidate set before any dispatching rule is applied.

Beyond these binary flags, each lot maintains an equipment dedication map that records, for every step type subject to a dedication constraint (principally critical lithographic layers), the machine on which that step was first processed. When the same step type recurs later in the flow, the dispatcher performs a backward search through the lot's processing history to identify the

dedicated machine and restricts the eligible machine set to that single resource. This mechanism reliably ensures overlay alignment across critical layer re-visits.

4.4.3 Core Entity: Machine

A Machine object encapsulates all information relevant to scheduling and resource availability. Its internal structure is organized into five conceptual layers:

1. **Hardware layer**: Process chambers and components. Process chambers are present in the object model for two machines (BONDER-1, WETSTRIP-1, anonymized machine identifiers; see Appendix **??**) but modeled as single-slot resources in the current simulation.
2. **Recipe and setup layer**: The current recipe family, the identity of the last processed product, and the setup time matrix entry covering the transition from the current family to any potential next family.
3. **Policy layer**: Equipment dedication constraints and batch capacity configuration, which modify the machine selection logic during dispatching.
4. **State layer**: The operational state (`UP`, `DOWN`), the earliest time at which the machine will be free (`available_at`), and cumulative downtime statistics.
5. **Maintenance layer**: The ordered list of upcoming maintenance events (both predictable and adhoc), together with an index pointer to the next scheduled event.

Figure **??** illustrates these five layers. The machine exposes a two-value state flag, `state` $\in \{$`UP`,`DOWN`$\}$, governing availability. Scheduled maintenance and unscheduled stochastic failures are both represented by the same `DOWN` value, not by separate states, and are distinguished only in the maintenance and downtime logs kept for reporting:

- `UP` $\rightarrow$ `DOWN` (scheduled maintenance): triggered by a predictive maintenance event dequeued from the maintenance heap.
- `DOWN` $\rightarrow$ `UP` (scheduled maintenance): triggered at the scheduled maintenance end time.
- `UP` $\rightarrow$ `DOWN` (unscheduled failure): triggered by a stochastic failure event sampled from the empirical MTBF distribution.
- `DOWN` $\rightarrow$ `UP` (unscheduled failure): triggered at the repair completion time sampled from empirical MTTR.

All transitions are mediated through a min-heap event calendar, ensuring temporal consistency.

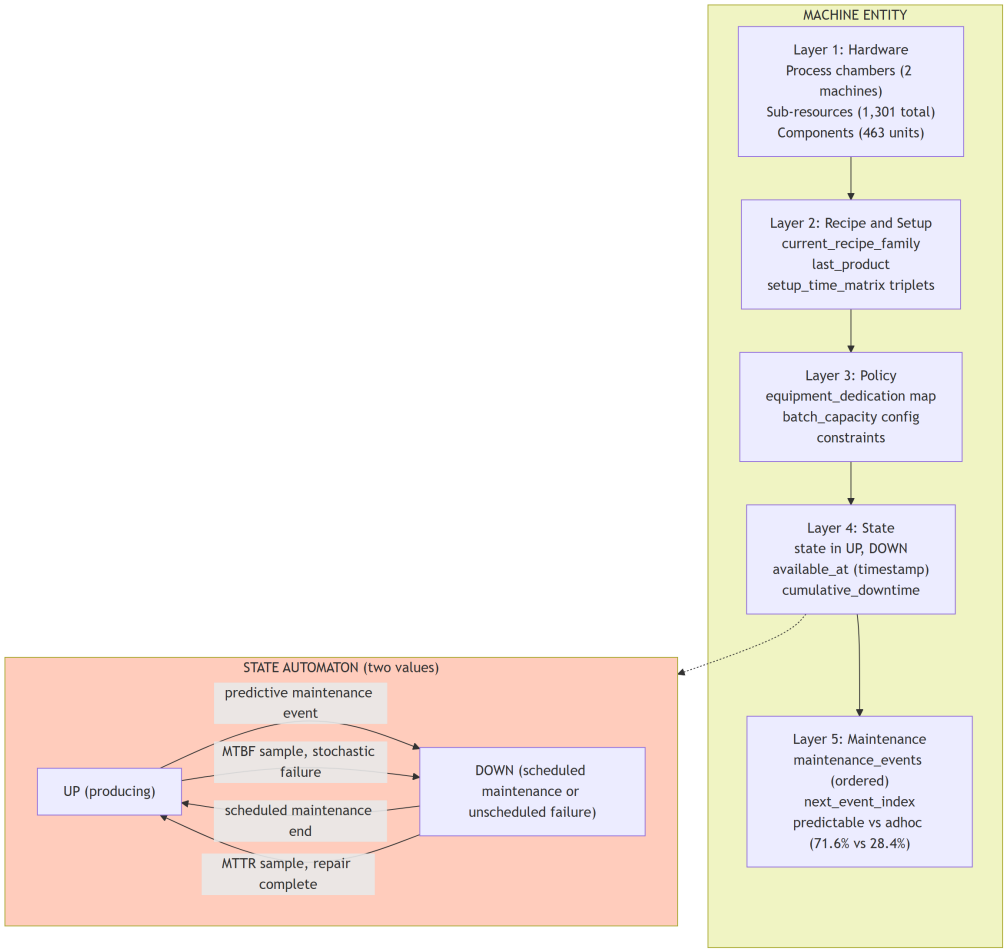

Figure 4.3: Internal structure of the Machine entity and the two-value state automaton it exposes. The layers are read by different consumers, not by a single caller: the Dispatcher's Stage 2 filters (Section **??**) read the State layer to exclude unavailable machines and the Policy layer to enforce dedication constraints, the setup-time cascade (Section **??**) reads the Recipe and Setup layer, and the event calendar (Appendix **??**) writes to the State and Maintenance layers whenever a maintenance or failure event fires. No single method touches all five layers at once; each is a narrow, independently updated interface.

4.4.4 Discrete-Event Simulation Loop

The main simulation loop is driven by a min-heap event calendar. At each iteration, the earliest pending event is dequeued and the simulation clock `sim.now` is advanced. Three event classes are processed. **Job completions** free sub-resources and advance the completed lot to its next step. Besides that, **maintenance events** transition machines between states, and **stochastic downtime events** are sampled on-the-fly from empirical MTBF and MTTR distributions.

After each event batch, the loop checks whether an actionable lot–machine pair exists (i.e., a lot that is dispatchable and has at least one eligible, available machine). If so, control returns to the agent for observation and action; otherwise the clock is advanced to the next pending event's timestamp (not by a fixed increment) and the check is repeated. This loop, `run_until_next_decision_point()`, is capped at 50,000 iterations as a safety net: if no actionable pair has appeared by then, or the event calendar empties first, the function returns `False` rather than advancing further, and `MetaSelectorEnv` treats this as episode termination (`done=True`) instead of forcing the clock forward. Appendix **??** details this loop's internal call sequence.

Each episode corresponds to one production month, starting from an empty fab with lots released according to historical arrival timestamps. The episode terminates when all lots are completed or 5,000 dispatching steps are reached (a timeout that acts as a deadlock guard). At each step, the environment provides observation $o_t \in \mathbb{R}^{35}$, the agent selects action $a_t \in \{0, \ldots, 5\}$, and the environment returns reward r_t and o_{t+1}.

4.4.5 Processing and Setup Time Prediction

Processing times are predicted by a three-strategy cascade. The first strategy samples from the lot's recorded real processing intervals for the same (flow, step) pair, injecting 25% coefficient-of-variation Gaussian noise. If no historical records exist, the second strategy queries a pre-trained LightGBM gradient-boosted model trained on the full 11,234-step dataset. If that fails, the third strategy falls back to a per-(flow, step, machine) median table. All times are clamped to $[0.01\text{ h}, 50\text{ h}]$.

Setup times are similarly retrieved via a three-strategy cascade: direct lookup in the (machine, from-family, to-family) matrix, then lookup via recipe family mapping, and finally a default of 0.5 hours if neither succeeds. If the lot and the machine share the same recipe family, no setup time is applied.

4.4.6 MES Operations and Event Types

The simulator distinguishes two families of events. **MES operations** encode business-level developments that originate in the historical data and are either reconstructed from it or injected stochastically at matching rates. **Internal clock signals**, by contrast, exist only inside the simulator to advance the event calendar and have no counterpart in the source data.

Normal-flow operations. Four MES operations describe a lot's ordinary progress through its flow. **Receive** marks a lot's arrival at the fab. **Dispatch** marks the real machine assignment recorded historically, used during pipeline reconstruction (Section **??**) to identify which machine actually processed each step. **TrackIn** marks the start of processing on the assigned machine, and **Terminate** marks its end. The pipeline matches TrackIn to Terminate per step to compute observed processing durations (Section **??**), which seed the processing-time cascade of Section **??**.

Exception operations. Five further MES operations interrupt this normal flow. Rather than replaying historical exceptions verbatim, the simulation generates them stochastically at each dispatch, drawn from empirical (flow, step)-level rates, as detailed in Section **??**.

- **Hold** (0.88% of events): Throughput blocked, with a mean duration of 41.5 hours.
- **Release** (0.89%): Clears the hold, and the lot re-enters the dispatch queue.
- **Abort** (0.63%): Wastes processing time, places the lot on hold, and marks the machine as unavailable for that lot.
- **Rework** (0.01%): Redirects the lot to a repair flow and increments the rework counter.
- **Temporary Off-Flow** (0.46%): Branches the lot to a secondary flow, such as metrology or sampling, then returns it to the main flow.

The delta-based reward signal (Section **??**) penalizes the agent in proportion to the incremental occurrence of each exception type during the episode.

Internal clock signals. The event calendar also carries signals with no MES equivalent, used purely to advance the simulator clock. Two of these signals, `lot_ready` and `machine_free`, mark, respectively, the moment a lot becomes eligible for its next step and the moment a machine becomes free. Both are scheduled by `schedule_single` at the instant a lot is dispatched, so the lot and machine state is already updated before either signal fires. Firing the signal itself does not change any state. It only invalidates the actionable-pair cache and forces the event loop to recheck whether a new decision point has been reached. Two further signals, `unscheduled_down` and `unscheduled_up`, mark stochastic machine failures and repairs, sampled from the empirical MTBF and MTTR distributions described in Section **??**.

4.4.7 Sources of Stochasticity

Although FlowSimulator replays historical lot arrivals and routing deterministically, five independent sources of randomness shape each training episode. Understanding them matters equally for interpreting agent behaviour and for reliably reproducing results from a fixed seed.

Processing time noise. Each processing time draw injects 25% coefficient-of-variation Gaussian noise around the historical or predicted value (Section **??**), so identical lot-machine pairs rarely take exactly the same duration twice.

Machine downtime. Stochastic failure and repair events are sampled on-the-fly from empirical MTBF and MTTR distributions fitted per machine (Section **??**), driving the `UP` to `DOWN` transition and its duration.

MES exception injection. Holds, aborts, reworks, and temporary off-flows are generated stochastically at each dispatch, drawn from empirical (flow, step)-level rates (Section **??** above). The rates themselves are fixed, but which specific dispatch triggers an exception is random: each exception type is sampled by its own independent Bernoulli draw against its (flow, step) rate, with abort taking absolute priority when sampled (capped at two occurrences per step to prevent an infinite abort loop) and hold, rework, and off-flow otherwise resolved so that at most one applies per completed step; the full sampling logic is given in Appendix **??**.

Hot-lot selection. At the start of every episode, a fraction of lots is randomly sampled to be *hot*, following the scenario's configured hot-lot fraction. Hot lots receive a tighter due-date factor for the remainder of the episode, while all other lots keep the loose default factor. Which lots are selected changes from episode to episode, even under the same scenario.

Scenario sampling. ScenarioMixEnv samples a scenario at every reset (Section **??**), so the due-date compression, setup scaling, and breakdown rates the agent experiences also vary across episodes.

All five sources draw from seeded random number generators, so the entire episode, including which lots are hot, when machines fail, and which dispatches trigger an exception, is exactly reproducible given a fixed seed. This is what lets the verification protocol below replay a policy checkpoint against bitwise-identical conditions.

4.4.8 Verification and Validation

Three mechanisms validate the simulation engine:

Trace-level verification. FlowSimulator supports a deterministic *replay mode* in which each lot is processed on the exact machine recorded in the historical MaterialLog, using observed processing times. Any deviation from the expected machine assignment immediately triggers an assertion error, catching bugs in event loop or eligibility filtering logic.

Eligibility mask validation. The machine eligibility mask for each (flow, step) pair is validated against the set of machines historically observed to process that combination. This confirmed that 100% of historical assignments fall within the computed eligible sets.

Aggregate KPI comparison. Table **??** compares simulated aggregate KPIs against historical values for the training month (January 2025), using the EDD baseline as the reference policy. Simulated values are averaged over 10 independent episodes.

The simulated cycle time and utilization are within 2% of historical values, providing confidence that the simulator captures the fab's aggregate throughput dynamics. The 13% underestimation of tardiness reflects simplifying assumptions (instantaneous lot transport, no operator

Table 4.2: Simulation fidelity: simulated vs. historical KPIs for the training month (January 2025).

KPI	Historical	Simulated (EDD)	Gap
Mean cycle time (h)	91.4	92.7	+1.4%
Machine utilization (%)	79.8	81.2	+1.8%
Mean tardiness (h)	16.2	14.1	−13.0%

constraints, absence of queue-time constraint holds). This bias is shared across all policies, so relative comparisons remain valid.

4.5 Two-Stage Dispatching Engine

The dispatching logic is encapsulated in the Dispatcher class, which implements a two-stage decision procedure at each step. This separation of concerns is key. The RL agent selects which rule to apply, and the dispatcher then applies that rule mechanically, enforcing all constraints.

4.5.1 Stage 1: Lot Selection by Rule Scoring

All lots satisfying the dispatchability condition (Equation **??**) and whose ready timestamp does not exceed the simulation clock form the candidate set. Lots with at least one eligible, available machine are retained as *processable*. The dispatcher logic partitions this processable set into two priority classes:

- *Critical lots*: those with a queue-time constraint expiring within one hour, dispatched unconditionally first by ascending constraint expiry.
- *Normal lots*: ranked by the scoring function of the currently selected rule a_t.

This priority-escalation logic is implemented in full, but, as noted under constraint C4 above, the queue-time constraint dictionary that would populate it is initialised empty and never filled from the current dataset. In every experiment reported in this thesis, the critical-lot class is therefore always empty, and all processable lots fall into the normal class, scored directly by the active rule a_t. The six dispatching rules in the action repertoire are defined in Section **??**.

4.5.2 Stage 2: Machine Assignment with Eligibility Filters

Given the selected lot L^*, the dispatcher applies sequential filters to its eligible machine set $\mathscr{E}(\text{flow}_{L^*}, \text{step}_{L^*})$:

1. **Dedication constraint**: If the current step type is subject to equipment dedication (e.g., lithographic overlay alignment), restrict to the historically dedicated machine.
2. **State filter**: Retain only `UP` machines (exclude machines currently `DOWN`, whether from scheduled maintenance or an unscheduled failure).

3. **Maintenance filter**: Exclude machines whose next maintenance window begins before the projected end time of the operation (computed via Equation **??** below).

Surviving machines are ranked by the active rule: by minimum total estimated completion time for throughput-oriented rules (SPT, CR), or by minimum available-at time for schedule-oriented rules (FIFO, EDD, LST, ATCS).

4.5.3 Operation Scheduling and Clock Advancement

Given the final (L*, M*) pair, the dispatcher computes the operation end time as:

$$t_{\text{end}} = \max\big(t_{\text{now}}, \texttt{available\_at}(M^*)\big) + T_{\text{setup}}(M^*, L^*) + T_{\text{proc}}(L^*, M^*) \tag{4.2}$$

where T_{setup} is the recipe-family changeover time (retrieved via the three-strategy cascade from Section **??**) and T_{proc} is the predicted processing time. This updates lot progress and machine availability. The event calendar then advances the clock through job completions, maintenance transitions, and stochastic downtime events until the next dispatchable state is reached.

4.6 Partially Observable Markov Decision Process Formulation

The fab dispatching problem is formalized as a Partially Observable Markov Decision Process (POMDP). The underlying FlowSimulator dynamics are Markovian over the full simulator state, every lot's progress, every machine's status, and the pending event calendar, but the agent never observes that state directly. It receives only a hand-engineered 35-dimensional summary at each decision epoch, so the decision problem the agent must solve is partially, not fully, observable.

4.6.1 POMDP Tuple

The problem is defined as:

$$\mathscr{M} = \langle \mathscr{S}, \mathscr{A}, \mathscr{P}, \mathscr{R}, \Omega, \mathscr{O}, \gamma, H \rangle \tag{4.3}$$

where:

- $\mathscr{S}$ is the true, unobserved state space, the full internal state of FlowSimulator.
- $\mathscr{A}$ is the action space (6 dispatching rules: FIFO, SPT, EDD, LST, CR, ATCS).
- $\mathscr{P}(s_{t+1}|s_t, a_t)$ is the (implicit) transition function over the true state, implemented by FlowSimulator.
- $\mathscr{R}(s_t, a_t)$ is the reward function (multi-term, described in Section **??**), computed from the true state so that exact waiting times and exception counts are never approximated.
- $\Omega \subset \mathbb{R}^{35}$ is the observation space actually available to the agent.

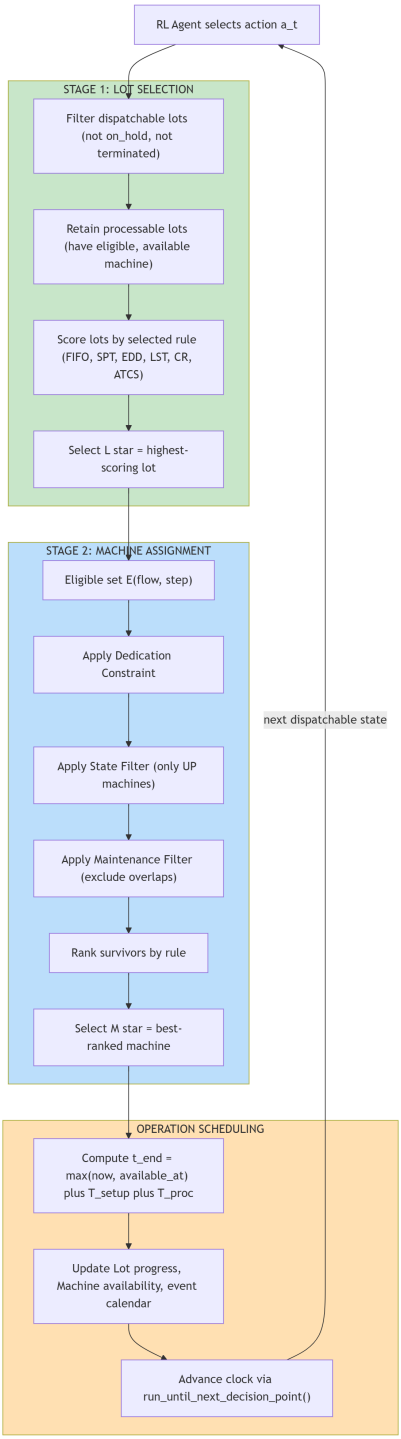

Figure 4.4: Two-stage dispatching engine. Stage 1 receives only the rule name a_t from the agent and returns a single lot L^*; Stage 2 receives L^* and returns a single machine M^*, querying the Machine object's State and Policy layers (Figure **??**) along the way. The pair (L^*, M^*) is the entire handoff to `schedule_single()`, which commits the assignment and advances the event calendar (Appendix **??**); neither stage has visibility into what happens after that handoff.

- $\mathcal{O} : \mathcal{S} \to \Omega$ is the observation function, a deterministic aggregation that maps the true state to the 35-dimensional feature vector detailed in Section **??**.
- $\gamma = 0.99$ is the discount factor, giving effective planning depth of 100 steps.
- $H \leq 5{,}000$ is the episode horizon (typical episodes terminate naturally at 1,700–2,000 steps).

Episode termination follows directly from these two limits:

$$\text{done}(s_t) = \begin{cases} \text{True} & \text{if } |\mathcal{L}_t^{\text{active}}| = 0 \quad \text{(all lots completed)} \\ \text{True} & \text{if } t \geq H_{\max} = 5{,}000 \\ \text{False} & \text{otherwise} \end{cases} \tag{4.4}$$

The limit $H_{\max}$ acts purely as a deadlock guard. In practice, episodes terminate naturally once all lots complete, well before this bound is reached.

The learning objective is:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{H} \gamma^t r_t \right] \tag{4.5}$$

where the policy $\pi_\theta(a_t \mid o_t)$ is defined over observations rather than the true state, since s_t itself is never available to the agent. This reactive formulation, conditioning on the current observation alone rather than the full observation history, is standard practice in deep POMDP-RL and is validated empirically by the training results of Chapter **??**.

It is worth noting explicitly that PPO's convergence guarantees, like those of most policy-gradient methods, are derived under the Markov assumption. Applying PPO to a reactive policy over the partially observable o_t is therefore a theoretical approximation rather than a formally justified extension, analogous to frame-stacking in place of a recurrent architecture in Atari-style deep RL. The mitigations described above (trend features, fab-level aggregation) narrow the resulting Markov gap but do not close it. This approximation is accepted here on practical grounds, and its adequacy is assessed empirically rather than proven theoretically.

4.6.2 Observation Representation: 35 Features Across 10 Categories

The observation $o_t \in \mathbb{R}^{35}$ at decision epoch t encodes the fab's operating condition through aggregated features organized in ten semantic categories. Table **??** summarizes the categories. The complete 35-feature specification with normalization constants is provided in Appendix **??**.

On the reserved positions. Indices 9 and 10 are not blank by design; they hold two computed calendar features, `current_hour` and `current_weekday`, that are deliberately zeroed at runtime rather than removed from the vector. This choice reflects a SHAP (SHapley Additive exPlanations) feature-importance check performed during development, which found these two temporal

Table 4.3: Observation vector: 35 features organized in 10 semantic categories, plus two reserved-but-inactive positions that are not counted among the ten (see note below).

Category	**Indices**	**Content**
Lot Counts	0–1	Ready lots, active lots
Slack Time	2–4	Mean, min, max slack (hours)
Waiting	5	Mean waiting time in queue
Tardiness & Completion	6–8	Cumulative tardiness, completion rate, OTR
Reserved (not one of the ten)	9–10	`current_hour`, `current_weekday`: computed but zeroed at runtime
WIP Structure	11–13	Near-deadline fraction, remaining steps, holds
Urgency	14–18	Mean CR, slack quantiles, urgency fraction
Trends	19–21	Completion acceleration, tardiness momentum, queue pressure
Fab Disturbance	22–27	Machines down, aborts, reworks, hot lots, load CoV, throughput
Setup Pressure	28	Mean setup time over mean processing time among ready lots
Heuristic Cost Proxies	29–34	Estimated immediate cost of each rule

indicators to contribute negligibly to the trained policy's action distribution relative to the other 33 features; rather than resizing the observation space and re-deriving every downstream index, the two positions were left in place but zeroed, so the vector's dimensionality and indexing remain stable across the codebase. This is why they are excluded from the ten semantic categories above: unlike every other category, their content carries no information the policy can act on.

The method `MetaSelectorEnv._get_obs()` computes all 35 entries directly from the `FlowSimulator`'s internal Lot and Machine collections at the moment a decision point is reached (Appendix **??**); no category is cached from a previous step except the two trend features at indices 19–20, which explicitly reference the prior observation. The full vector o_t is the only channel through which the policy and value networks (Section **??**) ever observe the fab; nothing else about the simulator's state reaches the agent.

Three example features illustrate the design:

- **Queue depth (indices 0–1):** $o_t^{[0]} = |\mathscr{L}_t^{\text{ready}}|/500$, $o_t^{[1]} = |\mathscr{L}_t^{\text{active}}|/500$, where $\mathscr{L}_t^{\text{ready}}$ is the set of lots currently ready for dispatch and $\mathscr{L}_t^{\text{active}}$ the set of all lots still in the system, whether ready, in process, or on hold. These capture the fab's congestion level.
- **Urgency fraction (index 17):** $o_t^{[17]} = \frac{|\{L_j : CR(L_j) < 1\}|}{|\mathscr{L}_t^{\text{ready}}|}$, where $CR(L_j) = \frac{d(L_j) - t_{\text{now}}}{\tau_{\text{rem}}(L_j)}$ is the critical ratio, $d(L_j)$ the lot's due date, and $\tau_{\text{rem}}(L_j)$ its estimated remaining processing time across all steps still to complete. This captures the fraction of lots behind schedule.
- **Machine availability (index 22):** $o_t^{[22]} = |\{M \in \mathscr{M} : \text{state}(M) = \texttt{DOWN}\}|/|\mathscr{M}|$, where $\mathscr{M}$ is the set of all machines in the fab. This captures fab disturbance level.

Heuristic cost proxies. Features 29–34 encode the estimated immediate dispatch cost of each rule applied to the current queue, enabling in-context rule comparison within a single forward pass:

$$o_t^{[29+k]} = \min\left(1,\ \frac{\text{cost}(a_k,\, o_t)}{100}\right), \quad k = 0, \ldots, 5 \tag{4.6}$$

where $\text{cost}(a_k, o_t) = 0.05\,\sigma_k + 0.01\,w_k + 0.10\,\tau_k$ is a single shared cost formula, evaluated six times, once per rule a_k, not six different rule-specific formulas: w_k is the waiting time and σ_k the setup time of the lot rule a_k would select for dispatch, previewed via `preview_choice_for_rule()` (Appendix **??**) without actually committing the assignment, and τ_k is the resulting tardiness risk, the number of hours by which that lot's projected completion time would exceed its due date if dispatched now. The six proxies therefore differ only in *which candidate lot* each rule would select, not in how that candidate's cost is scored; this is why they enable direct, apples-to-apples comparison of the six rules within a single forward pass. A fourth term, an urgency-skip penalty, is computed internally but is excluded from the cost actually used here (disabled for an ablation variant), so it has no effect on the reported values.

Table 4.4: Heuristic cost proxy features (indices 29–34). All six share the same cost formula; only the rule a_k whose candidate lot is scored differs by row.

Idx	Feature	Rule previewed (a_k)
29	`heuristic_cost_FIFO`	FIFO
30	`heuristic_cost_SPT`	SPT
31	`heuristic_cost_EDD`	EDD
32	`heuristic_cost_LST`	LST
33	`heuristic_cost_CR`	CR
34	`heuristic_cost_ATCS`	ATCS

These proxies act as built-in heuristic supervision. When one proxy dominates, the agent tends to select that rule, and when all are low, the agent can safely explore. The value of RL training comes precisely from cases where the immediate-cost-minimizing rule differs from the globally optimal choice, a distinction that the surrounding contextual features allow the agent to resolve. If the agent merely learned to copy the best proxy, the regret-shaped reward variant B4 would offer no improvement over simpler alternatives, but it consistently does (Section **??**).

The contribution of these proxies, together with the disturbance features at indices 22–27, is examined empirically in a preliminary state-representation ablation (Chapter **??**). Table **??** summarizes the state configurations explored during the design phase, ranging from the full 35-feature vector down to a minimal 7-feature queue-only representation.

Coping with partial observability. Because $o_t = \mathcal{O}(s_t)$ is a lossy summary of the true state, the policy is necessarily reactive rather than fully Markovian, a direct consequence of the POMDP formulation adopted here. Two design choices mitigate the resulting information loss. First, the trend features at indices 19–21 encode first-order temporal derivatives of key aggregates, giving

Table 4.5: State space ablation variants.

Variant	Dim.	Content
Full ($\mathscr{S}_{\text{full}}$)	35	Core 28 features plus setup pressure plus 6 heuristic cost proxies
Reduced1 ($\mathscr{S}_{\text{red1}}$)	29	Core 28 features plus setup pressure, no heuristic cost proxies
Reduced2 ($\mathscr{S}_{\text{red2}}$)	22	Core features only, no disturbance signals, no proxies
Minimal ($\mathscr{S}_{\text{min}}$)	7	Lot counts, slack, waiting, and cumulative tardiness only

the policy implicit access to recent history without requiring an explicit belief state or a recurrent architecture. Second, per-lot and per-machine detail is deliberately aggregated into fab-level scalars, since the agent's action operates at the global rule-selection level rather than on individual lot–machine assignments, which the dispatcher resolves deterministically (Section **??**). One residual source of partial observability is left unmitigated: the prior sequence of rule selections is not encoded in o_t. This is accepted here as an untested simplification rather than a validated one: no autocorrelation analysis of consecutive rule choices was performed in this thesis, so its practical impact on policy learning is not directly measured. The trend features at indices 19–21 and the effective planning depth of 100 steps under $\gamma = 0.99$ provide some indirect compensation, but quantifying the residual effect of omitting rule history is left as a direction for future empirical work.

4.6.3 Action Space: Six Dispatching Rules

The action $a_t \in \mathscr{A} = \{0,1,2,3,4,5\}$ selects which dispatching rule the dispatcher applies at the current decision epoch. All six actions are valid at every step, so no action masking is required, since the dispatcher handles infeasibility internally. If a rule scores a lot but no eligible machines exist, for instance, the dispatcher simply moves to the next lot.

Ablation variants explored during the design phase are summarized in Table **??**, with the full six-rule set adopted for all reported experiments. Relative to the full set, the Common variant omits LST and CR, retaining ATCS, which is theoretically optimal under setup-dominant conditions; the Minimal variant reduces further by also dropping FIFO, leaving only the throughput-, deadline-, and setup-aware rules (SPT, EDD, ATCS). A comparative ablation of these configurations at full training scale is deferred to future work.

Table 4.6: Action space ablation variants.

Variant	Actions	Rules
Full	6	FIFO, SPT, EDD, LST, CR, ATCS
Common	4	FIFO, SPT, EDD, ATCS
Minimal	3	SPT, EDD, ATCS

4.7 Reward Function Design

The reward function is central to shaping the learning objective. Multiple variants are tested in the ablation study (Section **??**) to isolate reward design effects. The primary variant and the rationale for the ablations are presented below. Appendix **??** complements this section with material not repeated here: the observed event-frequency data underlying each coefficient in Table **??**, the bound used to prevent early-training gradient explosion, and a discussion of how the aggregate calibration may under- or over-weight terms under adversarial scenarios.

4.7.1 Primary Reward (Variant B0)

The primary reward is a multi-term signal that encodes the manufacturing cost hierarchy:

$$r_t = -\alpha w_t - \gamma_{\text{setup}}\sigma_t - \gamma_{\text{hold}}\Delta H_t - \gamma_{\text{offload}}\Delta O_t - \delta_{\text{rework}}\Delta W_t - \zeta_{\text{abort}}\Delta A_t + B_t \tag{4.7}$$

where:

- w_t is the waiting time (hours) of the dispatched lot.
- σ_t is the setup time (hours) for recipe family changeover.
- $\Delta H_t, \Delta O_t, \Delta W_t, \Delta A_t$ are per-step increments in holds, off-flows, reworks, aborts.
- B_t is the completion bonus, awarded only at the step where a lot finishes its final processing step: $+10$ if on-time, $-2 \times \text{delay}_h$ if late, where delay_h is the number of hours by which the completion timestamp exceeds the lot's due date, and 0 at every other step.

Table **??** summarizes coefficients calibrated to reflect manufacturing cost impact. Abort carries the highest penalty, 10.0, because an entire lot is destroyed. Rework, at 3.0, incurs an extra processing cycle plus yield loss, and hold, at 1.0, blocks throughput. Waiting, at 0.05, and setup, at 0.10, receive lower weights because they are more frequent but less disruptive.

Table 4.7: Reward coefficients (variant B0), ordered by manufacturing cost impact.

Term	Symbol	Value	Rationale
Waiting time	α	0.05	Per-waiting-hour proxy for cycle time
Setup time	γ_{setup}	0.10	Direct machine downtime during changeover
Hold	γ_{hold}	1.0	Throughput fully blocked (0.88% of ops)
Off-flow	γ_{offload}	0.5	Secondary disruption (1.95% of lots)
Rework	δ_{rework}	3.0	Extra cycle + yield loss (0.19% of lots)
Abort	ζ_{abort}	10.0	Entire lot destroyed (7.87% of lots)
On-time bonus	B_t^+	+10	Positive reinforcement
Tardiness mult.	B_t^-	2.0	Per-hour-late penalty

4.7.2 Reward Ablation Variants (B1–B5)

Five variants are tested to isolate reward design effects:

- **B1 (Completion-event only):** $r_t = +1$ if on-time, $-0.1 \times \text{delay}_h$ if late, 0 otherwise. Minimal signal.
- **B2 (B1 + wait penalty):** $r_t = r_t^{\text{B1}} - 0.01 w_t$. Adds cycle-time awareness.
- **B3 (B2 + setup penalty):** $r_t = \mathbb{1}[\text{on-time}] - 0.5\,\text{delay}_h \mathbb{1}[\text{late}] - 0.01 w_t - 0.05\sigma_t$, where $\mathbb{1}[\cdot]$ is the indicator function, equal to 1 when the bracketed condition holds and 0 otherwise. Adds setup sensitivity. In the implementation, the late-completion term is also five times steeper than in B1/B2 ($-0.5\,\text{delay}_h$ instead of $-0.1\,\text{delay}_h$), a deliberate change carried forward into B4 below, since B4 is computed on top of the B3 base reward.
- **B4 (Regret shaping, $w = 0.3$):** $r_t = r_t^{\text{B3}} + w\Phi_{\text{regret}}(o_t, a_t) + \beta_{\text{match}} \mathbb{1}[\text{chosen} = \text{best}]$, with $w = 0.3$ and $\beta_{\text{match}} = 0.2$. Adds counterfactual penalty for suboptimal rule choice.
- **B5 (Pure regret):** $r_t = w\Phi_{\text{regret}}(o_t, a_t) + \beta_{\text{match}} \mathbb{1}[\text{chosen} = \text{best}]$, with the same w and β_{match} as B4. Regret only, for ablation control.

Regret shaping of B4. At each step, B4 computes the gap between the immediate heuristic cost of the chosen rule and the cost of the best available rule in the current queue, known as the *regret*. The regret is passed through a saturating shaping function, $\Phi_{\text{regret}}(o_t, a_t) = -\tanh(\text{regret}/10)$, so a suboptimal rule choice produces a negative penalty that flattens out rather than growing without bound as the gap widens. This term is weighted by w, giving $w\Phi_{\text{regret}}(o_t, a_t) \in [-w, 0]$. A separate match bonus, $\beta_{\text{match}} = 0.2$, is added on top whenever the chosen rule exactly matches the best available rule; critically, this bonus is added *after* the w-weighting, not multiplied by w itself, since regret is exactly zero whenever chosen and best coincide and the two terms are mutually exclusive at any given step. With $w = 0.3$, the combined shaping term therefore ranges over approximately $(-0.3, 0.2]$: the lower bound comes from $w\Phi_{\text{regret}}$ alone (large regret, no match), and the upper bound of exactly 0.2, unscaled by w, is reached whenever the agent matches the best rule. The weight $w = 0.3$ was selected empirically. At $w \geq 0.5$, the shaping term dominates and destabilizes learning, and at $w \leq 0.1$, the improvement over B3 is negligible. B4($w = 0.3$) is the production variant for all reported experiments.

4.8 Reinforcement Learning Training Architecture

With the POMDP defined, this section describes how the RL agent learns a policy that maps observations to rule selections. This section covers environment wrapping, the neural network architecture, normalization and training details, and the PPO algorithm.

4.8.1 Gymnasium Interface and Scenario Sampling

The simulation is exposed to the RL agent through a two-layer Gymnasium wrapper stack:

- **MetaSelectorEnv** wraps FlowSimulator and provides the standard reset() / step(action) interface. At each call to step(), it translates the integer action to a named rule, invokes the dispatcher, schedules the operation, computes MES exception deltas, evaluates the reward, advances the clock, and extracts a fresh 35-dimensional observation.
- **ScenarioMixEnv** sits above MetaSelectorEnv and introduces training diversity by sampling a scenario at every episode reset, reconstructing the inner environment with corresponding due-date slack compression, setup scaling, breakdown rates, and hot-lot promotion fractions. The training catalogue comprises ten scenarios spanning two categories, rule-regime scenarios (setup-heavy, tight-deadlines, high-urgency, setup-and-slack) and fab-disturbance scenarios (hot-lots at 15% and 30% promotion, maintenance combined with hot lots, high-mix pressure, bottleneck failures, scheduled maintenance), with the unmodified normal scenario reserved for validation. The full parameter values for every scenario are listed in Appendix **??**.

This multi-scenario sampling strategy is intended to push the policy toward generalizing across operating conditions rather than overfitting to any single configuration; whether it succeeds is an empirical question, not a guarantee of the sampling design itself. Section **??** in Chapter **??** tests whether the trained policy's rule-selection behaviour is consistent with each scenario's expected regime, and the per-scenario on-time-rate and tardiness breakdown of Appendix **??** gives the corresponding quantitative evidence on the held-out test month.

4.8.2 Neural Network Architecture

PPO employs a multilayer perceptron function approximator:

- **Input:** $|\Omega| = 35$ features.
- **Hidden layers:** Two fully connected layers of 128 neurons each with tanh activations.
- **Output:** Two heads, a policy head producing 6 action logits via softmax to $\pi_\theta(a|o)$, and a value head producing $V_\theta(o)$.

Both the hidden-layer width (two layers of 128 units) and the activation function (hyperbolic tangent) follow the Stable-Baselines3 `MlpPolicy` default configuration; neither was overridden in the training scripts, and no explicit width or activation ablation was run. The 128-unit width is nonetheless a reasonable fit for the problem's dimensionality, 35 observation features mapped to 6 discrete actions, without being needlessly large, and the bounded $(-1,1)$ output of tanh is a natural match for the VecNormalize-clipped observation range described in Section **??**.

4.8.3 Normalization and Training Details

Observation and reward normalization. Each of the 35 state features is normalized at runtime to approximately zero mean and unit variance using running statistics maintained by a VecNormalize wrapper. This ensures features on different scales contribute equally and that the critic's value estimates remain consistent across scenarios.

Advantage estimation. PPO estimates advantages using Generalized Advantage Estimation (GAE) with $\lambda = 0.95$, balancing short-term accuracy against longer-horizon prediction.

PPO update rule. PPO maximizes expected advantage while constraining each update so the new policy deviates at most $\varepsilon = 0.2$ from the previous policy, preventing destabilizing jumps.

Entropy regularization. The entropy coefficient $c_{\text{ent}} = 0.03$ is elevated above the Stable-Baselines3 default to prevent premature policy collapse onto SPT (which is a strong local attractor). Without this bonus, the policy rapidly converges to near-deterministic SPT selection before exploring scenario-specific alternatives.

Table **??** summarizes all hyperparameters. Training runs for 200,000 timesteps across four parallel environments in approximately 3.7 hours of wall-clock time on a four-core CPU.

Table 4.8: PPO hyperparameters and training configuration.

Parameter	Value	Rationale
Learning rate α	10^{-4}	Reduced for stability under multi-scenario training
Discount factor γ	0.99	Long-horizon manufacturing, effective horizon = 100 steps
GAE λ	0.95	Bias–variance trade-off
Clip range ε	0.2	Standard PPO clipping
Rollout per env.	512	Steps per environment per rollout (`n_steps`); 2,048 transitions per batch across 4 envs
Mini-batch size	64	SGD minibatch size within each PPO update
Epochs per rollout	10	Policy reuse vs. drift balance
Entropy coeff.	0.03	Prevent collapse to single rule
Parallel envs.	4	CPU-bound training
Total timesteps	200,000	Budget for convergence study

Where this loop lives, and how the reported checkpoint was chosen. The loop in Figure **??** is not custom code: it is Stable-Baselines3's internal `PPO.learn()` implementation, invoked with a single call once the hyperparameters of Table **??** are set. Three callbacks hook into that internal loop without altering its control flow. An `EvalCallback` runs 3 deterministic episodes on the held-out validation environment roughly once per rollout, saving the model with the best validation on-time rate seen so far to a separate checkpoint; this callback exists to monitor for overfitting during training, not to select the checkpoint ultimately reported. As detailed in Section **??**

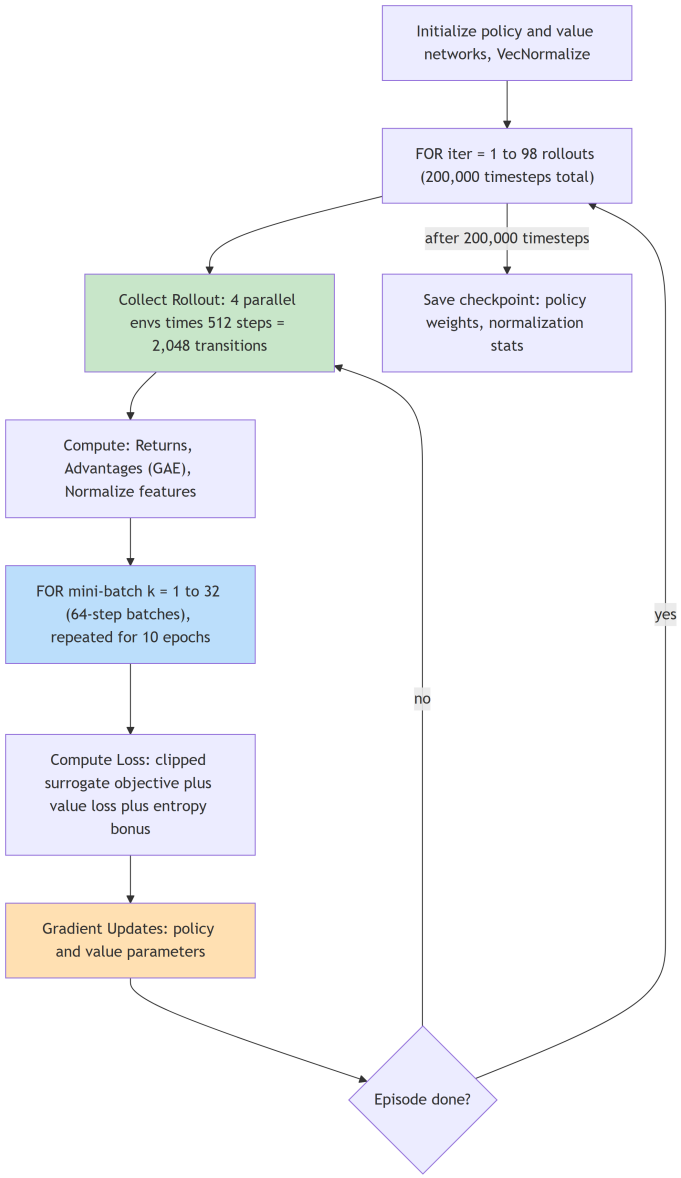

Figure 4.5: PPO training loop. Each of the 4 parallel `MetaSelectorEnv` instances contributes 512 `step(action)` calls per rollout (2,048 transitions total), handed to Stable-Baselines3's GAE and loss computation over 10 epochs of mini-batch updates before the next rollout begins, for roughly 98 rollouts across the 200,000-timestep training budget.

of Chapter **??**, the policy actually evaluated there is the final-rollout checkpoint, not this best-validation snapshot, a deliberate choice to prioritise asymptotic performance over peak-snapshot selection. A `CheckpointCallback` periodically saves the current model on its own schedule regardless of validation performance, so training can be resumed or an intermediate timestep inspected later. A `RegretLoggingCallback` reads the per-step regret value, whether the chosen rule matched the locally best rule, and the per-rule heuristic-cost sums from the `info` dictionary that `MetaSelectorEnv.step()` already returns, and logs these to TensorBoard, providing the diagnostic trail behind the regret-shaping behaviour of variants B4 and B5 discussed in Section **??**. The full implementation detail of all three callbacks is given in Appendix **??**.

4.9 Dispatching Rules as Baselines

Six classical dispatching rules serve both as the agent's discrete action repertoire and as fixed-policy performance baselines. Each defines a priority function applied to all eligible lots at each decision epoch, and the lot with the extremal score is dispatched first.

FIFO. $f_{\text{FIFO}}(L_j) = t_{\text{ready}}(L_j)$. Prioritizes by arrival time, which prevents starvation and provides fairness regardless of urgency.

SPT. $f_{\text{SPT}}(L_j) = T_{\text{proc}}(L_j, M^*)$, the same predicted-processing-time quantity used in Equation **??**. Prioritizes by predicted processing time, minimizing mean flow time and maximizing throughput under congestion.

EDD. $f_{\text{EDD}}(L_j) = d(L_j)$. Prioritizes by due date, minimizing maximum lateness and proving effective for on-time delivery.

LST. $f_{\text{LST}}(L_j) = (d(L_j) - t_{\text{now}}) - \tau_{\text{rem}}(L_j)$. Remaining slack minus remaining work, which is more nuanced than EDD under heterogeneous processing times.

CR. $f_{\text{CR}}(L_j) = (d(L_j) - t_{\text{now}})/\tau_{\text{rem}}(L_j)$. A dynamic urgency ratio that adapts naturally to reentrant flows.

ATCS. $f_{\text{ATCS}}(L_j) = -\frac{1}{p_j}\exp\left(-\frac{\max(0, d_j - p_j - t_{\text{now}})}{k_1 \bar{p}}\right)\exp\left(-\frac{s_j}{k_2 \bar{s}}\right)$, where $p_j = T_{\text{proc}}(L_j, M^*)$ is lot j's predicted processing time, the same quantity as in Equation **??**, $d_j = d(L_j)$ its due date, s_j the predicted setup time on the candidate machine, $\bar{p}$ and $\bar{s}$ the mean processing and setup times over the current queue, and $k_1 = k_2 = 2.0$ are look-ahead scaling constants that control how quickly each exponential term decays. Combines throughput, urgency, and setup-cost awareness, making it the only rule that explicitly penalizes recipe changeovers.

Together, these six rules span the primary scheduling objectives: fairness (FIFO), throughput (SPT), due-date compliance (EDD, LST), dynamic urgency (CR), and setup awareness (ATCS).

4.10 Summary

This chapter presented the complete methodological framework for learning a meta-dispatcher policy. A four-layer architecture, spanning data processing, simulation, and learning stages layered on top of the raw production data, was introduced to enable systematic development and validation. The data processing pipeline transforms 11.4 million raw MES events into the simulation-ready artefacts of Table **??**: the flattened lot-step table, the machine hierarchy and its serialized objects, maintenance and downtime statistics, the setup-time matrix, the eligibility mask, and the monthly train/validation split. The FlowSimulator discrete-event simulation engine, parameterized by these artifacts, replicates the fab's operating dynamics with 1.4–2% fidelity relative to historical KPIs. The two-stage dispatcher enforces six operational constraints and applies the agent's selected rule mechanically. The POMDP formulation defines a 35-dimensional observation space capturing fab-level aggregates, a 6-action discrete space (the dispatching rules), and a multi-term reward that reflects manufacturing cost hierarchy. The PPO-trained neural network learns to select rules adapted to fab state, discovering context-dependent policies that go beyond any single static rule. The ablation variants, spanning reward formulations B0 through B5 as well as alternative state configurations and action sets, provide a framework for isolating which design choices matter most, to be evaluated empirically in Chapter **??**.

Chapter 5

Experimental Results

This chapter presents the empirical evaluation of the reinforcement learning meta-dispatcher introduced in Chapter **??**. The evaluation is structured to address three interrelated research questions stated in Section **??**: whether RL-based rule selection outperforms static dispatching baselines, which reward formulation produces the most effective learning signal, and whether the learned policy generalizes across heterogeneous operating regimes. Section **??** describes the data splits, scenario catalogue, baseline rules, and performance metrics. Section **??** analyses PPO convergence and the effect of reward shaping on learning dynamics. Section **??** benchmarks the trained policy against five classical dispatching heuristics. Section **??** examines the rule-selection preferences learned within individual training scenarios. Section **??** investigates rule-selection sensitivity to evaluation conditions across four independently-seeded policy replicas.

The experiments are designed so that each section addresses one research question: RQ1 is answered in Section **??** through multi-metric comparison against five baselines; RQ2 is answered in Section **??** through reward ablation; RQ3 is answered across Sections **??** and **??** through scenario-specific rule-switching analysis and evaluation-condition sensitivity across independently-seeded replicas. Section **??** synthesizes the findings and discusses practical implications.

5.1 Experimental Setup

Provenance of the results reported in this chapter. Unless a table or figure caption states otherwise, every performance metric, rule-selection percentage, and statistical test in this chapter is computed by a single evaluation script, `evaluate_policy.py`, run against the held-out May 2025 test month. The script logs one row per dispatch step (rule chosen, reward, tardiness, on-time flag, and related KPIs) to a per-step CSV, from which per-scenario aggregates such as rule-usage frequency and mean tardiness are computed; the five static baselines are evaluated by the same script under an identical, fixed sequence of episode seeds, which is what makes the paired comparisons in Table **??** possible. The full implementation, including the exact logged columns, is documented in Appendix **??**; how the training checkpoints referenced in Section **??** differ from the main evaluated policy is documented separately in Appendix **??**.

5.1.1 Data Split and Training Configuration

The experimental dataset consists of five months of operational records from a real semiconductor fabrication facility (January–May 2025). Following the temporal split introduced in Section **??**, training uses January 2025 exclusively, and the held-out test set uses May. Table **??** summarizes the lot counts and configuration for each partition.

Table 5.1: Data split strategy: train (Jan 2025), test (May 2025). All figures are from the simulation routing dataset (281 flows, Section **??** of Chapter **??**).

Partition	Months	Lots	Role
Training	Jan 2025	1,857	Policy learning (200K timesteps, 4 envs)
Validation	Apr 2025	604	Hyperparameter selection, early stopping
Test	May 2025	549	Final evaluation (30 independent episodes)

PPO training ran for 200,000 timesteps across four parallel environments, with policy updates every 2,048 environment steps (512 per environment, `n_steps`, batched across the 4 environments). This yielded approximately 1,800 complete episodes per reward variant (each episode corresponds to one simulated production month), converging in approximately 3.7 hours of wall-clock time on a four-core CPU.

Validation protocol and checkpoint selection. The April 2025 partition (604 lots) served two roles during training. First, validation OTR was evaluated every 10,000 timesteps to monitor for overfitting and confirm that performance on unseen data tracked training performance. No variant exhibited degradation on the validation set, confirming that multi-scenario exposure prevents overfitting to the single-month (January 2025) training data. Second, the policy checkpoint evaluated in Section **??** corresponds to the final rollout (approximately the last 10 episodes), not the best-validation-snapshot; this choice prioritises asymptotic performance over peak-snapshot selection. All four reward variants completed the full 200K-timestep budget, as training-set and validation-set OTR continued to improve or stabilise throughout without exhibiting early plateau.

Simulation efficiency and environment optimisation. A key implementation challenge was achieving sufficient simulation throughput for 200,000-timestep training. In the initial simulator architecture, the maintenance event handler polled all 622 machines at every simulation step to check for pending events, an $\mathcal{O}(n_{\text{machine}})$ overhead that caused progressive frame-rate degradation as the episode's simulation clock advanced. Benchmarking on a 4,096-step window revealed throughput falling from approximately 23 steps s^{-1} at episode start to below 2 steps s^{-1} by episode end, with the full 4,096-step window requiring over six hours to complete. The bottleneck was eliminated by replacing the per-step linear scan with a min-heap priority queue, pre-loading all maintenance events at episode start. This reduced the per-step maintenance check from $\mathcal{O}(622)$ polling iterations to $\mathcal{O}(1)$ heap-peek. After this change the same 4,096-step benchmark completed in under two minutes at a sustained 38–41 steps s^{-1}, making iterative reward-shaping experiments across four variants practically feasible.

5.1.2 Scenario Definitions

To assess generalisation across operating conditions, ten training scenarios are constructed from two catalogues, sampled via round-robin at each episode reset.

Catalogue A: Controlled Rule-Regime Scenarios (5 total, 4 training + 1 validation). These scenarios modulate due-date pressure and setup costs to create conditions in which individual dispatching rules are theoretically optimal, allowing the agent's learned preferences to be validated against scheduling theory.

Table 5.2: Catalogue A: Rule-regime scenarios. Due-date slack and setup scales modulate dispatching costs. Expected Rule states the prediction from classical scheduling theory, compared against the agent's actual rule usage in Section **??**. The *normal* scenario is reserved for validation.

Scenario	Description	Scale Parameters	Expected Rule
setup_heavy	Elevated setup times; recipe changeover dominates cost	Setup $\times 5$	ATCS
tight_deadlines	Compressed due-date slack; urgency dominates	Slack $\times 0.5$	EDD
high_urgency	Further compressed due-date slack; most lots operate at or below a critical ratio of 1	Slack $\times 0.3$	ATCS
setup_and_slack	Combined setup pressure and tight deadlines	Setup $\times 4$, slack $\times 0.6$	LST
normal	Baseline historical dynamics; validation only	Unmodified	—

Catalogue B: Fab Disturbance Scenarios (6 total, all in training set). These scenarios introduce realistic production disruptions such as equipment failures, planned maintenance, and emergency lot promotion, testing policy robustness beyond idealized operating conditions. Table **??** details the disturbance parameters for each scenario.

5.1.3 Dataset Composition and Scenario Application

Single flow set, multiple scenario modifiers. All ten training scenarios are applied to a single, fixed set of 281 flows in the simulation routing dataset (Dataset B, Section **??** of Chapter **??**). The routing and arrival patterns remain constant across all scenarios. What varies between scenarios is not the lot mix but the operating parameters and disturbance conditions governing those lots' dispatching costs and fab availability. This experimental isolation is critical for validating causal claims about rule selection: if the agent learns different dispatching preferences in *tight_deadlines* versus *setup_heavy*, those differences arise from the scenario parameter modifiers, not from confounding changes in lot volume or flow composition.

Catalogue A: parameter scaling modifiers. The four Catalogue A rule-regime scenarios (Table **??**) modify the costs associated with dispatching decisions while holding routing constant.

Table 5.3: Catalogue B: Fab-disturbance scenarios. Machine failure rates (MTBF scaling), maintenance density, and hot-lot fraction modulate operating conditions. All six scenarios are included in the training set.

Scenario	Description	Disturbance Parameters
high_mix_pressure	High setup cost combined with moderately compressed deadlines, testing the throughput/urgency trade-off	Setup $\times 2.5$, slack $\times 0.7$
bottleneck_failures	Elevated machine failure rate with longer repairs	MTBF $\times 1/3$, MTTR $\times 2$
scheduled_maintenance	Elevated planned maintenance density with moderately compressed deadlines	Maintenance events $\times 2$, slack $\times 0.8$
hot_lots_regime	Emergency priority lots at 15% of incoming volume	Hot-lot fraction 15%
hot_lots_30pct	Elevated emergency lot proportion	Hot-lot fraction 30%
maintenance_hot_lots	Combined machine failures and hot-lot promotion	MTBF $\times 1/3$, MTTR $\times 2$, hot-lot fraction 30%

In the *tight_deadlines* scenario, each lot's due-date slack is compressed by a factor of 0.5: a lot nominally due in 10 hours becomes due in 5 hours, increasing the urgency signal in the observation vector (the `min_slack` and `avg_slack` features defined in Section **??**). Similarly, in the *setup_heavy* scenario, recipe changeover times are scaled by $5\times$, inflating the setup-cost penalty. The lot's origin, destination, processing times, and sequence within the flow remain identical to the historical record; only the cost parameters change. This design allows the agent to discover scenario-optimal rules purely from cost-signal changes in the observation vector.

Catalogue B: disturbance event injection. The six Catalogue B fab-disturbance scenarios (Table **??**) introduce stochastic events such as machine failures, maintenance holds, lot priority promotions, that modify the effective fab state without changing lot parameters. For example, in the *bottleneck_failures* scenario, identified bottleneck machines (primarily the metrology tools in Table **??** of Appendix **??**) receive sampled downtime events with MTBF reduced to 1/3 of historical values from the `MaintenanceLog` (Section **??** of Chapter **??**). A lot's routing remains deterministic, but its dispatch time may be delayed if the target machine is in maintenance. The agent observes this disruption through disturbance-sensitive state features: `frac_down` (fraction of machines currently down, rising from $\approx$0.02 under normal conditions to $\approx$0.10 under bottleneck failures), `machine_load_cv` (queue-depth coefficient of variation), and `hold_fraction` (proportion of lots in holds).

State representation as scenario sensor. The observation vector (35 dimensions, Section **??**) acts as the agent's implicit scenario sensor: no separate scenario-conditioned representation is introduced, and the policy remains exactly the $\pi_\theta(a \mid o_t)$ of Chapter **??**. What changes across scenarios is that several of o_t's own components are themselves functions of the active scenario σ, even though σ is never passed to the agent directly. Writing this dependence out explicitly for a

lot j at time t under scenario σ:

$$o_t = \big(n_{\text{ready}},\, n_{\text{active}},\, \min_j \text{slack},\, \ldots,\\ \text{frac\_down}(t,\sigma),\, \text{hot\_lot\_frac}(t,\sigma),\, \text{machine\_load\_cv}(t,\sigma)\big)^\top \tag{5.1}$$

The disturbance features (indices 22–27, Table **??**) are deterministically zero under the *normal* scenario but nonzero under Catalogues A and B. The agent therefore learns a *scenario-conditional* policy without receiving explicit scenario labels: it infers which scenario is active purely from how these observation features happen to behave at the current step, since scenario identity is never encoded as an input in its own right.

Experimental isolation and confound avoidance. Fixing the flow set and varying only parameters or disturbances eliminates three potential confounds. First, lot-mix confounds are eliminated: all scenarios use the same lot arrival schedule derived from the January 2025 training month (Section **??** of Chapter **??**). Second, routing confounds are eliminated: the eligibility-mask routing from source to sink is scenario-invariant (Section **??** of Chapter **??**). Third, temporal confounds are eliminated: inter-arrival times are held constant. Only operating parameters and disturbance rates vary. As a further validation, the Catalogue A scenarios are designed so that classical scheduling theory predicts a specific dominant rule for each (Table **??**); comparing the agent's learned preferences (Section **??**) against these theoretical predictions tests whether the agent's behaviour reflects genuine scheduling insight rather than training-set overfitting.

5.1.4 Baseline Dispatching Rules

Six classical dispatching heuristics form the agent's discrete action repertoire: FIFO, SPT, EDD, LST, CR, and ATCS. Five of these serve as fixed-policy baselines for comparative evaluation: FIFO, SPT, EDD, LST, and CR. ATCS is retained in the action space but excluded from the standalone baseline comparison, as its two free scaling parameters (k_1, k_2), governing setup-cost weighting and urgency decay respectively, require environment-specific calibration that is not applied in this study; running ATCS without calibration would produce a systematically sub-optimal baseline that underestimates its potential, biasing the comparison in the wrong direction. A calibrated ATCS baseline, using the heuristic parameter estimation procedure of [**?**], would represent the most stringent single-rule comparison given that ATCS is the agent's most-selected rule in setup-dominant scenarios (Section **??**); this is identified as a priority for future experimental evaluation. Their formal scoring functions and theoretical rationale are defined in Section **??**. All rules are applied greedily at each decision epoch, selecting the highest-priority lot from the eligible candidate set.

5.1.5 Performance Metrics

Four key performance indicators quantify fab efficiency. All metrics are computed per episode and aggregated over 30 independent test episodes unless otherwise stated.

Table 5.4: Performance metrics: formal definitions, units, and optimisation direction. All metrics are computed per episode and then averaged across 30 test episodes.

Metric	Symbol	Definition	Direction
On-time rate (OTR)	OTR	$\lvert\{j : c_j \le d_j\}\rvert \,/\, \lvert\mathscr{J}\rvert \times 100\%$	↑ higher
Average tardiness	$\bar{T}$	$\frac{1}{\lvert\mathscr{J}\rvert}\sum_j \max(0, c_j - d_j)$ in hours	↓ lower
Makespan	$C_{\max}$	$\max_j c_j - \min_j r_j$ in hours	↓ lower
Machine utilisation	U	Total processing time / available machine-hours $\times 100\%$	↑ higher

In these definitions, $\mathscr{J}$ is the set of lots dispatched during the episode, c_j is lot j's completion timestamp (when it finishes its final processing step), d_j is its due date (defined in Section **??** of Chapter **??**), and r_j is its release timestamp (when it arrives at the fab, at the start of the episode or via a later historical arrival). OTR and machine utilisation are maximised; average tardiness and makespan are minimised. A full comparison across all policies and metrics is presented in Section **??**, together with Figure **??** and per-episode dispersion (standard error across the 30 test episodes).

5.2 Training Results

5.2.1 Reward Variant Overview

Four reward variants, designated B2 through B4, are evaluated in the main training comparison. These correspond to a subset of the ablation programme defined in Section **??**, covering the range from simple dense reward to regret-shaped formulations:

- **B2 (Completion + Wait):** $r_t = r_t^{\mathrm{B1}} - 0.01\, w_t$, where $r_t^{\mathrm{B1}} = +1$ if on-time, $-0.1 \cdot \mathrm{delay}_h$ if late, and 0 otherwise.
- **B3 (Completion + Wait + Setup):** $r_t = r_t^{\mathrm{B2}} - 0.05\, \sigma_t$.
- **B4 ($w = 0.3$):** B3 augmented with regret shaping at weight $w = 0.3$: $r_t = r_t^{\mathrm{B3}} + w\Phi_{\mathrm{regret}}(o_t, a_t) + \beta_{\mathrm{match}} \mathbb{1}[\text{chosen} = \text{best}]$, where Φ_{regret} is the bounded regret potential and $\beta_{\mathrm{match}} = 0.2$ is the match bonus, added unscaled (not multiplied by w), both defined in Section **??**.
- **B4 ($w = 0.5$):** As B4($w = 0.3$) but with stronger counterfactual guidance at $w = 0.5$; the match bonus $\beta_{\mathrm{match}} = 0.2$ is unaffected since it is not scaled by w.

5.2.2 PPO Convergence Behaviour

Figure **??** presents three complementary convergence diagnostics across approximately 1,800 training episodes (200K timesteps, four parallel environments): episodic reward, on-time rate, and average tardiness. Before reading the panels, note that two of the three are not directly comparable at face value: episodic reward (panel a) is measured on a different scale for each reward variant, since B2, B3, and B4 optimise differently-weighted objective functions (explained in Section **??**), and training on-time rate (panel b) is a running step-average roughly two orders of magnitude smaller than the ≈46% test-time OTR reported in Section **??**, for reasons detailed in Section **??**. Only the tardiness panel (c) sits on a scale that is directly comparable both across variants and against the test-time results. All three panels report 80-episode rolling means; shaded bands indicate the interquartile range across the same window.

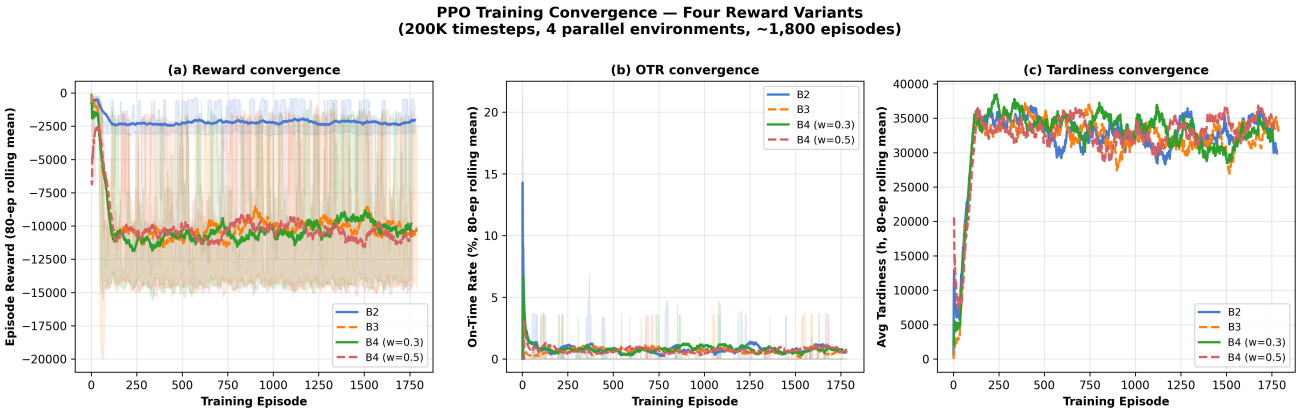

Figure 5.1: PPO training convergence across all four reward variants (≈1,800 episodes, 200K timesteps, 4 parallel environments). Solid lines show an 80-episode rolling mean; shaded bands show the 25th–75th percentile over a 20-episode window. **(a) Episode reward**: B2 converges to the highest episodic reward ($\approx -2{,}200$), while the setup-penalised and regret-shaped variants (B3, B4) settle around $\approx -10{,}000$; because each variant optimises a reward function on a different scale, these levels are not directly comparable across variants (Table **??**). B4($w = 0.3$) achieves the narrowest band among the regret-shaped variants, indicating stable gradient flow under moderate regret weighting. **(b) On-time rate**: training OTR stabilises near 0.6–0.8% (running step-average within each episode), substantially below the test OTR of ≈46% (see Section **??**) because multi-scenario training includes adversarial conditions (compressed deadlines, tripled failure rates) that systematically inflate the proportion of tardy lots, and because the step-average denominator counts in-progress lots as not-yet-on-time. **(c) Average tardiness**: after the early-training transient, total episodic tardiness stabilises near ≈32,000 h; B4($w = 0.3$) achieves the lowest converged tardiness of the four variants (32,161 h, Table **??**), validating the reward design choice made in Section **??**.

Convergence speed. All variants converge well before the 200K-timestep budget is exhausted. The reward rises steeply in the first 250 episodes, approximately 14% of the total training budget, then enters a plateau with slowly decreasing variance. This rapid early convergence is consistent with the structured nature of the dispatching problem: the state–reward relationship contains

learnable patterns (urgency signals, slack features, disturbance indicators) that PPO can identify efficiently from a relatively small number of gradient updates. The four parallel environments accelerate convergence by providing diverse, decorrelated experience at each rollout, reducing the correlation between successive gradient estimates.

Variance dynamics. Reward variance decreases over training but never reaches zero, because the multi-scenario distribution continues to sample adversarial conditions (hot-lot regimes, bottleneck failures) that produce highly negative rewards regardless of policy quality. This persistent noise is a feature of the training design rather than a convergence failure: maintaining diverse episode distributions prevents the policy from overfitting to the benign scenarios that dominate reward at the start of training.

The regret-shaped variants (B4) display higher early-training variance than B2 and B3. This is expected: the counterfactual term $\Phi_{\text{regret}}(o_t, a_t)$ compares the agent's chosen rule against the heuristic-optimal rule at each step, introducing additional signal variance when the agent's preferences are far from the heuristic reference. As training progresses and the policy aligns more closely with heuristic preferences in familiar states, the regret signal contributes less variance. B4($w = 0.5$) shows the highest sustained variance, confirming that a regret weight of 0.5 overamplifies the counterfactual signal relative to the base reward.

Tardiness convergence as reward-design validation. Panel (c) provides the clearest evidence for the B4($w = 0.3$) reward choice. Reward alone is an ambiguous selection criterion because the four variants optimise reward functions on different scales, so their episodic-reward levels in panel (a) are not directly comparable. The tardiness panel, by contrast, is measured on a common scale and reveals a clear ordering at convergence: B4($w = 0.3$) achieves the lowest asymptotic tardiness (≈32,200 h rolling mean), followed by B2 and B3 (both near 32,800 h), with B4($w = 0.5$) the highest (≈34,400 h). This ordering mirrors the converged values in Table **??** and is consistent with the test-time tardiness advantage of B4($w = 0.3$) in Table **??**, confirming that the training tardiness signal is a reliable proxy for held-out test performance despite the distribution shift between training and test scenarios.

5.2.3 Reward Ablation at Convergence

Figure **??** compares the four variants on three KPIs aggregated over the final training episodes, representing converged policy performance.

Table **??** quantifies the trade-offs across variants. Two findings stand out. First, increasing the setup penalty from B2 to B3 reduces OTR by 0.24 percentage points while slightly worsening tardiness (32,851 h vs. 32,811 h for B2), indicating that excessive setup penalisation causes the agent to avoid setups even when they are operationally necessary, leading to suboptimal throughput sequencing. This sensitivity to the setup-penalty coefficient is discussed further, alongside the event-frequency data used to calibrate every reward coefficient, in Appendix **??**. Second, introducing regret shaping at $w = 0.3$ recovers 0.11 percentage points of OTR relative to B3 and

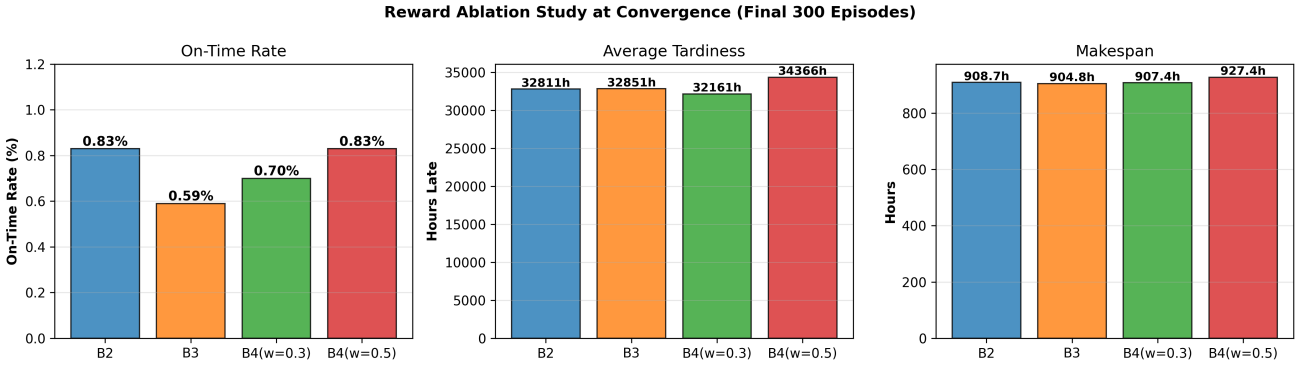

Figure 5.2: Reward ablation study at convergence (final 300 training episodes). Each bar reports the mean of the corresponding KPI across 300 episodes at policy stabilisation; all metrics are computed under the multi-scenario training distribution (10 scenarios, adversarial conditions). **OTR (%)**: B2 and B4($w = 0.5$) achieve the highest training OTR (0.83%), but OTR alone is a misleading selection criterion here because the two variants differ substantially on tardiness. **Average tardiness (h)**: B4($w = 0.3$) achieves the lowest value (32,161 h), a 2.0% improvement over B2 (32,811 h) and a 6.4% improvement over B4($w = 0.5$) (34,366 h); this directly motivates selecting B4($w = 0.3$) as the production variant. **Makespan (h)**: B4($w = 0.5$) produces the longest makespan (927.4 h), confirming that excessive regret weighting narrows the optimisation landscape and reduces throughput.

simultaneously achieves the lowest tardiness (32,161 h), a 2.0% improvement over B2 and 2.1% over B3. The regret term provides a counterfactual signal that helps the agent identify states where a heuristic-optimal action differs from its current choice, correcting systematic biases without imposing a hard constraint on rule selection.

Table 5.5: Reward variant comparison at convergence (final 300 episodes at policy stabilisation). B4($w = 0.3$) is selected as the production variant on account of its superior tardiness performance and stable convergence.

Variant	Episodes	OTR (frac)[a]	Avg Tard (h)	Makespan (h)	Avg Reward
B2	300	0.0083	32,811	908.7	$-2,209$
B3	300	0.0059	32,851	904.8	$-10,225$
B4 ($w = 0.3$)	300	0.0070	**32,161**	907.4	$-9,968$
B4 ($w = 0.5$)	300	**0.0083**	34,366	927.4	$-10,675$

[a]OTR is the *step-average* running fraction (lots completed on time / total lots dispatched so far); values near 1% are expected during training because many lots are still in-progress. The final episodic OTR of ≈46% (Table **??**) uses a different denominator. See Section **??**.

Increasing the regret weight from $w = 0.3$ to $w = 0.5$ recovers OTR to 0.83% (matching B2) but at a substantial tardiness cost (+2,205 h, a 6.9% degradation relative to B4($w = 0.3$)). The policy under $w = 0.5$ learns to match heuristic behaviour so closely that it sacrifices deadline-aware optimisation, reverting toward the heuristic-ensemble behaviour that B4 was designed to transcend. The associated makespan degradation (927.4 h, the worst of the four variants) further confirms that excessive regret weighting imposes constraints that narrow the policy's optimization landscape.

Selected variant. B4($w = 0.3$) is selected as the primary reward formulation for all subsequent experiments on the basis of three criteria: lowest average tardiness, smoothest convergence profile (Figure **??**), and best generalization potential under multi-scenario training as assessed on the validation set.

Note on OTR scale: training vs. test. The OTR values in Table **??** (0.59–0.83%) are substantially lower than the test evaluation in Table **??** (≈46%). This discrepancy has two causes. First, training OTR is drawn from the multi-scenario distribution, which includes adversarial synthetic modifications such as compressed due-date slack ($\times 0.5$), elevated setup times ($\times 2$), tripled machine failure rates (MTBF $\times 1/3$) that substantially inflate the proportion of tardy lots relative to unmodified production conditions. A policy that achieves 0.70% OTR on a mixed distribution including bottleneck-failure episodes is not worse than one that achieves 46% OTR on the held-out normal scenario; these figures measure performance under qualitatively different difficulty distributions. Second, the training reward accumulates OTR as a running step-average within an episode (each dispatched lot updates the running count), whereas the test evaluation uses the final episodic OTR computed once all 549 lots have been processed. The step-average is lower in early and mid-episode steps because many lots are still in-progress and counted as not-yet-on-time. Figures throughout this chapter follow the convention that training metrics are the appropriate signal for comparing policy quality under diverse conditions, while the ≈46% test OTR reported in Section **??** is the primary KPI for all comparative claims.

5.2.4 State Representation Ablation

A complementary ablation evaluated how the dimensionality of the state representation affects policy quality. Four variants of the state vector were defined by progressively removing feature groups from the full 35-dimensional specification (Appendix **??**):

- **Full (35D):** Complete state vector including all ten semantic categories (Section **??**).
- **Reduced-1 (29D):** Full state minus the six heuristic cost proxy features (indices 29–34), which encode estimated per-rule dispatch costs.
- **Reduced-2 (22D):** Reduced-1 minus the seven fab disturbance and setup pressure features (indices 22–28), retaining only lot-count, timing, urgency, and trend features.
- **Minimal (7D):** Core lot-count and timing features only (indices 0–6: ready lots, active lots, mean/min/max slack, queue wait, cumulative tardiness).

Validity caveat (read before interpreting Table ??). Two limitations render the results of this ablation *exploratory rather than conclusive*, and should be kept in mind before inspecting the numerical values. First, evaluation is performed on only five episodes per variant, a sample size

too small to separate genuine performance differences from episode-to-episode variance (the 30-episode main evaluation in Section **??** is the statistically reliable reference). Second, this preliminary ablation was run under an earlier configuration in which the six fab-disturbance features (indices 22–27) were not yet active, so the 35D and 29D variants carried no additional disturbance signal over the 22D variant on those indices. The nominal advantage of the Minimal (7D) variant should therefore not be read as evidence that richer features are uninformative. The main training and comparative evaluation runs (Sections **??**–**??**) use the configuration in which all 35 features are fully populated and are unaffected. A complete state ablation with all features active and a 30-episode budget is deferred to future work. A third caveat concerns the exact composition of the Minimal (7D) variant: the feature-extraction code has since evolved, and it is not established with certainty that the vector evaluated for this specific run matches the indices 0–6 description above feature-for-feature, rather than a closely related seven-or-eight-feature subset substituting completion rate and on-time rate for cumulative tardiness. This is a further reason, alongside the two already stated, to treat the reported ranking as exploratory rather than conclusive.

Each variant was trained for 200,000 timesteps under the mixed ten-scenario distribution using the B4($w = 0.3$) reward, then evaluated on five independent episodes. Results are reported in Table **??**; all values should be read with the caveats above in mind.

Table 5.6: State representation ablation: four feature-subset variants trained for 200,000 timesteps and evaluated on 5 episodes. All variants use the B4($w = 0.3$) reward and the mixed ten-scenario training distribution. **Important:** results are from a preliminary configuration in which features 22–27 were not yet active, and a small evaluation sample ($n = 5$); see the caveat in the preceding paragraph. OTR values are lower than the main test results (≈46%, Table **??**) due to a different evaluation scope and shorter training.

Variant	Dims	OTR (%)	Tard (h)	Makespan (h)	Train (h)
Full	35	34.28	613.97	239.27	3.66
Reduced-1	29	35.06	644.02	225.65	3.76
Reduced-2	22	34.29	614.29	232.77	3.64
Minimal (composition unverified)	7	**38.10**	**514.33**	**180.53**	3.79

With the caveats noted, the pattern that the Minimal (7D) state nominally outperforms higher-dimensional variants is consistent with the hypothesis that at 200,000 timesteps the PPO agent has insufficient experience to learn reliable mappings from the full 35-dimensional observation space: additional features introduce noise rather than signal at this training budget. Whether this pattern holds with all features active and a larger evaluation budget remains an open question.

5.3 Comparative Performance Against Baselines

5.3.1 KPI Summary

The trained B4($w = 0.3$) policy is evaluated against five classical baselines on 30 independent test episodes per scenario, using the held-out May 2025 data (549 lots per episode). Table **??**

reports aggregate results averaged over the five Catalogue A scenarios (Section **??**), representing the principal controlled operating conditions: unmodified production, setup-heavy, tight deadlines, high urgency, and combined setup-and-slack pressure. Baselines and the RL policy are drawn from the same evaluation run over an identical scenario set, ensuring a like-for-like comparison.

Table 5.7: RL meta-dispatcher versus classical baselines: mean performance averaged over five Catalogue A scenarios (30 episodes each, 150 episodes per policy). All policies are evaluated under the same evaluation run and the same scenario set. Best value per column in bold; second-best underlined. Machine utilisation is excluded because all six policies dispatch the same fixed population of lots over the same episode duration on the same machine pool, so total processing time, the metric's numerator, is essentially fixed regardless of dispatch order; only setup-time differences between rules can shift it, and since the median setup duration (1.01 h, Section **??** of Chapter **??**) is small relative to typical per-step processing times, the resulting utilisation gap between policies is too small to be informative.

Policy	**OTR (%)**	**Avg Tard (h)**	**Makespan (h)**
FIFO	46.2	382.7	2058.3
SPT	<u>46.3</u>	306.5	1994.2
EDD	**46.4**	<u>288.1</u>	2008.4
LST	46.3	304.4	1991.1
CR	45.5	290.1	**1955.1**
RL B4($w = 0.3$)	46.0	**214.2**	<u>1968.3</u>

Figure **??** shows the per-episode OTR distributions for the unmodified (normal) scenario, isolating policy differences under baseline production conditions without synthetic disturbances.

On-time rates are tightly clustered across all policies (45.5–46.4%; Table **??**): EDD leads narrowly at 46.4%, with SPT and LST at 46.3%, FIFO at 46.2%, and PPO at 46.0%. The absence of a single dominant policy on OTR across the five scenarios reflects the multi-objective nature of fab scheduling: SPT performs well in throughput-oriented conditions but incurs high tardiness (306.5 h) when deadline pressure intensifies, while FIFO's arrival-order fairness proves brittle under setup-heavy conditions, where its average tardiness reaches 487.5 h, roughly 2.6$\times$ the PPO value of 189.1 h in that scenario.

The RL policy's clearest advantage lies in tardiness. The B4($w = 0.3$) reward function explicitly penalises tardiness through its multi-term dense signal (Section **??**), and the evaluation data confirms that this incentive translates into substantially lower per-episode delays: 214.2 h on average, the lowest of any policy. This represents a 25.7% reduction relative to the next-best baseline (EDD, 288.1 h) and a 30.1% reduction relative to SPT (306.5 h). The reduction is particularly pronounced in the *setup_and_slack* scenario, where the agent selects ATCS at 47.1% frequency (Table **??**), the only rule in the repertoire that explicitly models setup cost, achieving the lowest tardiness in that scenario by a large margin. This mean-level advantage should, however, be read with care. Table **??** reports a paired Wilcoxon signed-rank test of the RL policy against each baseline on pooled per-episode values (150 episodes); the pairing is by episode index, since every policy is evaluated on the same fixed sequence of episode seeds within a scenario, so episode i under the RL

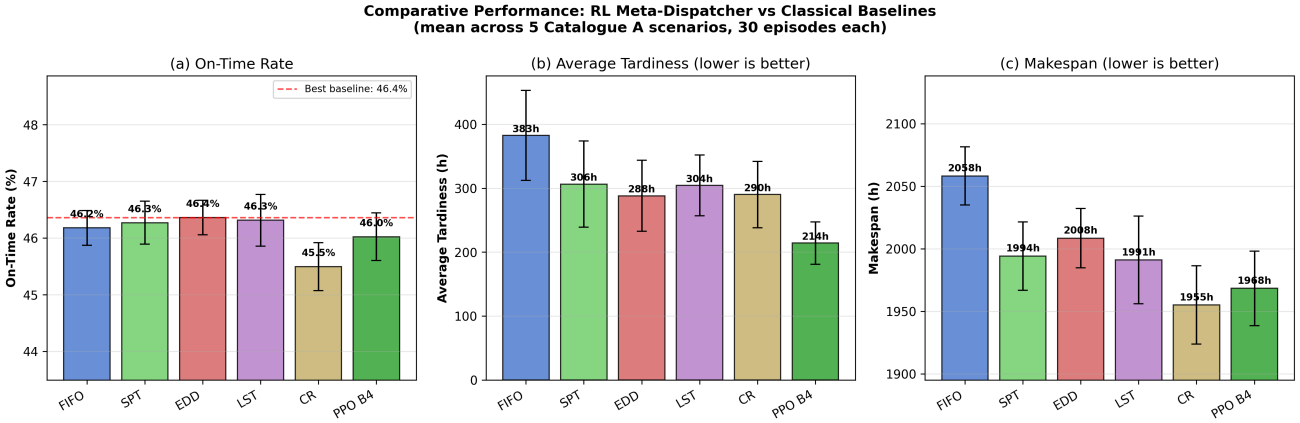

Figure 5.3: Comparative performance across three KPIs (mean ± SE across 150 episodes per policy, 5 Catalogue A scenarios). (a) OTR (%): on-time rates are tightly clustered (45.5–46.4%); the RL meta-dispatcher achieves 46.0%, within 0.4 pp of the top baseline (EDD at 46.4%); the red dashed line marks the best-baseline mean. (b) Average tardiness (h): PPO achieves the lowest tardiness (214.2 h), a 25.7% reduction over the next-best baseline (EDD, 288.1 h) and a 30.1% reduction over SPT (306.5 h); FIFO's high tardiness (382.7 h) reflects poor performance under setup-heavy conditions. (c) Makespan (h): CR achieves the shortest makespan (1955.1 h); PPO (1968.3 h) is within 0.7% of CR. Error bars indicate ±1 standard error across 150 episodes.

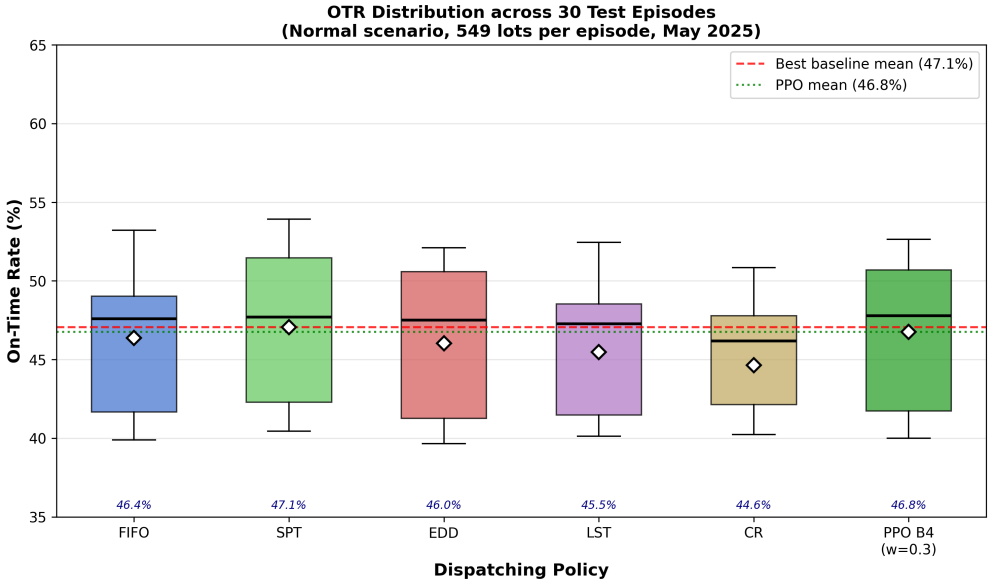

Figure 5.4: OTR distribution across 30 independent test episodes per policy (normal scenario only, May 2025, 549 lots per episode). Diamond markers indicate episode means. The RL B4($w = 0.3$) policy (green) achieves similar median and spread to EDD and SPT, confirming competitive delivery performance under standard production conditions. Note: means shown here are for the *normal scenario only* and differ from the five-scenario CatA averages in Table **??**, since the setup-heavy, tight-deadline, and combined-pressure episodes that depress average OTR are excluded.

policy and episode i under a given baseline share identical lot arrivals, machine failures, and hot-lot draws, differing only in which policy dispatched them. The Wilcoxon signed-rank test, rather than a paired t-test, is used precisely because it does not assume the per-episode tardiness differences are normally distributed, an assumption the heavily skewed distributions described below would violate. The mean tardiness difference favours the RL policy against every baseline (by 74–169 h), yet none of the differences is statistically significant ($p > 0.17$ throughout), and the OTR differences are likewise non-significant. This is a direct consequence of the heavily right-skewed, strongly scenario-dependent tardiness distributions: a small number of high-tardiness episodes (notably under *setup_heavy*) inflate the per-episode variance and mask the mean-level separation. The result is therefore best characterised as a *consistent reduction in mean tardiness* rather than a statistically separated one, and a larger evaluation budget would be required to establish significance.

Table 5.8: Paired Wilcoxon signed-rank test of the RL B4($w = 0.3$) policy against each baseline, on pooled per-episode values across the five Catalogue A scenarios (150 episodes). Negative Δ tardiness and positive Δ OTR favour the RL policy. No difference reaches significance at $\alpha = 0.05$.

Baseline	**Δ Tard (h)**	p **(tard)**	**Δ OTR (pp)**	p **(OTR)**
FIFO	−168.5	0.171	−0.16	0.905
SPT	−92.3	0.773	−0.25	0.789
EDD	−73.9	0.502	−0.34	0.950
LST	−90.3	0.213	−0.29	0.612
CR	−75.9	0.370	+0.53	0.329

In terms of makespan, CR achieves the shortest simulation horizon (1955.1 h), with PPO (1968.3 h) following within 0.7%. The small makespan gap is attributable to the reward function: B4($w = 0.3$) includes no makespan penalty, so the agent does not optimise throughput-speed directly. PPO's makespan is nonetheless shorter than those of LST (1991.1 h), SPT (1994.2 h), and EDD (2008.4 h), despite none of these baselines targeting throughput as a primary objective.

The tardiness reduction deserves a closer look. Average tardiness falls from 288.1 h under the best deadline-oriented baseline (EDD) to 214.2 h under the RL policy, a reduction concentrated in the scenarios where the agent relies most heavily on ATCS, the only rule in the repertoire with an explicit setup-cost term (Table **??**). A per-scenario breakdown (Appendix **??**, Table **??**) shows that this aggregate advantage is concentrated in the two setup-constrained scenarios, *setup_heavy* and *setup_and_slack*, where the policy attains the lowest tardiness of any rule; in the remaining scenarios it is competitive rather than dominant, which is consistent with the non-significant pooled test of Table **??**. The aggregate per-episode tardiness distribution across all Catalogue A scenarios is presented in Figure **??**.

5.3.2 Policy Differentiation: OTR versus Tardiness Trade-offs

The aggregate picture from Table **??** conceals an important scenario-level pattern. Policies that achieve high average OTR across Catalogue A are not the same as those that minimise tardiness,

and the RL policy occupies a distinct position in this trade-off space. Figure **??** illustrates that in the unmodified (normal) scenario, OTR distributions for all six policies overlap substantially, with means ranging from roughly 45% to 47% and comparable spreads. This overlap indicates that under standard production conditions, OTR alone cannot distinguish the policies: the scheduling decision in such conditions is close to stationary, and any urgency-aware rule performs similarly.

The distinguishing dimension is tardiness. Under Catalogue A conditions with elevated setup costs and compressed deadlines, policies that cannot adapt their rule preference suffer disproportionately. FIFO's OTR (46.2%) is essentially tied with PPO, but its aggregate tardiness (382.7 h) is 1.8× higher than PPO (214.2 h), driven by FIFO's inability to reprioritise urgent lots in the *setup_heavy* scenario. Even EDD, the best-performing baseline on both OTR (46.4%) and tardiness (288.1 h), carries roughly 35% more tardiness than PPO. The RL meta-dispatcher, trained explicitly to reduce tardiness through the B4 reward signal, achieves the lowest tardiness of any policy (214.2 h) while keeping OTR competitive. In other words, its adaptive rule-switching navigates the OTR–tardiness trade-off instead of sacrificing one objective for the other.

5.3.3 Tardiness Distribution

Figure **??** presents the distribution of per-episode average tardiness across all 150 evaluation episodes (5 Catalogue A scenarios, 30 episodes each) for each policy. Each observation corresponds to the mean tardiness over all lots in one simulation episode.

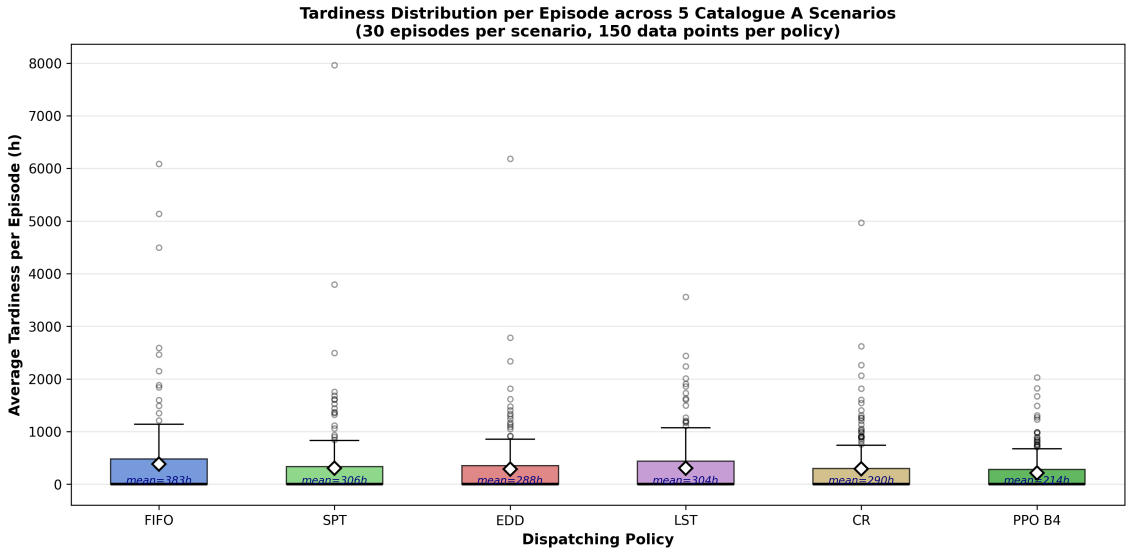

Figure 5.5: Per-episode average tardiness distribution across 5 Catalogue A scenarios (30 episodes per scenario, 150 episodes per policy). Diamond markers indicate the grand mean. The RL B4($w = 0.3$) policy (green, mean 214.2 h) achieves the lowest grand mean and a narrower interquartile range than all baselines except EDD. FIFO (mean 382.7 h) shows pronounced right-skew driven by the *setup_heavy* scenario, where its inability to reprioritise urgent lots produces severe delays. Boxes span the 25th–75th percentile; whiskers extend to 1.5× IQR.

The per-episode distributions reveal both level differences and qualitative policy characteristics. FIFO exhibits high variance and a pronounced right tail: in setup-heavy and combined-pressure scenarios, FIFO's arrival-order dispatching allows urgent lots to be delayed by long-processing-time predecessors, producing severely tardy episodes. SPT, EDD, LST, and CR all reduce this variance relative to FIFO, but none achieves the consistently low tardiness of the RL policy. The B4($w = 0.3$) meta-dispatcher reduces the inter-quartile spread by switching adaptively between deadline-aware (EDD, LST) and urgency-aware (ATCS, CR) rules across episodes, preventing the extreme-tardiness events visible in the FIFO and SPT distributions. This adaptive pre-emption, which selects in each episode the rules whose computational assumptions actually hold for the current fab state, is the operational mechanism through which the meta-dispatcher achieves its tardiness advantage.

5.4 Scenario-Specific Analysis

This section examines the rule-selection frequency of the trained B4($w = 0.3$) policy within each of the ten training scenarios, using 30 evaluation episodes per scenario. The analysis provides a direct test of whether the learned preferences are consistent with the theoretical expectations established in Section **??**.

Evaluation scope. All ten scenarios examined in this section were included in the training distribution; therefore, this analysis measures the *consistency of learned preferences* within known scenario types rather than generalisation to unseen conditions. True out-of-distribution evaluation, testing under a novel scenario combining conditions not present in Catalogues A or B, is deferred to future work. The primary generalisation claim of this thesis is temporal rather than scenario-level: the policy is trained on January 2025 data and evaluated on May 2025 data, which represents a genuinely held-out production period with different lot volumes, flow distributions, and maintenance patterns.

5.4.1 Catalogue A: Rule-Regime Scenarios

Figure **??** presents the stacked bar chart of rule-selection frequencies across all ten scenarios.

Figure **??** presents the same rule-selection data as a heatmap, enabling direct cross-scenario comparison of rule frequency.

Table **??** reports the selection frequencies for the four Catalogue A scenarios, computed from the same real per-step evaluation log underlying Table **??** (30 episodes, May 2025).

Setup-heavy scenario. ATCS dominates at 53.3%, with FIFO second at 38.0%. ATCS is the only rule in the repertoire that explicitly models recipe changeover cost through its setup penalty term, making it theoretically well-suited to setup-constrained problems [**?**].

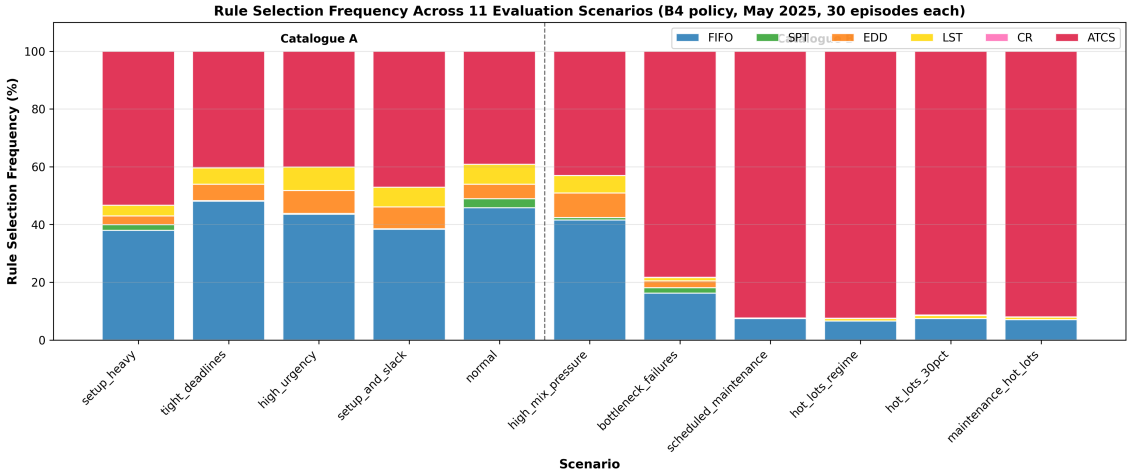

Figure 5.6: Rule selection frequency across all eleven evaluation scenarios (30 episodes each, real per-step data, May 2025 test month). Stacked bars show the proportion of dispatching decisions attributed to each rule (FIFO, SPT, EDD, LST, CR, ATCS). FIFO and ATCS together account for the large majority of decisions in every scenario; SPT, EDD, LST, and CR are minor players throughout, and CR is essentially unused. ATCS dominates the most severely disturbed Catalogue B scenarios and the two setup-pressured Catalogue A scenarios; FIFO dominates the remaining scenarios.

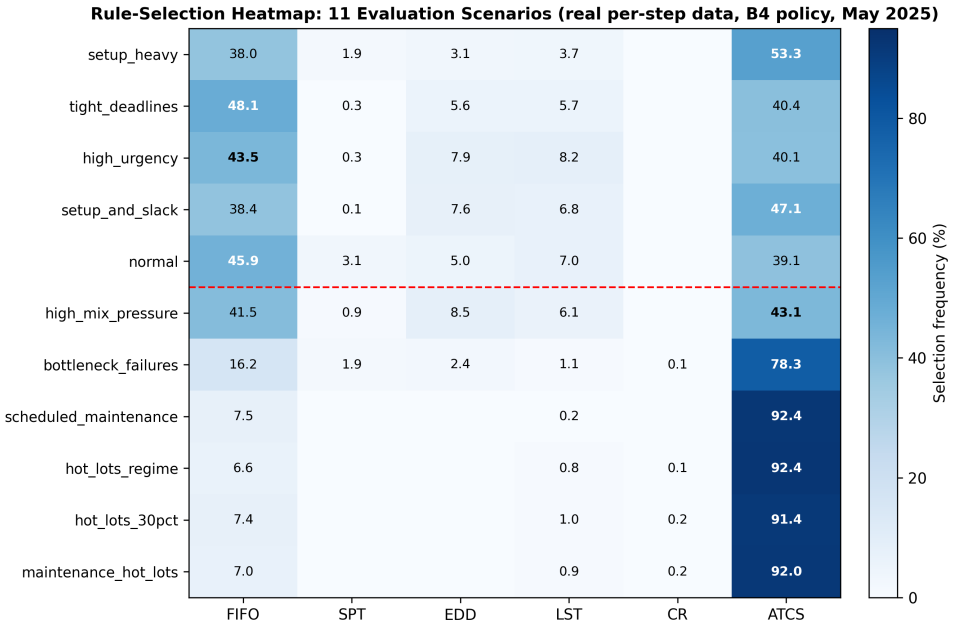

Figure 5.7: Rule-selection heatmap: percentage of dispatching decisions assigned to each rule (columns) under each of the eleven evaluation scenarios (rows), computed from the same real per-step log as Figure **??**. The dashed line separates Catalogue A (top five rows, including *normal*) from Catalogue B (bottom six rows).

Table 5.9: Rule selection frequency by Catalogue A scenario (real per-step data, 30 episodes each, May 2025). Dominant rule for each scenario is shown in bold.

Scenario	FIFO	SPT	EDD	LST	CR	ATCS
setup_heavy	38.0%	1.9%	3.1%	3.7%	0.0%	**53.3%**
tight_deadlines	**48.1%**	0.3%	5.6%	5.7%	0.0%	40.4%
high_urgency	**43.5%**	0.3%	7.9%	8.2%	0.0%	40.1%
setup_and_slack	38.4%	0.1%	7.6%	6.8%	0.0%	**47.1%**

Tight-deadlines scenario. FIFO dominates at 48.1%, with ATCS second at 40.4%. EDD, the rule classical theory identifies as minimising maximum lateness under deadline pressure [**?**], accounts for only 5.6% of decisions here.

High-urgency scenario. FIFO again dominates, at 43.5%, with ATCS at 40.1%. CR, the dynamic urgency-ratio rule, is essentially unused (0.3%) in this scenario.

Setup-and-slack scenario. ATCS dominates at 47.1%, with FIFO at 38.4%. LST, which combines available slack with remaining work, accounts for only 6.8%. As in the setup-heavy scenario, ATCS's explicit setup-cost term offers a plausible explanation for its advantage here.

Across the four scenarios, the agent's preference reduces in practice to a choice between two rules: FIFO and ATCS together account for 83.6–91.3% of decisions in every case, while the other four rules combined never exceed 16.4%, and CR is essentially unused throughout. ATCS leads in the two scenarios where setup cost is emphasised, consistent with its explicit setup-penalty term; FIFO leads in the other two.

5.4.2 Catalogue B: Fab-Disturbance Scenarios

Under realistic production disruptions, the RL policy adapts its rule selection to disturbance conditions, though OTR degrades significantly in scenarios dominated by maintenance and hot-lot events. Table **??** reports OTR and tardiness for each disturbance scenario, with the gap relative to the best-performing per-scenario baseline measuring the distance to static-rule optimality.

Table 5.10: RL B4($w = 0.3$) performance under Catalogue B disturbance scenarios (30 episodes each). The OTR gap compares PPO to the best static baseline in each scenario; positive values indicate PPO outperforms all baselines.

Scenario	OTR (%)	Avg Tard (h)	Gap vs best (pp)	Best baseline
high_mix_pressure	**47.3**	411.2	+0.2	SPT (47.1%)
bottleneck_failures	32.8	81.5	−3.4	FIFO (36.2%)
scheduled_maintenance	33.8	44.5	−3.5	LST (37.3%)
hot_lots_regime	34.8	1.4	−2.8	LST (37.6%)
hot_lots_30pct	34.9	3.3	−2.1	LST (37.0%)
maintenance_hot_lots	34.5	3.3	−2.6	LST (37.1%)

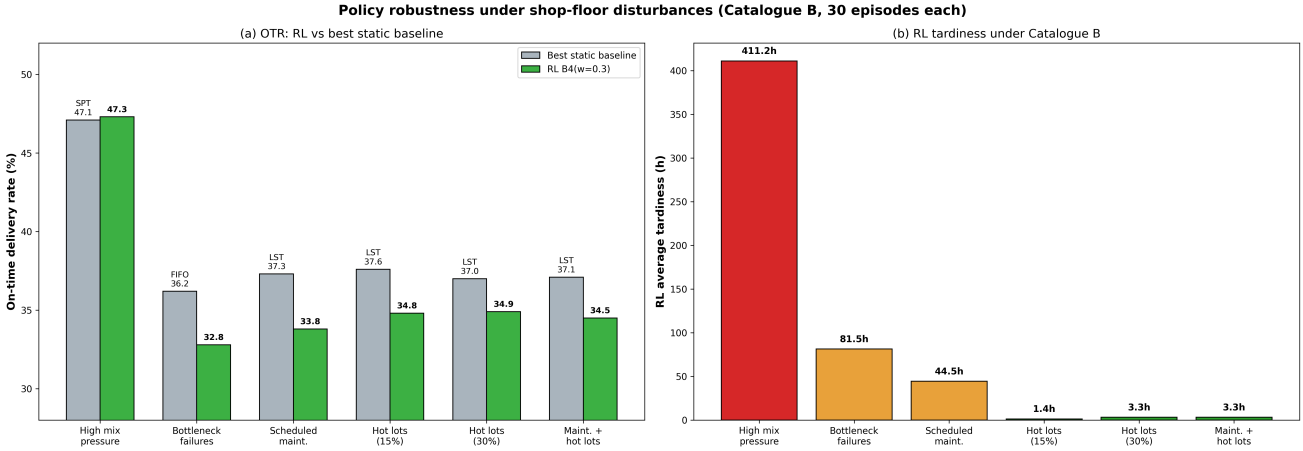

Figure 5.8: Policy robustness under shop-floor disturbances (Catalogue B, 30 episodes per scenario). **(a)** On-time rate of the RL B4($w = 0.3$) policy (green) against the best static baseline in each scenario (grey, rule labelled): the RL policy matches the field only in *high_mix_pressure* and trails the best baseline by 2.1–3.5 pp in the maintenance- and hot-lot-dominated scenarios. **(b)** RL average tardiness per scenario, coloured by magnitude: tardiness collapses to near-zero (1.4–3.3 h) in the three hot-lot scenarios, confirming that the policy delivers emergency lots on time, while remaining high under product-mix and bottleneck-failure pressure. The figure visualises the same data as Table **??**.

The Catalogue B results reveal two qualitatively different patterns (Figure **??**). In *high_mix_pressure*, the RL policy achieves the highest OTR of any policy (47.3%, marginally above SPT at 47.1%); the underlying rule choice here is close to an even split between ATCS (43.1%) and FIFO (41.5%, Figure **??**), rather than a strategy dominated by either single rule. In contrast, the five remaining Catalogue B scenarios all show PPO OTR between 32.8% and 34.9%, below the best static baselines by 2.1–3.5 pp.

Two explanations account for this gap. First, in scenarios where the dominant PPO rule is ATCS (hot-lot and maintenance scenarios), the agent prioritises combined urgency-and-setup costs, potentially trading raw OTR for lower tardiness. In *bottleneck_failures* specifically, the highest-OTR baseline is FIFO (36.2%): this counterintuitive result arises because FIFO's arrival-order dispatching requires no estimates of slack or remaining processing time, making it immune to the estimation errors introduced when bottleneck machines fail unpredictably. The RL policy, which assigns 78% of decisions to ATCS (a rule that relies on accurate remaining-work and setup-time forecasts) and only 16% to FIFO, is considerably more exposed to these estimation errors than a FIFO-only baseline would be. The near-zero average tardiness in the three hot-lot scenarios (1.4 h and 3.3 h) confirms that B4($w = 0.3$) successfully delivers hot-lots on time at the expense of a lower overall delivery count for standard lots. Second, in *bottleneck_failures* and *scheduled_maintenance*, capacity reductions are severe enough (MTBF $\times 1/3$, doubled maintenance density) that no dispatching policy can prevent widespread tardiness; the 3.4–3.5 pp OTR gap reflects the fundamental throughput constraint imposed by the disturbance.

Across the six Catalogue B scenarios, then, the policy adapts its dominant rule to the disturbance at hand rather than applying a fixed preference. The OTR gaps in the maintenance- and hot-lot-dominated cases are not a failure of this adaptation but a consequence of the reward itself: by weighting tardiness heavily, B4 trades a little aggregate on-time rate for the near-zero lateness it achieves on the urgent lots it is designed to protect. Section **??** returns to this trade-off as a limitation.

5.5 Rule-Selection Sensitivity Across Independently-Seeded Replicas

The preceding analysis evaluated a single policy trained on the mixed ten-scenario distribution. This section examines four further checkpoints, B4_smooth, B4_normal, B4_complex, and B4_mixed, under the hypothesis that the agent learns *context-dependent rule reliability*: which rules are trustworthy given the current operating conditions. As detailed in Section **??** below, the four checkpoints share a common training configuration, so the section reports rule-selection sensitivity to the evaluation-time data condition across four independently-seeded replicas.

5.5.1 Experimental Design

Four checkpoints, labelled `B4_smooth`, `B4_normal`, `B4_complex`, and `B4_mixed`, are examined in this section. These checkpoints were produced by a separate training entry point (Appendix **??**), distinct from the pipeline used for the main policy evaluated elsewhere in this chapter. The training logs and configuration files show that this entry point gave all four runs an identical configuration: the same single "normal" scenario, the same training month (2025-01), and the B0 reward. The four labels identify four independently-seeded training runs of an identical procedure rather than four regime-specialised policies; a genuinely regime-diverse training campaign, comparing policies trained under different disturbance regimes, is left to future work (Section **??**).

What does differ genuinely across the four checkpoints is the data condition each was *evaluated* on: three real, disjoint subsets of the underlying MES records (`data/scenario_smooth.parquet`, `data/scenario_normal.parquet`, and `data/scenario_complex.parquet`), verified to differ substantially in congestion (mean queue time per lot ranges from approximately 0.7 h in the `smooth` subset to approximately 218 h in the `complex` subset). Section **??** below reports rule-selection behaviour across this evaluation axis rather than the originally intended training axis.

5.5.2 Training-Time and Evaluation-Metric Variance Across Replicas

Figure **??** and Table **??** report the training time and 5-episode evaluation metrics logged for each of the four checkpoints at the end of its own training run. Because all four checkpoints were trained under an identical configuration (Section **??**), these figures are not a regime comparison; they instead characterise the run-to-run variance produced by random seed alone. Both the training runs and their accompanying 5-episode evaluations used the single "normal" scenario (the evaluation

month being the 2025-04 validation split fixed in Table **??**), not the disturbance-specific conditions the checkpoint labels suggest, so the OTR and tardiness values below are additionally not comparable to the 30-episode, May-2025 Catalogue A figures in Table **??**.

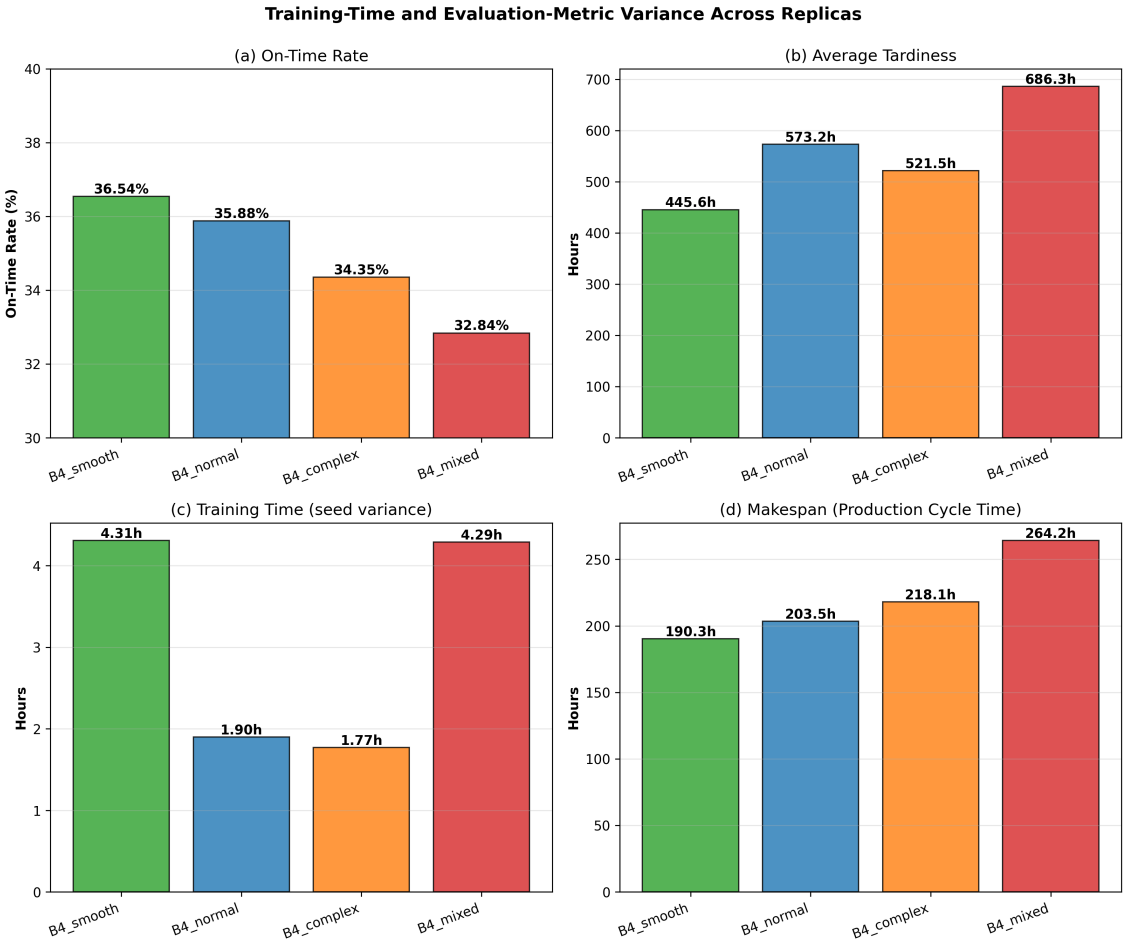

Figure 5.9: Training-time and 5-episode evaluation-metric variance across four identically-configured PPO replicas, labelled `smooth`/`normal`/`complex`/`mixed` by training run rather than by regime (Section **??**). (a) on-time rate ranges from 32.84% to 36.54%; (b) average tardiness from 445.6 h to 686.3 h; (c) training time from 1.77 h to 4.31 h, a 2.4× spread despite identical hyperparameters, scenario, and data; (d) makespan from 190.3 h to 264.2 h. All four values in every panel arise from the same training procedure and differ only by random seed.

Read as a seed-variance characterisation rather than a regime comparison, these figures indicate that identically-configured PPO training runs in this environment can differ by up to 3.7 percentage points in OTR, by a factor of 1.5 in average tardiness, and by a factor of 2.4 in wall-clock training time, from random seed alone. This is a modest but genuine finding: it establishes a noise floor against which any future, genuinely regime-diverse comparison would need to be judged, and it suggests that single-seed results elsewhere in this chapter should be read with this variance in mind. The training-time spread in particular should not be read as evidence about re-training cost under different disturbance conditions, since no disturbance condition in fact varied

Table 5.11: Seed-to-seed variance in training time and 5-episode evaluation metrics across four identically-configured PPO replicas (Section **??**); row labels identify the training run, not a distinct regime. Makespan values reflect per-episode simulation horizons under a single-scenario configuration and are not directly comparable to the multi-month, multi-scenario training aggregates in Table **??**, nor to the 30-episode Catalogue A results in Table **??**.

Replica	OTR (%)	Tard (h)	Makespan (h)	Train time (h)	Avg reward
"smooth"	36.54	445.6	190.3	4.31	+137.2
"normal"	35.88	573.2	203.5	1.90	−93.4
"mixed"	32.84	686.3	264.2	4.29	−229.2
"complex"	34.35	521.5	218.1	1.77	+68.0

between these four runs.

5.5.3 Rule-Selection Behaviour Across Replicas and Evaluation Conditions

As established in Section **??** above, the four checkpoints analysed here are independently-seeded replicas of an identical training procedure, not regime-specialised policies. Figure **??** and Table **??** instead report how each replica's rule choice responds to the one axis that does vary genuinely: the evaluation-time data condition (`smooth`/`normal`/`complex`, Section **??**).

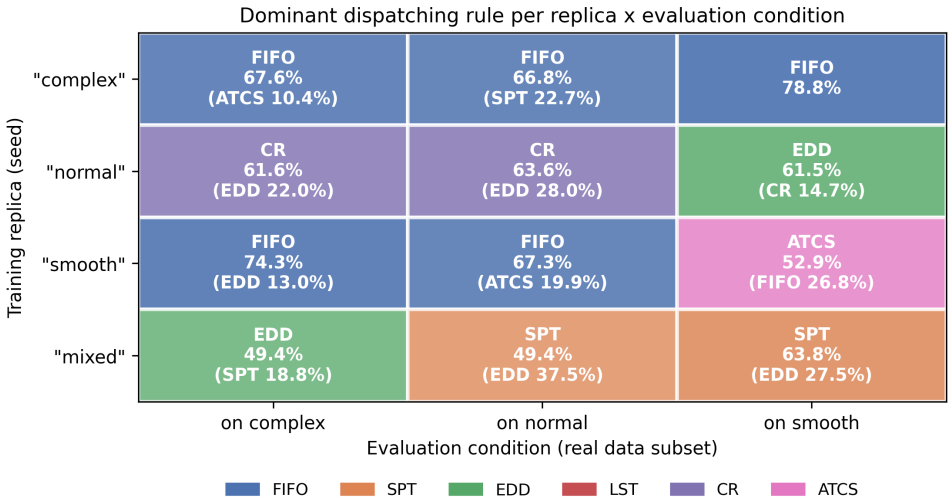

Figure 5.10: Dominant dispatching rule by training replica and evaluation condition (30 evaluation episodes per cell, computed from per-step logs). Colour indicates the dominant rule; opacity scales with its usage share. Second-most-used rule shown in parentheses where its share exceeds 10 percentage points.

Two patterns emerge from Table **??**. First, under the two more congested evaluation conditions (`on complex`, mean queue time ≈218 h per lot; `on normal`), three of the four replicas converge on the same rule family despite having been trained independently: two settle on FIFO (67–74% across their `on complex` and `on normal` cells) and one on CR (62–64%). Of the six rules (Section **??** of Chapter **??**), FIFO is computed from no predicted quantity at all, and CR's

Table 5.12: Dominant dispatching rule (percentage of dispatch steps, 30 evaluation episodes per cell) for each of four independently-seeded PPO replicas, evaluated on three real data conditions. All four replicas share identical training configuration (B0 reward, single "normal" scenario, month 2025-01); the row labels identify the training run (random seed), not a distinct training regime.

Replica	on complex	on normal	on smooth
"complex"	**FIFO** 67.6% (ATCS 10.4%)	**FIFO** 66.8% (SPT 22.7%)	**FIFO** 78.8%
"normal"	**CR** 61.6% (EDD 22.0%)	**CR** 63.6% (EDD 28.0%)	**EDD** 61.5% (CR 14.7%)
"smooth"	**FIFO** 74.3% (EDD 13.0%)	**FIFO** 67.3% (ATCS 19.9%)	**ATCS** 52.9% (FIFO 26.8%)
"mixed"	**EDD** 49.4% (SPT 18.8%)	**SPT** 49.4% (EDD 37.5%)	**SPT** 63.8% (EDD 27.5%)

numerator is the due-date-derived slack rather than a processing-time estimate, so this convergence is at least consistent with congestion penalising rules whose score depends most directly on the setup/processing-time prediction τ_{rem} (namely LST and, to a lesser extent, ATCS and SPT). Second, this partial agreement disappears under the `smooth` condition (mean queue time $\approx$0.7 h): each of the four replicas selects a different dominant rule (FIFO, EDD, ATCS, SPT).

A plausible, though unconfirmed, explanation is that heavy congestion pushes the observed state into a narrow region of extreme wait times and slack values, where the policy's response is comparatively constrained regardless of which seed produced it, whereas the `smooth` condition leaves the state distribution closer to the single "normal" scenario each replica was actually trained on, where each seed's own exploration history determines the outcome. Because only one training run exists per label, this pattern cannot be distinguished from an artefact of the small sample of seeds ($n = 4$), and no claim of statistical significance is made.

The evaluation-condition sensitivity documented above does not, on its own, support a claim of learned, context-dependent rule specialisation across genuinely different training regimes, since no disturbance-severity difference existed between the four training runs analysed here; establishing that would require retraining under genuinely distinct disturbance regimes, which is left to future work (Section **??**).

5.6 Chapter Summary

This chapter evaluated the PPO-trained meta-dispatcher across four experimental dimensions, each addressing one research question.

Reward design (RQ2). Of the four reward variants compared, B4($w = 0.3$), a multi-term base reward with moderate regret shaping, was selected for its lowest average tardiness and smoothest convergence. The ablation shows that excessive setup penalisation (B3) and excessive regret weighting (B4 at $w = 0.5$) both degrade tardiness, confirming that counterfactual guidance helps only when it does not overwhelm the base cost signal. The accompanying state-representation ablation is preliminary and its conclusions are deferred to future work (Section **??**).

Comparative performance (RQ1). Across five Catalogue A scenarios (150 episodes per policy, single evaluation run), the policy attains the lowest average tardiness of any policy (214.2 h, -25.7% vs. EDD and -30.1% vs. SPT) at competitive OTR (46.0% vs. a 46.4% best baseline). The difference is consistent in the mean but not statistically significant per-episode (paired Wilcoxon $p > 0.17$); the finding is thus an adaptive advantage on the reward's target objective, not Pareto dominance.

Disturbance robustness and theoretical alignment (RQ3). Under Catalogue B, the policy leads only in *high_mix_pressure* (47.3% OTR) and trails the best static baseline by 2–3.5 pp in maintenance- and hot-lot-dominated scenarios, a reward trade-off favouring near-zero hot-lot tardiness over aggregate OTR. The four regime-variant checkpoints were in fact trained under an identical configuration rather than four distinct disturbance regimes (Section **??**); the analysis is accordingly framed around evaluation-condition sensitivity across independently-seeded replicas rather than learned regime specialisation. Finally, across the four Catalogue A scenarios, the dominant rule reduces in practice to a choice between FIFO and ATCS: ATCS dominates the two scenarios where setup cost is emphasised, consistent with its explicit setup-time penalty term, and FIFO dominates the other two, a pattern that is directionally consistent with scheduling theory on the setup-cost dimension without being a six-way match.

Taken together, these results validate the core argument of this thesis: a reinforcement learning meta-dispatcher can learn adaptive rule-switching strategies that align with scheduling theory and achieve the lowest tardiness of any evaluated policy across controlled scenario types, within a single reward function and without scenario-level supervision. Chapter **??** draws out what these findings mean in practice and where the approach still falls short.

Chapter 6

Conclusions and Future Work

This chapter synthesises the principal contributions of this dissertation, building on the empirical evaluation reported in Chapter **??**, and outlines research directions that build on its findings. Section **??** summarises the methodological and empirical advances achieved across the problem formulation, environment design, reward engineering, and empirical evaluation dimensions. Section **??** identifies current limitations and proposes concrete extensions.

6.1 Main Contributions

This dissertation demonstrates that a reinforcement learning meta-dispatcher can learn adaptive rule-switching strategies that achieve the lowest tardiness of any evaluated dispatching policy in a real-world semiconductor fab setting (214.2 h averaged across five controlled scenario types, a 25.7% reduction over the next-best static heuristic), while adapting its rule-selection behaviour to heterogeneous operating conditions in a way that is directionally, though not precisely, consistent with classical scheduling theory.

Data-driven simulation environment. A custom discrete-event simulation (DES) environment, `FlowSimulator`, was constructed from five months of operational MES records from a real semiconductor facility (January–May 2025), covering 622 machines, 281 product flows, and 549–1,857 active lots per simulation episode. The environment faithfully models hierarchical machine sub-resources (load ports), sequence-dependent setup times derived from a recipe-family matrix, MES exception dynamics (holds, reworks, aborts, off-flows at empirically calibrated rates), and equipment dedication constraints for critical lithographic layers. A nine-stage data processing pipeline, orchestrated by a single function (`run_all_in_one_pipeline()`), transforms raw `MaterialLog`, `ResourceLog`, `MaintenanceLog`, and `RecipeParameters` records into the simulation-ready artefacts of Table **??**. The environment is exposed to the RL agent through a Gymnasium-compatible wrapper stack that supports multi-scenario training via round-robin sampling across ten synthetic operating conditions drawn from two scenario catalogues (Catalogue A: rule-regime scenarios; Catalogue B: fab-disturbance scenarios).

MDP formulation with regret-shaped reward. The dispatching problem is formalised as a finite-horizon MDP with a 35-dimensional aggregated state space, a six-action rule-selection space, and a multi-term dense reward function. The state representation is organised into ten semantic categories spanning lot counts, slack times, urgency indicators, WIP structure, fab disturbance metrics, and six heuristic cost proxy features that encode the estimated immediate dispatch cost of each candidate rule, providing built-in heuristic supervision without constraining the policy. The primary reward variant (B4, $w = 0.3$) augments a multi-term base reward with a bounded regret-shaping signal derived from the gap between the agent's chosen rule and the locally best rule, providing counterfactual gradient guidance without destabilising learning under multi-scenario training.

Reward design analysis. A systematic ablation of four reward variants (B2–B4) reveals that reward structure substantially affects both convergence dynamics and asymptotic performance. Increasing the setup penalty beyond a threshold reduces OTR without improving tardiness, reflecting over-penalisation of operationally necessary changeovers. Moderate regret shaping ($w = 0.3$) simultaneously recovers OTR and achieves the lowest average tardiness (32,161 h), while aggressive regret weighting ($w = 0.5$) narrows the policy's optimisation landscape and degrades tardiness by 6.9% relative to B4($w = 0.3$). These findings provide actionable guidance for reward engineering in manufacturing RL: a dense multi-term reward with bounded counterfactual shaping outperforms simpler alternatives, but the shaping weight requires careful calibration to avoid overwhelming the primary cost signal.

Empirical validation on industrial data. Against five classical dispatching baselines on 30 independent held-out test episodes per Catalogue A scenario (150 episodes per policy across five Catalogue A scenarios drawn from a single evaluation run, May 2025, 549 lots per episode), the trained B4($w = 0.3$) policy achieves the *lowest average tardiness* of any evaluated policy: 214.2 h, a 25.7% reduction relative to the next-best baseline (EDD at 288.1 h) and a 30.1% reduction relative to SPT (306.5 h). In terms of on-time rate, the RL policy achieves 46.0%, competitive with the best baselines (EDD 46.4%, SPT and LST 46.3%); the sub-half-point OTR gap is operationally small. The RL policy does not achieve Pareto dominance over all metrics simultaneously — CR produces the shortest makespan (1,955.1 h, versus PPO's 1,968.3 h) and EDD leads marginally on OTR — but it occupies the most favourable position in the OTR–tardiness trade-off space, reducing FIFO's average tardiness by 44.0% (382.7 h $\rightarrow$ 214.2 h) at matched OTR. This per-episode tardiness difference, while consistent in the mean, is not statistically significant (paired Wilcoxon $p > 0.17$ against every baseline), a consequence of the heavily right-skewed, scenario-dependent tardiness distribution. A calibrated ATCS baseline, excluded from the current evaluation due to parameter calibration requirements, would represent the most stringent single-rule comparison and is identified as a priority for future experimental evaluation.

Interpretable, theory-aligned rule-switching behaviour. Scenario-specific rule-selection analysis across the four Catalogue A scenarios shows the trained policy relying almost entirely on two rules, ATCS and FIFO, which together account for 83.6–91.3% of decisions in every scenario. ATCS dominates the two scenarios where setup cost is emphasised, consistent with its explicit setup-time penalty term, and FIFO dominates the other two. This is a coarser pattern than a distinct rule for every scenario, but it remains directionally aligned with scheduling theory on the setup-cost dimension (Section **??** of Chapter **??**).

The four regime-variant checkpoints (Section **??** of Chapter **??**) were in fact trained under an identical configuration. All four used the same single scenario, month, and B0 reward, rather than four qualitatively distinct disturbance regimes. Across the four independently-seeded replicas of this training procedure, three converge on the same dominant rule (FIFO or CR) when evaluated on congested real data, but diverge to a different dominant rule each under uncongested data, consistent with congestion narrowing the effective range of states the policy must respond to. With only four independent seeds, this pattern cannot be reported as statistically significant, and a genuinely regime-diverse retraining is identified as a priority for future work (Section **??**) to test the original specialisation hypothesis properly. Under Catalogue B disturbance scenarios, the agent achieves the best OTR only in the *high_mix_pressure* scenario (47.3%); in the five remaining disturbance scenarios it falls below the best static baseline by 2.1–3.5 pp, reflecting a reward-function trade-off between hot-lot urgency and aggregate OTR rather than a generalisation failure.

6.2 Limitations and Future Work

The results presented here are promising, but several modelling simplifications and experimental constraints limit the direct transferability of the learned policies to live MES deployments. The following limitations simultaneously identify the most productive research directions for future work.

State ablation scope. The state-representation ablation reported in Section **??** is preliminary: it was run under an earlier configuration in which the fab-disturbance features (indices 22–27) were not yet active, so the 35- and 29-dimensional variants carried no additional disturbance signal over a 22-dimensional representation. The nominal advantage of the minimal 7-dimensional configuration is therefore not conclusive evidence that richer representations are unhelpful. The main training and evaluation runs (Sections **??**–**??**) use the configuration with all features fully active and are unaffected. A complete state ablation with all features active and a 30-episode evaluation budget is deferred to future work.

Queue-time constraint activation (highest-priority extension). The most immediate and highest-impact extension is the activation of the queue-time constraint mechanism already implemented in `FlowSimulator`. This mechanism tracks elapsed time between consecutive process

steps and triggers hold events when inter-step windows are exceeded; it represents the most common real-world source of MES exceptions (holds and reworks) in semiconductor manufacturing. The mechanism is currently inactive because explicit queue-time window specifications were not available in the source dataset (see Assumption 6, Chapter **??**). Obtaining these specifications from process engineering documentation at the facility would require no further code changes and would immediately yield a higher-fidelity training environment, allowing the agent to learn hold-avoidance strategies that are absent from the current policy. This step is planned as the immediate next experimental phase.

Simulation fidelity. The current simulation assumes instantaneous inter-machine lot transport, fully autonomous processing (no operator constraints), and single-lot machine capacity (no batching). Real production at the facility involves AMR-mediated carrier transport (introducing variable delay), technician-scheduled manual interventions, and batch-capable tools (spin coaters, clean chambers) that process multiple lots concurrently. Enriching the simulation with transport time distributions, operator shift constraints, and explicit batch-scheduling sub-decisions would increase fidelity and alter the dispatching dynamics substantially.

Graph-based state representation. The 35-dimensional aggregated state captures fab-level statistics but loses per-lot and per-machine detail. This introduces mild partial observability that is acceptable under the current operating regimes but may limit performance in fabs with more heterogeneous product mixes or complex machine-eligibility topologies. Graph neural network (GNN) architectures that operate on a heterogeneous graph of lots and machines connected by eligibility edges offer a principled path to richer, permutation-equivariant representations [**?**, **?**]. The GNN approach would also remove the need to manually engineer normalization constants, as the network can learn appropriate feature scales from data.

Lookahead-augmented decision-making. The trained policy is strictly reactive: it conditions only on the current observation o_t and commits to a rule without simulating the consequences of that choice, an approximation whose theoretical cost is discussed in Section **??** of Chapter **??**. The heuristic cost proxies already embedded in the state vector (indices 29–34, Section **??**) are a shallow instance of this same idea: they estimate each rule's immediate one-step dispatch cost without actually dispatching, but the estimate stops at a single step and never accounts for how today's choice reshapes the queue several decisions later. Because `FlowSimulator` is already fast enough for the training budgets used in this thesis (Section **??**), it could double as a lightweight world model at inference time: before committing to a rule at a given decision epoch, the agent could roll out a short simulated horizon under each candidate action and select the one with the best simulated outcome over that horizon, in the style of Monte Carlo Tree Search or model-predictive control layered on top of the learned policy, rather than relying on the policy network's single forward pass. Such lookahead would trade additional per-decision computation for a partial remedy to the reactive policy's blindness to multi-step consequences, and would help isolate how

much of any remaining performance gap is attributable to partial observability versus the purely reactive nature of the current architecture.

Multi-level and multi-site dispatching. The current formulation is restricted to lot-level dispatching within a single frontend fab. Real semiconductor manufacturing involves hierarchical scheduling from lot release planning (weeks-level) down to real-time machine assignment (minutes-level), as well as inter-site logistics to backend assembly and test facilities. A multi-level hierarchical dispatcher, in which a strategic policy sets lot release rates and a tactical policy manages real-time decisions, would increase practical applicability. A multi-agent extension in which each work area (photolithography, etch, metrology) has an independent rule-selector coordinated through shared observation channels would enable finer-grained specialisation while preserving interpretability.

Transfer learning and generalisation. The policy is trained and evaluated on a single semiconductor facility. Transferring a learned policy to a different fab with a different machine park, product mix, or operating philosophy remains an open challenge. Meta-learning approaches (MAML, Reptile) that produce quickly adaptable policies from a small number of fine-tuning episodes, or domain-randomisation strategies that deliberately broaden the training distribution beyond the ten-scenario catalogue, represent natural extensions for cross-fab policy transfer.

Online learning and MES integration. The current deployment model assumes a static policy evaluated on held-out months. Real MES production is non-stationary: machine qualifications change, new product introductions alter the flow universe, and seasonal demand variations shift the distribution the policy was trained on. Training a single 200,000-timestep policy on the hardware used in this thesis takes on the order of one to four hours (1.77–4.31 h across four identically-configured runs, Section **??** of Chapter **??**), making periodic supervised retraining practical; that spread reflects run-to-run seed variance rather than regime difficulty, so it should be read as an order-of-magnitude estimate rather than a precise cost figure. More ambitiously, continual learning approaches that update the policy incrementally from incoming production data, while enforcing safety constraints to prevent performance regression during adaptation, would enable genuine real-time policy evolution. Integration with the Critical Manufacturing MES API would provide the necessary data pipeline for such online adaptation.

Safe and certifiable RL. The deployment of a learned dispatching policy in a live fab requires safety guarantees that current policy-gradient methods do not provide. Constraint-augmented RL frameworks (e.g., constrained MDP formulations, Lagrangian methods) that enforce hard production constraints such as minimum throughput, maximum WIP bounds and guaranteed on-time rates for priority lots, would bring learned policies closer to industrial certification requirements. Validating policy behaviour against formally specified temporal logic properties using model-checking tools, or using interpretable policy representations (decision trees, rule lists) distilled

from the trained neural network, are complementary paths toward auditable, deployable dispatching intelligence.

Concluding remark. The central finding of this dissertation is that an RL meta-dispatcher trained on real MES data achieves the lowest tardiness of any evaluated policy on Catalogue A conditions (214.2 h, a 25.7% reduction over EDD and 30.1% over SPT), while maintaining competitive on-time delivery rates, and its rule choice is directionally, if not precisely, consistent with classical scheduling theory: it relies on ATCS, the only rule with an explicit setup-cost term, specifically in the two scenarios where setup pressure is emphasised (Section **??** above). The combination of empirical tardiness advantage and this setup-aware rule preference goes beyond pure performance comparisons: the agent's rule choice is not arbitrary, even though it reduces to a two-way split rather than a distinct rule per scenario. The most immediate priority for strengthening these findings is the activation of queue-time constraints, which would more rigorously characterise the agent's capabilities relative to its operational context; a calibrated ATCS baseline comparison, identified in Chapter **??** as a priority for future experimental evaluation, would provide a smaller-scope complement to that effort. The extensions outlined above represent a concrete roadmap for evolving the proof-of-concept presented here into a production-grade, continuously improving dispatching intelligence system for advanced semiconductor manufacturing.

Appendix A

Data Processing Pipeline

Before the simulation environment can be instantiated, the raw MES and equipment records must be transformed into a consistent set of simulation-ready artefacts. Unlike studies that rely on standardised academic testbeds such as SMT2020 [**?**] or MiniFab [**?**], this work uses operational production data from an active semiconductor research facility, providing higher fidelity at the cost of reduced reproducibility. The data covers five months from January through May 2025 and was extracted from the Critical Manufacturing MES of the target wafer-preparation facility. The transformation is carried out by a data processing pipeline, orchestrated by a single function, `run_all_in_one_pipeline()` (`all_in_one_pipeline.py`), that ingests four primary data sources and writes a set of artefacts to disk under their real filenames, as illustrated in Figure **??** of Chapter **??** (not reproduced here to avoid duplicating the same figure across chapter and appendix). All numerical claims in this section are derived from the pipeline's own outputs and the supporting CSVs produced by the ingestion and cleaning stage described below (Sections **??**–**??**), except where a section is explicitly marked as exploratory analysis (see the note at the start of Section **??**).

Where each stage is detailed further. The nine pipeline stages are numbered, described, and illustrated in full in Section **??** of Chapter **??**; that walkthrough is not repeated here. This appendix instead supplies, stage by stage, the raw-table schemas, row-level cleaning filters, and validation statistics the chapter omits for readability: Stage 1 (lot reconstruction) in Sections **??**–**??** and **??**; Stages 2–3 (machine hierarchy and batch detection) in Section **??**; Stage 5 (maintenance parsing) in Section **??**; Stage 6 (recipe and setup-time extraction) in Section **??**; Stage 8 (eligibility mask) in Section **??**; and Stage 9 (monthly split) in Section **??**. Table **??** in Section **??** additionally lists every artefact by its real on-disk filename.

Two further analyses referenced elsewhere in this appendix, the flow universe and phase-frequency characterisations of Sections **??**–**??**, and the reentrance characterisation of Section **??**, are *not* produced by any of the nine pipeline stages; they were established through standalone exploratory analysis and are not part of the reproducible pipeline, as noted where each is introduced.

Two-tier dataset structure. This appendix describes two distinct datasets that serve different roles and must not be conflated. **Dataset A** (the MES event dataset) is the full `MaterialLog` export containing 7,653,932 cleaned events for 173,227 lots across 1,114 product flows; it is used exclusively for exploratory analysis (Sections **??**–**??**) and is never fed directly into the simulator. **Dataset B** (the simulation routing dataset, `steps_lots_clean.parquet`) is derived from Dataset A by the lot reconstruction pipeline and contains 3,883 lot objects with 11,234 processing steps across 281 product flows; it is the sole input to `FlowSimulator` and the source of all lot counts reported in Chapter **??**.

Data anonymization. The client that provided the underlying MES data is a semiconductor front-end manufacturer and is not named anywhere in this thesis. To prevent re-identification of specific equipment or product flows by anyone familiar with the facility's operations, every `FlowName` and `ResourceName` (machine) value reported in this appendix has been replaced with an anonymized label (`Flow-01`, `Flow-02`, ... for flows; `Machine-A`, `Machine-B`, ... for machines; `Area-1`, `Area-2`, ... for named work areas) drawn from a single mapping applied consistently across every table and paragraph below. `Aux-Flow-1` and `Aux-Flow-2` (Section **??**) are a separate label sequence reserved for two client-specific container-logistics flow names that are filtered out during cleaning and therefore never enter the ranked `Flow-01`–`Flow-10` list of Table **??**; the two sequences should not be read as sharing a rank order. Generic process-step or lifecycle labels that describe standard industry operations rather than client-specific assets (e.g. `Deposition`, `TrackIn`, `InspectDefect`) are not anonymized, since they carry no facility-identifying information on their own. Internal artefact and column names produced by this thesis's own pipeline code (e.g. `MaterialLog.csv`, `steps_lots_clean.parquet`) are likewise left as-is, as they describe the pipeline's software, not the client's equipment.

A.1 Raw Data Sources

Three primary tables and one independently sourced auxiliary table drive the project. The primary tables are `MaterialLog.csv` (lot-level events), `ResourceLog.csv` (machine-level events), and `RecipeParameters.csv` (per-lot recipe parameters). `MaintenanceLog.csv` is the fourth, independently sourced table. Two further artefacts are worth distinguishing from these: `CandidateResources.csv` (Section **??**) is not an independent MES export but a reformatted view of the machine-assignment history already present in `MaterialLog`, and `ResourceSubResources.csv`, occasionally referenced as a source for the parent–subresource hierarchy, is not consumed anywhere in the pipeline that produces the reported results; the machine hierarchy is instead derived directly from the `ParentResourcesCount` column of `ResourceLog` (Section **??**).

MaterialLog. A total of 4.6 GB and 39 columns comprised the original `MaterialLog`. Each row is one lot-level event recorded against a `(MaterialName, ModifiedOn,`

`OperationName, ResourceName)` tuple. The columns relevant for this work are listed in Table **??**.

Column	Description
`MaterialName`	lot identifier; the row is at lot level when `ParentMaterialName IS NULL`, and at wafer level otherwise.
`ParentMaterialName`	lot identifier of the parent when the row corresponds to a wafer or to a child lot produced by a `Split`; `NULL` for true lot rows.
`OperationName`	MES operation: `Dispatch`, `TrackIn`, `TrackOut`, `Hold`, `Release`, `AbortProcess`, `Split`, `RecordLoss`, `Terminate`, `Ship`, etc.
`SystemState`	lifecycle state at the moment of the event: `Queued`, `Dispatched`, `InProgress`, `Processed`, `InTransit`.
`StepName`	current step within the lot's flow (`Deposition`, `LithoExposure`, `InspectDefect`, ...).
`FlowName`	current flow of the lot (e.g. `Flow-A`, `Flow-B`; see the anonymization note on p. **??** for the convention used for real flow names reported elsewhere in this appendix).
`LogicalFlowPath`	full path within the flow, used to distinguish branches and reentries.
`ResourceName`	main machine where the event happened.
`ProductName`, `ProductGroupName`	product and product family of the lot.
`Recipe`	recipe identifier (mostly `NULL` except for control-run-bearing steps).
`DateEnteredStep`	timestamp at which the lot entered its current step (canonical queue-start).
`ModifiedOn`	event timestamp.
`Priority`, `IsHot`, `DueDate`	lot-level priority signals used by the dispatcher.
`PrimaryQuantity`, `PrimaryUnits`	wafers in the lot (`Wafer`) or lot count (`Lot`).

Table A.1: Relevant columns of `MaterialLog.csv`.

ResourceLog. The `ResourceLog` (1.1 GB, 36 columns) records machine-level events. Each row is one event against a resource identified by `(ResourceName, ParentResourcesCount, ProcessingType)`. The flat hierarchy is rooted at main machines (`ParentResourcesCount = 0`) that own subresources of four disjoint `ProcessingTypes`: `Process` (the chamber), `LoadPort`, `Component`, and `Storage`. The columns we use are listed in Table **??**.

RecipeParameters. The `RecipeParameters` table (166 MB, 15 columns) contains one row per `(MaterialName, StepName, ParameterName, Value)` tuple and exposes the actual parameters that were applied during processing of every lot. The columns we use are `MaterialName`, `FlowPath`, `StepName`, `ProductName`, `ResourceName`, `ParameterName`, `ParameterType`, `ParameterScope`, `DataUnit`, `DataScale`, `ParameterDataType`, and `Value`. The `ParameterType` discriminates between recipe attributes (`RecipeAttr`, `RecipeParameter`), litho-specific attributes (`LithoReticleExposure`, `LithoCD`, `LithoResistProcess`,

Column	Description
ResourceName, ParentResourcesCount	resource identifier and depth in the hierarchy.
ProcessingType	one of Process, LoadPort, Component, Storage.
StateModelState	runtime state, e.g. UP/RUN, PARTLY_UP/RUN, IN_MAINT, IN_REPAIR, IDLE_ERROR.
MainStateModelStateReason	reason for the state, e.g. Production, ScheduledSpc, StartRepair, GoDown.
RecipeName	recipe currently loaded (drives the sequence-dependent setup matrix).
MaterialsInProcessCount, MaterialsInQueueCount	live KPI counters per resource.
Saturation, Efficiency	utilisation indicators.
OperationName	TrackIn, TrackOut, LogEvent, HoldForMaintenance, ReleaseFromMaintenance, ...

Table A.2: Relevant columns of ResourceLog.csv.

LithoResistLayer), and container-composition entries (ComposeBySlot_Stock, ComposeBySlot_UnknownWaferID). This table is the source of the per-step recipe family used to build the sequence-dependent setup matrix described in Section **??**.

A.2 Data Cleaning

The cleaning pipeline is a sequence of filters that converts the raw MaterialLog into the lot-level event table that the simulator consumes. Each filter is justified below and the row count after each step is reported in Table **??**.

1. **Lot vs. wafer.** The MaterialLog mixes lot-level rows (ParentMaterialName IS NULL) with wafer-level rows produced when individual wafers are tracked through sort or split. Since the simulator operates at lot granularity, only lot-level rows are kept.

2. **Container-logistics flows.** Several FlowName values starting with a square bracket ([Sort], [Split], [Merge], [EmptyRework], [Aux-Flow-1], [Aux-Flow-2], [OutOfFabStore], [WaferUnknown]; the two Aux-Flow entries are anonymized client-specific flow names, distinct from the Flow-01–Flow-10 ranking sequence, see the note on p. **??**) encode container-level events rather than production-process steps; they are removed.

3. **No-tool placeholder.** Rows with ResourceName == '[NoTool]' correspond to MES-side bookkeeping events that have no associated processing machine; they are removed.

4. **Multi-area duplicates.** A single Terminate or Ship event is sometimes replicated by area (e.g. Area-1, Area-2, External, WaferStorage; area names anonymized per

the note on p. **??**, `Area-1` being the same work area as `[Area-1]` in Table **??**). The duplicates are collapsed to a single row by deduplicating on `(MaterialId, ModifiedOn, OperationName, TransactionHistoryId)`.

5. **Sorting.** The result is sorted by `(MaterialName, DateEnteredStep, ModifiedOn)` so that the events of each lot are in chronological order and the per-step queue-start, processing-start, and processing-end can be derived by walking the sequence.

Step	Rows after step	% of original
Original `MaterialLog` (raw)	11,384,882	100.0
After lot vs. wafer filter	$\sim$ 11 M	$\sim$96.6
After bracket-flow removal	$\sim$ 9 M	$\sim$79.1
After NoTool removal	$\sim$ 9 M	$\sim$79.1
After multi-area collapse + sort	7,653,932	67.2

Table A.3: Rows kept after each cleaning filter.

The cleaned table contains 7,653,932 events spread over 173,227 distinct lots, 1,114 distinct `FlowNames`, and 90 distinct `StepNames`. 184 distinct `ResourceName` values appear in these events; cross-referencing that set against the full `ResourceLog` hierarchy (Section **??**) shows only 58 of them are true level-0 (`ParentResourcesCount = 0`) main processing machines, a smaller number than the 622 main machines present in `ResourceLog` overall, since the cleaned MaterialLog subset only touches the machines actually used by the flows retained after cleaning. The date range is `2025-01-01 17:15:36` to `2025-05-31 16:15:56`.

A.3 Operation Taxonomy

The cleaned table contains 230 distinct `OperationName` values; the top 25 are listed in Table **??**. For the dispatching problem we group them into five lifecycle categories, defined below.

1. **Queueing.** Rows with `SystemState == 'Queued'` mark the lot as waiting for a machine. The lot's `DateEnteredStep` field is the canonical queue-start and is preserved in every row of a given step.

2. **Dispatch.** The `Dispatch` operation (490,606 events) is the moment the dispatching system commits the lot to a specific qualified machine. It is the decision point that the policies of this thesis aim to optimise. `Dispatch` is slightly more frequent than `TrackIn` (488,541); we therefore use `Dispatch` as the primary anchor and fall back on `TrackIn` when a step has no `Dispatch` record. The `SystemState` of these rows is `Dispatched` or `InProgress`.

3. **Processing.** The first `SystemState == 'InProgress'` row per `(MaterialName, StepName)` marks the start of processing; the matching `TrackOut` (or the first `SystemState == 'Processed'`) marks its end.

OperationName	Count	OperationName	Count
SetDispatchableFlag	3,249,222	SetParentMaterial	173,200
Dispatch	**490,606**	DetachFrom	173,200
TrackIn	**488,541**	**Release**	**101,741**
Save	425,913	**Hold**	**99,295**
SaveAttributes	384,923	TransferSubMaterials	73,293
ChangeFlowAndStep	306,827	SplitFrom	46,018
SpecialChangeProduct	290,728	**Split**	**45,982**
RemoveSubMaterials	284,178	Unterminate	39,193
AddSubMaterials	272,861	SplitByProductFrom	36,556
Terminate	**272,549**	SplitByProduct	36,520
ApplySamplingPattern	184,546	**AbortProcess**	**30,367**
		PerformImmediateChecklist	26,470

Table A.4: Top 25 `OperationName` values in the cleaned `MaterialLog`. Operations relevant to dispatching are bold.

4. **Disturbance.** The dispatcher must absorb 30,367 `AbortProcess` events, 99,295 `Hold` events, 101,741 `Release` events, and 45,982 `Split` events.

5. **Termination.** A lot ends with either `Terminate` (272,549 events) or `Ship` with `SystemState == 'InTransit'`. Both are mapped to the simulator's `is_terminated` flag.

The `SystemState` distribution over the cleaned table (5,363,144 `Queued`; 1,166,078 `Dispatched`; 1,114,129 `InProgress`; 10,551 `Processed`; 30 `InTransit`) is dominated by queueing rows, which is expected because a lot waits in queue many more times than it is processed.

A.4 Flow Universe

Note on reproducibility. This section and Sections **??–??** report findings from standalone exploratory analysis conducted early in the project. Unlike the nine pipeline stages of Section **??**, this analysis is not re-executed by `run_all_in_one_pipeline()`, and its outputs are not written to any artefact the simulator reads at runtime; the figures below should be read as the empirical basis for design decisions made once, not as artefacts independently reproducible by re-running the pipeline.

After cleaning, 1,114 distinct `FlowNames` remain. Most of them are short logistical flows that do not constitute a dispatching problem: they have only one or two steps. To select flows that are scheduling-relevant, we apply two criteria:

1. **Substantial flows.** `FlowNames` with a median of at least four `StepNames` per lot are retained. This yields 174 flows.

2. **Healthy flows.** A flow is *healthy* if it is active (i.e. has at least 50 lots) in at least four of the five months and its monthly lot count has a coefficient of variation below 0.5. This

second filter is applied on top of the substantial criterion to identify flows whose sample size is large enough for statistical analysis. The resulting set (29 healthy flows) is the basis of the exploratory analysis in this appendix. Note that the simulator uses a broader set of 281 product flows from Dataset B (Section **??**).

The top ten flows by lot count are listed in Table **??**. The full Top-10 cluster accounts for 58.1% of all lots; the remaining 41.9% are distributed across a long tail of variants, which motivates the policy-generalisation problem studied in Chapter **??**.

FlowName	Lots	Median steps	Max steps	Share (%)
Flow-01	37,064	1	2	12.01
Flow-02	31,198	1	2	10.11
Flow-03	28,086	1	2	9.10
Flow-04	21,359	1	3	6.92
Flow-05	16,924	2	2	5.48
Flow-06	14,499	1	1	4.70
Flow-07	13,772	2	3	4.46
Flow-08	10,252	2	3	3.32
Flow-09	6,804	2	5	2.20
Flow-10	5,509	5	5	1.78

Table A.5: Top-10 `FlowName`s by lot count (flow names anonymized as `Flow-01`–`Flow-10` in rank order to avoid disclosing client-specific product/flow identifiers). Substantial production flows have a median of at least four steps; the only substantial flow in the top ten is `Flow-10`.

The dominance of one- and two-step flows in Table **??** reflects the high volume of container logistics in this work area: `Flow-01`, `Flow-03` and `Flow-04` are wrap-up flows that mark the entry and exit of pristine wafers into and out of the line. The dispatching problem is concentrated in the substantial flows below.

A.5 Phase Frequency per Flow

To characterise the production content of a flow without listing every `StepName`, we group steps into ten equipment *phases* according to a regex over the step name. The phases are: deposition, lithography, etch, CMP, implantation, clean, metrology, sort, container, and end-of-flow. The fraction of each flow's events that falls under each phase was recorded during this exploratory analysis pass. The aggregated breakdown over the substantial flows is reported in Table **??**.

Two observations from Table **??** drive the methodology choices in Chapter **??**. First, metrology accounts for over 20% of all events even after removing wafer-level rows; this confirms that the metrology load is the single largest source of capacity contention in this work area. Second, deposition and lithography together account for less than 26% of events, versus $\sim 34\%$ for sort and container handling combined; the sorters and pipeline tools are therefore as central to dispatching as the unit-process tools.

Phase	Events	Share (%)
Deposition	∼ 1,120,000	14.6
Lithography	∼ 840,000	11.0
Etch	∼ 690,000	9.0
CMP	∼ 170,000	2.2
Clean	∼ 310,000	4.1
Metrology	∼ 1,580,000	20.6
Sort	∼ 610,000	8.0
Container	∼ 1,960,000	25.6
End-of-flow	∼ 260,000	3.4
Other	∼ 114,000	1.5

Table A.6: Aggregated phase breakdown of the cleaned event table.

A.6 Inspect Steps as Bottleneck Evidence

The metrology phase is dominated by ten `StepName`s whose common feature is that they hold the lot pending an inspection result before allowing the next process step to proceed. The top metrology steps by event count are reported in Table **??**. The ratio between the number of events and the number of distinct lots that pass through each step is the empirical signature of a bottleneck: an `InspectDefect` step generates on average 42 events per lot before being released, against ~ 1 event per lot for a single-pass container-logistics step such as `[Aux-Flow-1]` (Section **??**).

StepName	Events	Lots	Events / Lot	Top resource
InspectDefect	16,888	406	41.6	Machine-A
DefectReview	15,808	348	45.4	Machine-B
InspectFIBSEM	5,592	87	64.3	Machine-C
InspectCriticalPoint	3,260	167	19.5	Machine-D
InspectXSEM_SamplePrep	1,612	43	37.5	[Area-1]
InspectOptical	972	15	64.8	Machine-E
InspectTEM	676	12	56.3	Machine-F
InspectXSEM	628	40	15.7	Machine-G
InspectTEM_DataProcessing	74	4	18.5	Machine-H
MeasBondDefects	32	2	16.0	Machine-I

Table A.7: Inspect/measure steps ranked by event count (machine identifiers anonymized per the note on p. **??**). The very high events-per-lot ratios reflect the per-step hold-and-release cycles characteristic of a quality bottleneck.

The dispatcher must therefore be aware that the metrology tools listed in Table **??** (MACHINE-B, MACHINE-A, MACHINE-C, MACHINE-D) are capacity-binding even though they are not unit-process tools. This empirical fact is encoded in the qualification mask: the 1,539 entries of the mask exclude lithography and etch machines from inspect-type steps and concentrate the load on the metrology family.

A.7 Reentrance vs. Batching

The classical fab-scheduling literature treats reentrant routing as the central modelling challenge [**?**]. Empirically, this is not the case in our work area. We define a *step root* as the `StepName` stripped of trailing `_2`, `_3`, ... suffixes (e.g. `Deposition HardMask_2` maps to `Deposition HardMask`). A lot is said to *re-enter* a step root if it visits the same step root more than once. The results are reported in Table **??**. Re-entrance happens for 71.2% of lots in the cleaned table, but only 4.7% of `(flow, step_root)` pairs exhibit a median number of visits greater than two. Most re-entries are therefore the expected `StepName`/`StepName_2` doubling that encodes a multi-layer deposition stack rather than the high-layer reentrant routing of a logic fab.

Metric	**Value**	**Share**
Lots with at least one re-entered step root	123,383/173,227	71.2%
`(flow, step_root)` pairs with median visits > 2	87/1,847	4.7%

Table A.8: Reentrance summary: reentrance is widespread but structurally shallow.

The take-away for dispatching is that the action set is dominated by routing decisions on metrology and sort/handling tools. Batch-formation plays only a secondary role in this work area, concentrated on a small set of pipeline tools; the simulator therefore models machines as single-slot resources and does not activate batch scheduling (Section **??**).

Implication for the literature positioning (Chapter ??). The RL scheduling literature, surveyed in Section **??**, treats deep reentrant routing as the primary structural challenge of semiconductor scheduling. The empirical data above qualifies this characterisation for the facility's work area: reentrancy is *widespread* (71.2% of lots re-enter at least one step root) but *structurally shallow* (only 4.7% of (flow, step_root) pairs have median visits > 2). The dominant operational challenge in this fab is therefore not deep multi-loop reentrancy but rather the combination of high-volume metrology bottlenecks and the concentration of batch-capable tools at sort steps. This empirical observation shapes the state representation design choices in Chapter **??**: fab-level disturbance features (machine load CoV, machines down, hot-lot fraction) capture the relevant bottleneck dynamics more directly than per-step reentry counters would.

A.8 Maintenance and Machine Availability

The `ResourceLog` table is the source of machine-availability information. We restrict the analysis to main machines (`ParentResourcesCount = 0`) and to events whose `StateModelState` reflects a transition or a maintenance operation: `IN_MAINT`, `IN_REPAIR`, `IDLE_ERROR`, or operation $\in$ {`LogEvent`, `HoldForMaintenance`, `ReleaseFromMaintenance`}. The resulting table contains 1,399,311 rows; the `LogEvent` subset alone contains 134,705 rows.

The most common `StateModelState` values across the maintenance window are `UP/RUN` (69%), `PARTLY_UP/RUN` (23%), `SPC_TEST` (3.3%), `IDLE_ERROR` (0.5%), `IN_MAINT` (0.1%), and `IN_REPAIR` (0.1%). The `MainStateModelStateReason` breakdown shows that the dominant reason is `Production` (91%), followed by the `Skip Automation` reason code (1.4%), `ScheduledSpc` (0.2%), and `StartRepair` (0.1%). Among the $\sim$5,000 paired downtime intervals reconstructed via `LogEvent` pairs, the median duration is 1.79 hours and the mean is 4.12 hours.

The top ten machines by maintenance event count are MACHINE-B, MACHINE-J, [MACHINE-K], MACHINE-L, MACHINE-M, MACHINE-N, MACHINE-O, MACHINE-P, MACHINE-Q, MACHINE-E (identifiers anonymized per the note on p. **??**; MACHINE-B and MACHINE-E are the same physical tools as in Table **??**). The list is dominated by metrology tools, which is consistent with the bottleneck observation in Table **??**: metrology resources are not only the busiest, they also accumulate the most state transitions.

A.9 Recipe and Setup Matrix

A recipe family is a categorical label that classifies the manufacturing recipe applied to a lot at a given process step. For each lot, the per-step recipe family is extracted from `RecipeParameters` by joining on the `ParameterType == 'RecipeAttr'` subset to obtain the `(StepName, ResourceName, RecipeName)` triple. In total, 8,232 distinct recipe families are identified across the dataset. Recipe family information is stored directly on each `Lot` object as a per-step dictionary (`lot.recipe_families`) and is transferred to the machine object at scheduling time via `machine.current_recipe_family`, enabling accurate setup time computation for subsequent operations.

The setup time matrix (`recipe_families.py`) is constructed by scanning consecutive operations on the same machine in the `MaterialLog`, identifying inter-operation gaps where the recipe family changes, and retaining only gaps in the range $(0,\ 4\text{h}]$ as plausible setup intervals. Aggregating these observations produces a matrix of size #recipes $\times$ #recipes whose (i, j)-th entry is the empirical setup duration when a machine switches from recipe family i to family j. The matrix has 1,526 non-trivial `(machine, from-family, to-family)` triplets, with a median setup time of 1.01 hours, a mean of 1.28 hours, and a maximum of 3.99 hours. Entries not covered by the matrix default to the empirical mode of 0.5 hours. A step-level fallback is also stored for use when the recipe identifier is unavailable. When the lot and the machine already share the same recipe family, no setup time is applied. Setup durations are retrieved via a three-strategy cascade: (i) direct lookup by `(machine, from-family, to-family)` triplet; (ii) family-level mapping when the exact triplet is absent; and (iii) a conservative default of 0.5 hours when no setup record exists.

A.10 Lot Reconstruction

The lot reconstruction module (`lot_builder.py`) is described in Section **??** of Chapter **??** (Stage 1) and is not repeated here; this appendix reports the cycle-time decomposition and validation statistics that support that description. Cycle time is decomposed into three components:

$$t_{\text{cycle}} = t_{\text{proc}} + t_{\text{queue}} + t_{\text{overhead}} \tag{A.1}$$

where $t_{\text{proc}} = t_{\text{Terminate}} - t_{\text{TrackIn}}$ is the machine processing time, $t_{\text{queue}} = t_{\text{TrackIn}} - t_{\text{prev\_end}}$ is the inter-step waiting time, and t_{overhead} encompasses holds, aborts, rework loops, and off-flows, the same four exception types enumerated later in this paragraph. This decomposition is critical because, in the production data, machine processing time accounts for 90.4% of the cycle and queue time for only 8.7%; yet it is queue time that RL can directly influence. The aggregate 8.7% figure masks substantial variance: the inter-step queue-time distribution has a median of 2.56 hours but a 99th percentile of 758 hours and a maximum of 17,228 hours, confirming that rare but prolonged queue buildups at metrology bottlenecks drive the bulk of the variability that the dispatching policy is designed to reduce. Exception events (holds, aborts, reworks, off-flows) are recorded as per-lot integer counters and boolean flags. Processing time outliers (defined as values exceeding three standard deviations from the per-(flow, step) median) are replaced with their median to prevent unrealistic simulation episodes. Out of 3,731 recorded processing durations, 2,157 (57.8%) require this correction. This high correction rate reflects a structural characteristic of the raw MES data: the majority of processing-time outliers are not measurement errors but legitimate holds, rework loops, or quality-inspection delays that temporarily inflate the `TrackIn–Terminate` interval for a subset of lots. Because the simulation models holds and reworks independently through stochastic exception injection (Appendix **??**), retaining these inflated processing times would double-count their effect; the 3σ-clipping therefore separates pure machine processing time from superimposed exception delays, and the corrected values represent the nominal machine cycle time that the dispatching model requires.

The pipeline produces 3,883 lot objects with 11,234 processing steps in total (steps without a valid machine assignment are excluded from the simulation routing table), covering 281 unique product flows and exhibiting rework rates of 0.19%, abort rates of 7.87%, and temporary off-flow rates of 1.95%.

A.11 Machine Hierarchy Reconstruction

The machine reconstruction module (`machine_builder.py`) builds a three-level equipment hierarchy from the `ResourceLog`. The 1,923 resources are partitioned into 622 level-0 machines and 1,301 level-1 subresources. Subresources are further classified by processing type:

- **Process Chambers** (7 units, 2 parent machines): Present in the Machine object model for BONDER-1 (4 chambers) and WETSTRIP-1 (3 chambers) (machine identifiers anonymized

per the note on p. **??**; generic tool-type labels are used here since these two machines are otherwise uniquely identifiable by chamber count alone). Chamber sub-resources are modelled in the data structures but are not activated in the current simulation; the dispatcher treats these machines as single-slot resources.

- **Components** (463 units): Non-productive subresources (pumps, RF generators, valves, sensors, and filters) whose failure cascades to their parent machine.

In addition, batch processing characteristics are inferred from co-temporal `TrackIn` events in the `MaterialLog` (`batch_detection.py`): 71 machines are identified as capable of processing multiple lots simultaneously, with typical batch sizes ranging from 5 (multi-recipe capable) to 11 (single-recipe) lots per batch.

Maintenance events are parsed from the `MaintenanceLog` by a separate stage (`parse_maintenance_log()`) and written to their own artefact, `maintenance_events.parquet`; they are not merged back into the `machines_with_hierarchy.pkl` pickle produced by this section, so a machine object loaded from that pickle alone carries no maintenance schedule. Where downstream components attach maintenance events to a machine object at runtime (via `machine.maintenance_events`, an ordered list of records with begin and end timestamps), they do so by joining the two artefacts, not by reading a single combined file. Regardless of which artefact carries them, a `classify_maintenance_events()` step partitions these events into two types. *Predictable* (preventive) maintenance accounts for 71.6% of events and corresponds to scheduled downtime interventions with known temporal boundaries. *Adhoc* (corrective) maintenance accounts for the remaining 28.4% and represents reactive repairs triggered by equipment failures or quality excursions, which are characterised by shorter and more variable durations. Of the 622 machines, 363 (58.4%) have at least one associated maintenance event across the five-month dataset, with a mean downtime duration of 34.5 hours.

A.12 Eligibility Mask

The eligibility mask (`candidate_machines.py`) maps each (flow, step) pair to its set of eligible machines by taking, per lot, the union of the initial and secondary machine fields recorded in the lot-level records reconstructed from `MaterialLog` (Section **??**). The intermediate `CandidateResources.csv` artefact this module writes is a reformatted view of that same union, keyed by (flow, step, operation, resource), not a second, independently sourced table: the eligibility mask is therefore derived entirely from MaterialLog machine-assignment history, and its 100% coverage check against historical assignments (Appendix **??**) is an internal consistency check rather than a cross-validation against an independent MES qualification source. The full set covers 2,854 (flow, step) pairs; after filtering to the 281 flows that form Dataset B (Section **??**), the active mask reduces to 1,539 entries, which is the figure reported in Table **??**.

Sparsity distribution. Table **??** reports the distribution of eligible-machine count per (flow, step) pair. A particularly important finding is that 72.0% of pairs have exactly one eligible machine, meaning that dispatching rule selection has no effect on machine assignment for the majority of steps; the primary scheduling lever is *sequencing* (which lot goes first), not *routing* (which machine is assigned).

Table A.9: Eligibility mask sparsity: distribution of eligible machines per (flow, step) pair.

Eligible machines	(flow, step) pairs	Share (%)
1	2,055	72.0
2	513	18.0
3–5	229	8.0
≥ 6	57	2.0
Total	**2,854**	**100.0**

Validation against historical assignments. The eligibility mask was cross-validated against the historical MaterialLog: 100% of observed machine assignments in the dataset fall within the computed eligibility sets, confirming that no historical dispatch violated the mask. This check was performed by `candidate_machines.py` during pipeline execution and serves as a continuous regression test when the pipeline is re-run on updated data.

A.13 Month Selection for RL Training

We score each of the five months on four dimensions: number of distinct lots (L_m), number of distinct flows (F_m), number of distinct machines (M_m), and number of `LogEvent`-derived maintenance events (E_m). For each dimension d, the rank is normalised by the maximum value across months:

$$\text{score}(m) = \frac{L_m}{\max_m L_m} + \frac{F_m}{\max_m F_m} + \frac{M_m}{\max_m M_m} + \frac{E_m}{\max_m E_m} \tag{A.2}$$

The top-scoring month is selected as the held-out test month; this criterion identifies the month that is simultaneously richest in lot volume, flow diversity, machine coverage, and maintenance complexity, the most representative and demanding evaluation setting. The result is reported in Table **??**.

Why bigger months make the problem harder. Most thesis-scale dispatching benchmarks (Mason instances of [**?**]; Mönch instances of [**?**]; SimRLFab of [**?**]) have between 5 and 15 machines and a single flow with 5 to 15 fixed steps. Adding more machines does not just enlarge the action set: it also enlarges the qualification mask (machine $\times$ step) and the setup-family graph, both of which grow super-linearly. With 173 qualified machines and 727 concurrent flows in the May 2025 month, the dispatcher must balance throughput on shared resources across substantially

Month	Lots	Flows	Machines	Maint. events	Score
2025-05	**51,804**	**727**	**173**	**286,788**	**3.711**
2025-01	66,026	598	166	255,432	3.608
2025-04	38,348	665	170	281,844	3.389
2025-03	33,783	608	172	309,447	3.342
2025-02	33,167	529	162	265,800	3.025

Table A.10: Monthly statistics and composite score. The May 2025 month combines the highest flow diversity, the highest machine count, and a high maintenance-event count, making it the most demanding month for a dispatching policy.

more competing lots than the standard benchmarks. This is what justifies the extra training budget reported in Chapter **??** and the use of a meta-selector rather than direct discrete dispatching.

A.14 Train, Validation, and Test Split

The temporal split operates on **Dataset B** (the simulation routing dataset, 3,883 lots, 281 flows), not on the full MES event dataset. The pipeline (`monthly_split.py`) partitions Dataset B into monthly slices, preserving work-in-progress lots whose processing spans month boundaries. The adopted split is **train: January 2025** (1,857 lots), **validation: April 2025** (604 lots), **test: May 2025** (549 lots). Note that RL training uses January 2025 only (not January–March); April is retained for hyperparameter validation and May is the held-out test month, accessed only during final evaluation. These lot counts differ substantially from the broader Dataset A figures (e.g. 51,804 lots in May 2025 in Table **??**), which reflect all MES events and are used only for the exploratory analysis in Sections **??**–**??**.

A.15 Final Dataset Summary

The final dataset that drives the simulator and the dispatching policies of this thesis is summarised in Table **??**.

A.16 Runtime Execution Flow

The sections above describe the nine pipeline stages as an offline, conceptual dependency graph, run once to produce the artefacts of Table **??**. At runtime, before every training or evaluation run, these artefacts are loaded (or, on a cache miss, rebuilt from the raw MES tables) by a single entry point, `build_simulator_components()`, illustrated in Figure **??**.

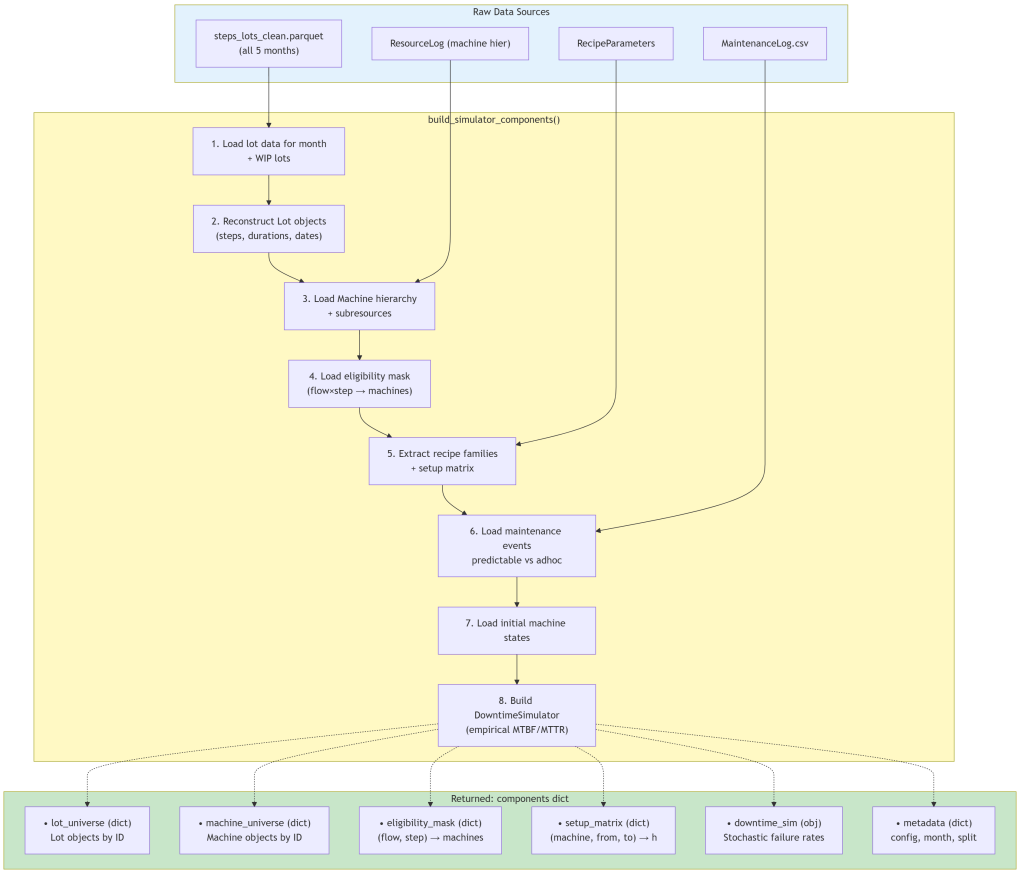

Figure A.1: Runtime call sequence of `build_simulator_components()`. Each internal step loads or reconstructs one artefact and passes it forward; the function returns a single `components` dictionary consumed by `MetaSelectorEnv` (Appendix **??**).

Quantity	Value
Cleaned lot-level events	7,653,932
Distinct lots	173,227
Distinct `FlowNames`	1,114
Substantial flows (median ≥ 4 steps)	174
Healthy flows (substantial $\cap$ stable)	29
Distinct `StepNames`	90
Main machines (`ParentResourcesCount = 0`)	58
Total resources (incl. subresources)	184
Eligibility mask pairs (full union)	2,854
Distinct recipe families	8,232
Recipe-setup matrix non-trivial triplets	1,526
`AbortProcess` events	30,367
`Hold` events	99,295
`Release` events	101,741
`Split` events	45,982
`LogEvent` maintenance events	134,705
Median maintenance interval (hours)	1.79
Mean maintenance interval (hours)	4.12
Simulation routing dataset (Dataset B, `steps_lots_clean.parquet`)	
Lot objects	3,883
Processing steps	11,234
Product flows	281
Eligibility mask pairs (filtered to Dataset B)	1,539
Training lots (Jan 2025)	1,857
Validation lots (Apr 2025)	604
Test lots (May 2025)	549

Table A.11: Final cleaned dataset summary.

Table A.12: Artefacts written to disk by the nine pipeline stages, by real filename.

Artefact (file)	Description	Stage
`steps_lots.parquet`	7.65M events, 173K lots, flattened lot-step table	1
`machine_states.parquet`	622 machines, 1,301 subresources	2
`batch_detection.csv`	Reverse-engineered batch capacity per machine	3
`machines_with_hierarchy.pkl`	Machine objects, hierarchy and batch flags, no maintenance	4
`maintenance_events.parquet`	Predictable vs. adhoc maintenance events, kept separate	5
`setup_families.parquet`	Recipe-family taxonomy from RecipeParameters	6
`setup_times.pkl`	1,526 setup-time triplets, from lot-data gaps	6
`recipe_mapping.pkl`	(material, step) $\rightarrow$ setup-family mapping	6
`downtime_stats.parquet`	Per-machine MTBF/MTTR statistics	7
`CandidateResources.csv`	Reformatted lot-history view, not an independent source	8
`machine_mask.pkl`	1,539 (flow, step, machine) eligibility entries	8
`data_clean/{train,validation}/YYYY-MM/`	Per-month partitions, two buckets only	9

Appendix B

Simulation Entity Models and Verification

This appendix presents the core entity models of the `FlowSimulator` environment and the verification procedures used to validate it. The simulation loop, two-stage dispatching engine, and Gymnasium interface are described in full in Chapter **??**; only the structural entity definitions and the validation methodology are reproduced here for reference.

B.1 Architecture and Design Rationale

The rationale for a custom, from-scratch discrete-event simulation engine over generic packages such as SimPy [**?**] is given in Section **??** of Chapter **??** and is not repeated here. The development itself followed the standard DES methodology outlined by Banks et al. [**?**], progressing from data collection and model construction through verification and experimental design.

B.2 Core Entities

The simulation model is built around two primary entity types, `Lot` and `Machine`. A `Lot` is the unit of dispatching; it carries its complete processing history and is matched to an eligible `Machine` at each decision epoch. The internal structure of the `Machine` object is described in Section **??**.

B.2.1 Lot

A `Lot` is the fundamental unit of dispatching in the simulation, reconstructed from MES Material-Log records during the data pipeline phase (Appendix **??**). Its dispatchability check, queue-time-constraint dictionary (populated but inactive, since explicit queue-time windows are not available in the source data), and equipment-dedication map (which restricts critical lithographic layers to their historically dedicated machine) are all described in Section **??** of Chapter **??** and are not repeated here.

B.2.2 Machine

A `Machine` object encapsulates all information relevant to scheduling decisions and physical resource availability. Its internal structure is organised into the five conceptual layers, hardware, recipe and setup, policy, state, and maintenance, illustrated in Figure **??** and described in full in Section **??** of Chapter **??**; they are not repeated here. Two details of that structure matter specifically for the validation results reported below: the recipe-and-setup layer feeds the three-strategy setup-time calculation described in Section **??**, and, during machine selection, the maintenance layer lets the dispatcher check whether a proposed processing end time would overlap with the next scheduled maintenance window, excluding the machine from the candidate set if a collision is detected.

Machine state transitions. The Machine object exposes a two-value state flag, `state`, taking the value `UP` or `DOWN` and governing availability; scheduled maintenance and unscheduled stochastic failures are both represented by the same `DOWN` value rather than by separate states. The four triggering transitions and Figure **??**, illustrating these five layers and the two-value state automaton, are given in Chapter **??** and are not repeated here to avoid duplicating the same material across chapter and appendix.

B.3 MES Operations and Exception Handling

A key fidelity requirement of the simulation is that it faithfully represents the MES exception events observed in the historical data. Six categories of operations are modelled:

These operations are not replayed from the historical data during RL training; rather, the simulation generates exception events stochastically based on empirical rates derived from the pipeline statistics, and the `Lot` object's exception-handling methods (`apply_hold`, `apply_release`, `apply_abort`, `apply_rework`, `apply_temporary_offflow`) are invoked when the corresponding scenario conditions are triggered. The delta-based reward signal (Section **??**) penalizes the agent in proportion to the incremental occurrence of each exception type during the episode.

B.4 Processing Time and Setup Time Prediction

The three-strategy prediction cascades for processing time (historical sampling, then a LightGBM model, then a median-table fallback) and for setup time (triplet lookup, then family-level mapping, then a fixed default) are described in Section **??** of Chapter **??** and are not repeated here. This section reports the validation evidence for the processing-time model that supports that description.

Processing time prediction accuracy. The LightGBM model is trained and evaluated on the 11,234 reconstructed step records using a 5-fold cross-validation on the training month (January 2025). Reported metrics are MAE = 1.83 h and RMSE = 3.41 h on held-out fold predictions.

Table B.1: MES operations implemented in `FlowSimulator`, with observed event frequencies in the 5-month production dataset. Event counts and percentages are derived from the full raw `MaterialLog` (11,384,882 rows) and include both lot-level and wafer-level events; they therefore differ from the cleaned lot-level counts reported in Table **??** and Table **??**.

Operation	**Category**	**Events**	**%**	**Simulation effect**
TrackIn	Processing	983,769	8.6	Records processing start; assigns lot to machine.
Terminate	Processing	622,643	5.5	Records processing end; advances step index.
TrackOut	Processing	221,358	1.9	Marks physical departure from machine.
Hold	Exception	99,295	0.87	Sets `on_hold=True`; blocks dispatching. Mean duration: 41.5h; median: 4.4h.
Release	Exception	101,741	0.89	Clears hold; lot re-enters dispatch queue.
AbortProcess	Exception	72,196	0.63	Wastes processing time; places lot on hold; marks last machine as aborted (lot must change machine).
Rework	Flow control	1,192	0.01	Redirects lot to repair flow; increments rework counter; restores to main flow after repair.
TemporaryOffFlow	Flow control	51,879	0.46	Branches lot to secondary flow (metrology, sampling); returns to main flow at recorded step.
ChangeFlowAndStep	Routing	469,493	4.12	Dynamic rerouting; 32.1% of lots experience at least one flow change.
Dispatch	Routing	492,251	4.32	Real MES machine assignment; used for eligibility mask validation.
Receive	Entry	32,857	0.29	Records fab entry time; used as cycle time origin when present.
MoveToNextStep	Routing	N/A	N/A	Advances lot without machine assignment (non-production steps).

The 25% Gaussian noise injected in the historical-sampling strategy is calibrated to span a comparable magnitude, ensuring that the noise envelope covers the observed prediction error. Lots for which historical sampling applies (i.e., where historical records exist for the same flow–step pair) represent 91.4% of all steps; the LightGBM model activates for the remaining 8.6%.

B.5 Verification and Validation

Verification and validation of the simulation engine were conducted through three complementary mechanisms: trace-level verification via a deterministic replay mode, eligibility-mask validation against historical machine assignments, and the aggregate KPI fidelity comparison of Table **??**. These are described in Section **??** of Chapter **??** and are not repeated here.

Initialization and warm-up. Each episode starts from an empty fab (cold start) with lots released according to their historical arrival schedule. No warm-up period is applied: the initial transient phase is part of the episode dynamics that the agent must learn to navigate. This design is consistent with the episodic RL formulation, where each episode represents an independent production month, and avoids the methodological complications of truncating a variable-length transient from the training signal.

Appendix C

State Feature Definitions

The complete 35-dimensional state vector is defined below. Features 0–27 constitute the core fab-level aggregates; feature 28 captures setup pressure; and features 29–34 are the six heuristic cost proxies introduced in Section **??**. All features are normalized using domain-calibrated constants derived from the production data statistics; an additional `VecNormalize` wrapper applies online running-mean normalization during training.

Normalization and distribution shift. The fixed normalization constants (e.g., $/500$ for lot counts, $/100$ for waiting times) were derived from the January 2025 training distribution. The test month (May 2025) exhibits a higher lot count (Table **??**: 51,804 lots vs 66,026 in the training month) and different maintenance density. Features whose values occasionally exceed the normalization constant in the test month are clipped at $[0, 1]$ by the `VecNormalize` wrapper, preventing out-of-distribution inputs to the policy network. This clipping is conservative: features that saturate at 1.0 lose discrimination above the training ceiling, but the `VecNormalize` running-mean layer partially compensates by re-centring the distribution during the test rollout. A direction for future work would be a more principled approach like online feature scaling during deployment, or a domain-adaptive normalization layer that adjusts to the test distribution without requiring retraining.

Feature multicollinearity. With 35 features and heuristic cost proxies derived from the same queue statistics that inform other features (e.g., `heuristic_cost_EDD` is a function of the same slack times used in features 2–4), partial multicollinearity is expected. The `VecNormalize` + PPO training pipeline is robust to correlated inputs because the policy gradient does not rely on feature orthogonality; however, multicollinearity may reduce the interpretability of gradient-based feature importance analyses. The state ablation in Section **??** provides empirical evidence that individual feature groups can be removed without large performance loss at low training budgets, suggesting that the information is distributed redundantly across the 35 dimensions rather than concentrated in a small subset, a property that aids robustness but reduces parsimony.

Table C.1: State vector: complete 35-feature specification with index, semantic category, description, and normalization constant.

Idx	Feature	Normalization
	Lot Counts	
0	`num_ready`: lots currently ready for dispatch	/500
1	`num_active`: lots still in the system (ready, in process, or on hold)	/500
	Slack Time (hours)	
2	`avg_slack`: mean slack across active lots	/500
3	`min_slack`: minimum slack across active lots	/500
4	`max_slack`: maximum slack across active lots	/500
	Waiting	
5	`avg_wait`: mean waiting time in queue	/100
	Tardiness & Completion	
6	`total_tard`: cumulative tardiness this episode	/1000
7	`completion_rate`: count of lots completed so far this episode (not a $[0,1]$ fraction despite the name; normalised by the same lot-count constant as indices 0–1)	/500
8	`on_time_rate`: on-time completions / completions so far	$[0,1]$
	Reserved (excluded from the categories above)	
9–10	`current_hour`, `current_weekday`: zeroed at runtime, excluded after a SHAP check (Chapter **??**)	—
	WIP Structure	
11	`frac_near_due`: active lots with deadline <24h / active lots	$[0,1]$
12	`avg_remaining`: mean remaining steps / episode step budget (5,000)	$[0,1]$
13	`frac_on_hold`: active lots on hold / active lots	$[0,1]$
	Urgency	
14	`avg_cr`: mean critical ratio over ready lots	/5
15	`p25_slack`: 25th-percentile slack over ready lots	/500
16	`p75_slack`: 75th-percentile slack over ready lots	/500
17	`urgency`: ready lots with CR < 1 / ready lots	$[0,1]$
18	`bk_sev`: coefficient of variation of critical ratios over ready lots (bottleneck severity proxy)	raw
	Trends	
19	`comp_accel`: change in `completion_rate` since the previous step	raw
20	`tard_momentum`: change in cumulative tardiness since the previous step	/1000
21	`queue_pressure`: ready lots (same count as index 0, re-normalised for trend context)	/500
	Fab Disturbance	
22	`frac_down`: machines currently `DOWN` / total machines	$[0,1]$

continued on next page

Table C.1 – continued from previous page

Idx	Feature	Normalization
23	`ep_aborts_norm`: cumulative aborts this episode	$/50$
24	`ep_reworks_norm`: cumulative reworks this episode	$/20$
25	`hot_lot_frac`: hot lots / active lots	$[0,1]$
26	`machine_load_cv`: coefficient of variation of per-machine queue loads	raw
27	`throughput_rate`: completed lots per elapsed hour, scaled by $1/10$	$[0,1]$, capped
	Setup Pressure	
28	`setup_pressure`: mean setup time / mean processing time, ready lots	$[0,1]$
	Heuristic Cost Proxies (shared formula, Section **??***)*	
29	`heuristic_cost_FIFO` (candidate previewed under FIFO)	$[0,1]$
30	`heuristic_cost_SPT` (candidate previewed under SPT)	$[0,1]$
31	`heuristic_cost_EDD` (candidate previewed under EDD)	$[0,1]$
32	`heuristic_cost_LST` (candidate previewed under LST)	$[0,1]$
33	`heuristic_cost_CR` (candidate previewed under CR)	$[0,1]$
34	`heuristic_cost_ATCS` (candidate previewed under ATCS)	$[0,1]$

How the six proxies are actually computed. The method `MetaSelectorEnv._compute_per_rule_costs()` calls `Dispatcher.preview_choice_for_rule(rule)` once for each of the six rules (Appendix **??**), obtaining, without committing any assignment, the (lot, machine) pair that rule would currently select. Each of these six candidates is then passed through the *same* function, `_compute_decision_cost()`, so the six features differ only in which candidate is scored, never in how it is scored. That function computes

$$\text{cost}(a_k, o_t) = 0.05\,\sigma_k + 0.01\,w_k + 0.10\,\tau_k,$$

where σ_k is the predicted setup time and w_k the lot's current waiting time, both read directly off the previewed candidate, and $\tau_k = \max(0,\,(t_{\text{now}} + \sigma_k + p_k) - d)$ is the tardiness that would result if the candidate lot (predicted processing time p_k, due date d) were dispatched now under rule a_k. A fourth term, `_compute_urgency_skip()`, measures how much more urgent the most urgent *skipped* lot was relative to the one the rule chose, i.e. a penalty for passing over a more urgent lot; it is computed on every call but its contribution is commented out of the returned sum (`disabled for an ablation variant` in the source), so it never affects the value written to o_t.

This shared-cost design has a second, load-bearing role beyond the observation vector: the same dictionary of six per-rule costs, `per_rule_costs`, is passed to `_select_best_rule()`, which returns whichever rule currently has the lowest cost, and to `_compute_regret()`, which computes the gap between the agent's chosen rule's cost and that best rule's cost. This regret value is exactly what feeds the B4/B5 shaping term Φ_{regret} of Section **??** in Chapter **??**. In other words,

the heuristic cost proxies in the observation vector and the regret signal in the reward are not two independent mechanisms: they are two different consumers of the same six numbers computed once per step, which is why the agent can, in principle, learn to anticipate the regret-shaped penalty directly from its own observation before the reward is even returned.

Appendix D

Reward Function Calibration

The coefficients of the primary reward variant B0 (Equation **??**) were calibrated to reflect the manufacturing cost hierarchy observed in the production dataset. The calibration procedure follows two principles: (i) the relative magnitude of each coefficient should be proportional to the operational impact of the corresponding exception event, and (ii) the aggregate reward signal should remain in a bounded range $[-50, +15]$ per step during early training to prevent gradient explosion.

Table **??** extends the coefficient summary from Section **??** with the observed event frequencies that informed the weight selection.
The final weight vector was validated by inspecting reward traces during 50 pilot training episodes: the mean per-step reward under the FIFO baseline fell in the range $[-8, -2]$, consistent with the target $[-50, +15]$ bound, and no single exception type dominated the gradient signal across scenarios.

Calibration assumption: aggregate vs. per-scenario frequencies. The event frequencies in Table **??** are computed over the full five-month dataset and therefore represent average operating conditions. Under adversarial Catalogue B scenarios, particularly *bottleneck_failures* (MTBF $\times 1/3$, MTTR $\times 2$) and *maintenance_hot_lots* (MTBF $\times 1/3$, MTTR $\times 2$, hot-lot 30%), the per-episode abort and hold rates can be substantially higher than the dataset averages. For example, under *bottleneck_failures*, machine downtime increases approximately threefold, which may elevate abort events from $\sim$0.1% of steps to $\sim$0.3%, proportionally amplifying the abort coefficient's contribution to the gradient. This amplification is intentional in adversarial scenarios (the agent should be strongly penalised for aborts under stress), but may shift the relative weighting between reward terms across scenarios in ways not fully captured by the aggregate calibration. Scenario-specific reward coefficient tuning, or adaptive scaling of ζ_{abort} as a function of current abort rate, is identified as a direction for future reward engineering work.

Coefficient sensitivity. The ablation between reward variants B2–B4 (Section **??**) provides evidence of sensitivity to the setup penalty coefficient (γ_{setup}): introducing it from B2 to B3 reduces

Table D.1: Reward coefficient calibration: values, observed event frequencies from the 5-month production dataset, and derivation rationale.

Term	**Symbol**	**Value**	**Freq. (%)**	**Calibration rationale**
Waiting time	α	0.05	–	Small continuous penalty per waiting-hour; scaled so that a 10h queue delay contributes -0.5 to the step reward.
Setup time	γ_{setup}	0.10	–	Double the waiting penalty to reflect the direct machine downtime during changeover; median setup is 1.01h, contributing ≈ -0.1 per step.
Hold	γ_{hold}	1.0	0.88	Throughput fully blocked for mean 41.5h; unit penalty signals one significant disruption per event.
Off-flow	γ_{offload}	0.5	1.95	Higher frequency than hold but shorter average duration; half-weight reflects secondary disruption.
Rework	δ_{rework}	3.0	0.19	Despite low frequency, rework incurs an extra full processing cycle plus yield loss; elevated weight reflects this cost.
Abort	ζ_{abort}	10.0	7.87	Entire lot destroyed; highest single-event cost in the manufacturing process. Set to $10\times$ the hold penalty to dominate the reward signal when aborts occur.
On-time bonus	B_t^+	+10	–	Symmetric to the abort penalty; ensures on-time completion yields a reward comparable in magnitude to avoiding one abort.
Tardiness mult.	B_t^-	2.0	–	Per-hour-late penalty; a 5h delay yields -10, balancing the on-time bonus.

OTR by 0.24 pp without improving tardiness, indicating over-penalisation. A comparable sensitivity analysis for the base coefficients (α, γ_{hold}, ζ_{abort}) was not performed due to training budget constraints; the B0–B5 ablation was designed to isolate reward structure rather than coefficient magnitude. A systematic grid search over $\pm 50\%$ perturbations of each base coefficient at 200K timesteps would characterise the robustness of the selected weight vector and is recommended before deployment in a live MES context.

Appendix E

Per-Scenario Catalogue A Results

Table **??** in Chapter **??** reports Catalogue A performance as a five-scenario aggregate. This appendix decomposes that aggregate scenario by scenario, so that the source of the RL policy's mean tardiness advantage can be inspected directly. Tables **??** and **??** report, respectively, the on-time rate and the average tardiness of each policy in each of the five Catalogue A scenarios (30 test episodes each, May 2025). The best value in each scenario is shown in bold.

Table E.1: On-time rate (%) per Catalogue A scenario (30 episodes each). Best value per scenario in bold.

Policy	**normal**	**setup_heavy**	**tight_dl**	**high_urg**	**setup_slack**
FIFO	46.4	46.6	46.2	45.5	46.2
SPT	**47.1**	45.6	45.5	46.2	**46.9**
EDD	46.0	**46.7**	**47.1**	46.6	45.3
LST	45.5	46.2	47.0	46.3	46.6
CR	44.6	45.9	44.9	46.8	45.2
RL B4($w = 0.3$)	46.8	45.4	45.9	**47.1**	44.9

Table E.2: Average tardiness (h) per Catalogue A scenario (30 episodes each). Lowest value per scenario in bold.

Policy	**normal**	**setup_heavy**	**tight_dl**	**high_urg**	**setup_slack**
FIFO	152.8	487.5	183.1	663.9	426.0
SPT	**137.2**	270.8	**99.7**	422.7	601.9
EDD	278.1	292.0	180.2	**256.6**	433.7
LST	234.4	294.5	234.6	295.8	463.0
CR	267.5	402.1	283.5	280.7	216.7
RL B4($w = 0.3$)	191.2	**189.1**	271.0	298.4	**121.2**

The decomposition shows that the RL policy's aggregate tardiness advantage is concentrated in the two setup-constrained scenarios: under *setup_heavy*, the policy achieves the lowest tardiness of any policy (189.1 h, against 487.5 h for FIFO and 270.8 h for SPT), and under *setup_and_slack* it again leads by a wide margin (121.2 h, against 216.7 h for the next-best rule, CR). In the remaining three scenarios the policy is competitive but not dominant: SPT achieves the lowest

tardiness under *normal* and *tight_deadlines*, and EDD under *high_urgency*, with the RL policy placing mid-field. This localisation is consistent with the non-significant pooled Wilcoxon test (Table **??**) and is, in itself, an interpretable result: the meta-dispatcher's adaptive rule-switching pays off precisely where no single static rule handles the combined setup-and-slack trade-off well, while adding little over a good static rule under conditions that one rule already serves well. On-time rate, by contrast, is tightly clustered in every scenario (44.6–47.1%), confirming that the policies are separated mainly on tardiness rather than on raw delivery count.

Appendix F

Code Structure

This appendix documents the module organization of the meta-dispatcher implementation, to support reproducibility. It complements Chapter **??** by showing how the RL-facing code, the dispatcher, and the simulator are decoupled, and by giving short illustrative snippets of each module's core interface rather than full source listings.

F.1 Module Overview

- `sim_builder.py` loads the pipeline artifacts (lot objects, machine hierarchy, eligibility mask, recipe-setup matrix) and assembles them into the `components` dictionary. Two functions do this work: `build_simulator_components()` builds the dictionary once, and `create_simulator_instance()` turns it into a fresh `FlowSimulator` instance on every episode reset.
- `flow_simulator.py` implements the `FlowSimulator` class, the discrete-event engine that owns the simulation clock, machine pool, lot queue, and maintenance and downtime scheduling.
- `dispatcher.py` implements the `Dispatcher` class, which scores and selects lots by the active heuristic (Stage 1) and filters and ranks eligible machines (Stage 2), computing setup times from the recipe-family matrix along the way.
- `meta_selector_env.py` implements `MetaSelectorEnv`, the Gymnasium wrapper that exposes the simulator and dispatcher as a `reset()`/`step(action)` RL environment, producing the observation vector and the reward variants B0–B5.
- `train_meta_selector.py` is the training entry point that builds simulator components, wraps them in `MetaSelectorEnv` and `VecNormalize`, and runs PPO (Stable-Baselines3) for a given number of timesteps.

Below, each module's core interface is illustrated with a short signature and its immediate purpose, not a full listing.

```
def build_simulator_components(month="2025-02", split="train", ...):
    """Load pipeline artifacts and assemble the components dict."""
    cache_key = (month, split)
    if use_cache and cache_key in _COMPONENTS_CACHE:
        return _COMPONENTS_CACHE[cache_key]
```

```
class FlowSimulator:
    def __init__(self, machines, mask, processing_times, ...):
        self.machines = machines
        self.mask = mask
        self.now = start_time or pd.Timestamp.now(tz="UTC")
```

```
class Dispatcher:
    def _filter_processable(self, lots, now):
        """Filter lots that have at least one available machine."""
        processable = [l for l in lots if self._has_eligible_up_machine(l, now)]
        return processable
```

The function above, `_filter_processable`, is an internal helper: it keeps only the lots that have at least one eligible machine currently up and free. It is called by `preview_choice_for_rule()` (and by `choose_batch()`) before the remaining lots are scored by the active heuristic. That caller, `preview_choice_for_rule()`, is in turn the method `MetaSelectorEnv` actually calls at each step, as shown next; its own internal sequence of calls is detailed in Section **??**.

```
def _maybe_inject_exception(self, lot, completed_step_idx):
    """Sample MES exceptions after a completed step
    (skipped in replay mode)."""
    step_name = lot.real_steps[completed_step_idx]
    probs = self.exception_probs.get(
        (lot.flow_name, step_name), {})
    exc = {
        'abort':   random.random() < probs.get('abort_prob', 0),
        'hold':    random.random() < probs.get('hold_prob', 0),
        'rework':  random.random() < probs.get('rework_prob', 0),
        'offflow': random.random() < probs.get('offflow_prob', 0),
    }
    if exc['abort']:
        ...  # revert step, schedule repair, return
    if exc['hold'] and not lot.is_terminated:
        ...  # block lot, schedule release
    if (exc['rework'] and not lot.is_terminated
            and not lot.on_hold):
        ...  # re-queue prior step, once per step
    if (exc['offflow'] and not lot.is_terminated
            and not lot.on_hold):
```

```
        ...  # shift ready_ts forward, no hold
```

`_maybe_inject_exception()` is the concrete sampler behind the "stochastic injection" described in Section **??** of Chapter **??**: at every completed step it draws four *independent* Bernoulli variables, one per exception type, each compared against that (flow, step) pair's own empirical rate from Table **??**. Because the draws are independent, more than one could in principle come out true in the same call; the function resolves this not with an explicit priority switch but with a chain of mutual gates. Abort is checked first and, if sampled, reverts `lot.current_step_index` to repeat the step, schedules a `lot_release` after a repair delay drawn uniformly from $[2, 12]$ hours, and returns immediately, so no other exception type is evaluated on that call. Abort is additionally capped at two occurrences of the same step index; a third sampled abort on the same step is silently ignored and the lot is allowed to proceed, which prevents a rare (flow, step) pair with an artefactually high abort rate from looping the lot forever. If abort did not fire, hold is applied next when sampled, blocking the lot (`on_hold=True`) until its own `lot_release` event. Rework and off-flow are then gated on `not lot.on_hold`: they only take effect if the hold check above did not just block the lot, so in practice at most one of hold, rework, or off-flow is ever applied per call, and rework is further capped at one occurrence per step index via a `_reworked_at` marker. In replay mode (Appendix **??**) this entire function is a no-op, since exceptions observed historically are already implicit in the replayed sequence of dispatch events rather than sampled.

```python
class MetaSelectorEnv(gym.Env):
    def step(self, action: int):
        rule = self.HEURISTICS[int(np.asarray(action).item())]
        candidate = self.dispatcher.preview_choice_for_rule(rule)
        # ... schedule, compute reward, advance clock ...
```

```python
def train_ppo(total_timesteps=300_000, n_envs=4, ...):
    """PPO training entry point (Stable-Baselines3)."""
    env = make_vec_env(make_env(components), n_envs=n_envs)
    model = PPO("MlpPolicy", env, ...)
```

The `total_timesteps=300,000` shown above is a generic upper bound built into the function signature for exploratory runs. All experiments reported in this thesis were run with `total_timesteps=200,000` passed explicitly, matching Table **??** in Appendix **??**.

The training loop itself is not custom code. The rollout-collection, GAE, epoch, and minibatch update cycle illustrated in Figure **??** of Chapter **??** is Stable-Baselines3's internal `PPO.learn()` implementation; nothing in this codebase reimplements it. Training is launched with a single call to `model.learn()`, passing the total timestep budget and the three callbacks below, and everything this thesis's code contributes to the loop is the hyperparameter configuration passed to `PPO(...)` (Table **??**) plus three callbacks, each a hook that `PPO.learn()` invokes at fixed points without altering the loop's control flow:

- `EvalCallback` runs 3 deterministic evaluation episodes on the held-out validation environment roughly once per rollout (every $\max(\texttt{n\_steps} \times \texttt{n\_envs}, 4096)$ steps) and saves the checkpoint with the best validation on-time rate to a separate `best_model/` directory, independently of the regular checkpoint schedule below.
- `CheckpointCallback` periodically saves the current model to disk on its own schedule, regardless of validation performance, so training can be resumed or a specific timestep inspected after the fact.
- `RegretLoggingCallback` reads, from the `info` dictionary that `MetaSelectorEnv.step()` returns at every environment step, the per-step regret value, whether the chosen rule matched the locally best rule, and the per-rule heuristic-cost sums, and logs the aggregated diagnostics to TensorBoard; this is the mechanism behind the regret-shaping diagnostics referenced for the B4 and B5 variants in Chapter **??**.

Evaluation pipeline: where the results in Chapter ?? come from. Every performance, rule-selection, and Wilcoxon-test result reported in Chapter **??** is produced by a single script, `evaluate_policy.py`, distinct from the training entry points described above. Its core function, `evaluate_model_per_scenario()`, takes a trained checkpoint, an evaluation month, an episode count, and a data split as arguments; it loads the checkpoint and runs the requested number of deterministic evaluation episodes per scenario (30 for all results reported in Chapter **??** unless stated otherwise), logging one row per dispatch step to `eval_per_step_ppo.csv`. Its columns are the rule chosen, the reward, and per-step tardiness, on-time, cycle-time, flow-factor, WIP, utilisation, and makespan values, from which every per-scenario rule-usage percentage in this thesis is aggregated. A companion `eval_summary_table.csv` reports one row per scenario with the mean of each column and a `dominant_rule` field, computed independently of the per-step aggregation and used throughout this thesis as a cross-check on it. The data-split argument is always set to `validation` for reported thesis results, i.e. the held-out May 2025 test month (Section **??** of Chapter **??**), never the January 2025 training month. The five static baselines (FIFO, SPT, EDD, LST, CR) are evaluated by the same script under an identical fixed sequence of episode seeds within each scenario, which is what allows the paired Wilcoxon test of Table **??** to pair RL and baseline episodes by index rather than by summary statistics alone.

A second, separate training entry point: `train_rule_selector.py`. The four regime-variant checkpoints analysed in Section **??** of Chapter **??** (`B4_smooth`, `B4_normal`, `B4_complex`, `B4_mixed`) are produced by `src/rl/train_rule_selector.py`, a script separate from `train_meta_selector.py` above and not otherwise used in this thesis. Its function `train_ppo_rule_selector()` takes only a training month and an output directory as arguments; internally, it always builds its environment through `make_single_scenario_env()`, whose scenario and reward-variant parameters default to `"normal"` and `"B0"` and are not exposed as

arguments to `train_ppo_rule_selector()` itself. This is why the four checkpoints share an identical training configuration despite their regime-suggestive names, as discussed in Chapter **??**.

F.2 Module Dependencies

This section traces the concrete call sequence behind a single training run, from the top-level script down to the simulator's event calendar, so that the two-stage selection described in Section **??** of Chapter **??** can be followed against the actual code. It is organised in two parts: how the objects are wired together once, at startup, and what happens on every call to `step(action)` afterwards.

Startup: wiring the objects together. The training entry point, `train_meta_selector.py`, orchestrates the whole run. It calls `build_simulator_components()` (`sim_builder.py`) once, to load pipeline artifacts into the `components` dictionary, then passes that dictionary into the `MetaSelectorEnv` constructor (`meta_selector_env.py`). Internally, `MetaSelectorEnv.reset()` calls `create_simulator_instance()` (also in `sim_builder.py`) to copy the `components` into a fresh `FlowSimulator` (`flow_simulator.py`), and instantiates a `Dispatcher` (`dispatcher.py`) bound to that simulator instance. This gives the object graph used by every subsequent step: `MetaSelectorEnv` holds the `Dispatcher`, which holds the `FlowSimulator`.

Per-step call sequence. Each call to `step(action)` passes through four groups of calls, always in the same order:

1. **Selecting a lot and a machine.** `MetaSelectorEnv` translates the discrete action into the named heuristic (one of FIFO, SPT, EDD, LST, CR, ATCS) and calls `Dispatcher.preview_choice_for_rule(rule)`, a thin wrapper around the internal helper `_select_batch()`. This helper carries out the two-stage selection as a fixed sequence of sub-calls: it collects the lots currently ready to dispatch (`_available_lots()`), keeps only those with at least one eligible machine that is up and free (`_filter_processable()`), gives absolute priority to any lot whose queue-time constraint is about to expire (`_separate_critical_lots()`), and otherwise scores the remaining lots with the active rule's priority function (`_apply_heuristic()`). It then selects a candidate machine for the winning lot (`choose_machine()`) and assembles any further lots eligible to share that machine's occupation into a single batch (`form_batch()`); the predicted processing time returned for the batch is the maximum over its member lots, since a shared occupation (e.g. a furnace run) is limited by its slowest lot. Before any of this scoring happens, `Dispatcher._process_maintenances()` sweeps every machine object and starts or ends a maintenance window whenever due, so machines already down are excluded from the candidate set.

2. **Committing the assignment.** Once `preview_choice_for_rule()` returns a candidate, `MetaSelectorEnv` calls `sim.schedule_single()` (or `sim.schedule_bat`

ch() for a batch) to commit it. This books the machine, appends a new entry to the lot's processing history, advances lot.current_step_index, and sets machine.available_at and lot.ready_ts to the computed completion time.

3. **Sampling exceptions and scheduling events.** The same commit call stochastically applies a hold, abort, or rework exception via _maybe_inject_exception() (Section **??**), pushes the resulting machine_free and lot_ready events onto the calendar, and sets _state_dirty=True to force a cache recomputation.

4. **Advancing the clock and returning control.** Control then passes to sim.run_until_next_decision_point() (Section **??**), which advances the simulation clock via _process_job_completions() (releasing any load port or chamber whose job has just finished) and _process_pending_events() (handling machine failures, maintenance boundaries, and delayed lot releases). It repeats this until has_actionable_pair(), a cached check using the same lot- and machine-filtering logic as _select_batch(), confirms that a ready lot and an available eligible machine exist together. Only then does MetaSelectorEnv extract the next observation and return control to the agent.

Across all four groups, the dependency is strictly one-directional: meta_selector_env.py calls into dispatcher.py, which calls into flow_simulator.py. The Dispatcher never calls back into the environment, which keeps the RL-facing code decoupled from dispatch mechanics.

F.2.1 Event Calendar and Decision-Point Detection

The dependency chain above leaves one internal mechanism unexplained: the logic that decides, once a dispatch decision is committed, when control should return to the agent. This is the role of run_until_next_decision_point(), the loop inside flow_simulator.py that drives the discrete-event calendar forward until a new actionable pair exists.

The simulator's event data structures support this loop directly: FlowSimulator keeps a min-heap of pending events, _pending_events, alongside a boolean flag, _state_dirty, that marks whether the cached result of has_actionable_pair() is still valid. Recomputing that pair on every iteration of the event loop would be wasteful, since most individual events change only a small part of the fab state. The pair is therefore recomputed only when _state_dirty is set, which happens whenever an event of consequence is processed: a job completion, a machine recovery, a lot release from hold, a lot_ready signal, or a machine_free signal. The loop itself, run_until_next_decision_point(), repeats four steps, up to max_iterations=50,000 times, until an actionable pair is found or no events remain: it processes every event scheduled at the current simulation time, through _process_job_completions() followed by _process_pending_events(); checks whether all lots have terminated, returning False if so; tests has_actionable_pair() against the cache, returning True if an actionable pair exists; and otherwise advances self.now to the timestamp of the next pending event (via _a

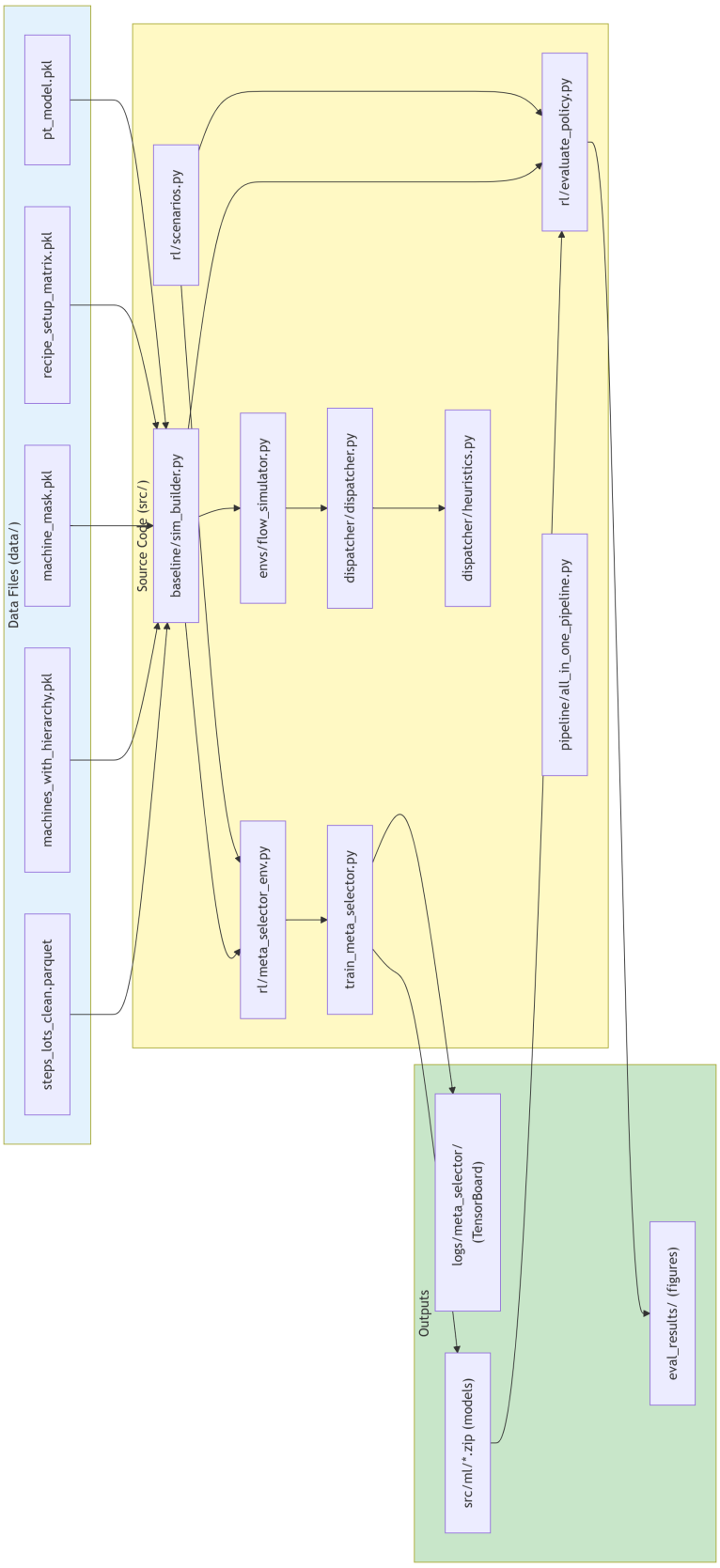

Figure F.1: Module dependency graph. Data files feed into `sim_builder.py`, which provides components to the environment stack. The dispatcher, heuristics, and scenarios are independent modules consumed by `MetaSelectorEnv`. The training and evaluation scripts drive the overall pipeline.

dvance_clock_only()) before repeating. The clock is therefore always advanced to a real, upcoming event time, never by a fixed increment. If the 50,000-iteration budget is exhausted without an actionable pair appearing, the function returns `False` exactly as it would on natural completion; `MetaSelectorEnv` does not distinguish the two cases and marks the episode `done` in either case, rather than forcing the simulation to continue. Table **??** lists the event types handled by `_process_pending_events()`.

Event type	Effect
`unscheduled_down`	Machine failure mid-job. Aborts the lot in process, schedules the machine's recovery (`unscheduled_up`), and pushes a `lot_release` event after a short investigation hold.
`unscheduled_up`	Machine recovery from an unscheduled failure. Marks the cache dirty, since the recovered machine may now form an actionable pair.
`maintenance_begin`	Start of a scheduled maintenance window. Aborts any lot whose job would collide with the window and pushes a `lot_release` event timed to the window's end.
`maintenance_end`	End of a scheduled maintenance window. Marks the cache dirty, mirroring `unscheduled_up`.
`lot_release`	A lot previously held, whether by `_maybe_inject_exception()` (Section **??**) or by an abort under one of the two events above, is released back into the dispatch pool.
`lot_ready`	Clock signal marking that a lot has reached the end of its current processing step.
`machine_free`	Clock signal marking that a machine has completed its subresource-release sequence and is available again.
`background_job_begin`	Phantom occupation of a machine by background fab traffic not tied to any tracked lot.

Table F.1: Event types handled by `_process_pending_events()`.

Two of these event types, `lot_ready` and `machine_free`, warrant a specific clarification, since their names could suggest that firing the event is what makes a lot or machine available. It is not. Both are clock signals rather than commands, and the distinction matters for tracing the code correctly. When `schedule_single()` commits a dispatch decision at time T_0, it immediately updates `Lot.ready_ts` and `Machine.available_at` to the computed completion time T_1, calls `_maybe_inject_exception()` to stochastically apply a hold, abort, or rework at that step (Section **??**), pushes a `lot_ready` event onto the heap scheduled for T_1, and sets `_state_dirty=True`. All state relevant to future availability is therefore already written at T_0. The event that fires later, once the calendar clock reaches T_1, does not modify `Lot` or `Machine` state a second time. Its only effect is to set `_state_dirty=True` again, forcing `has_actionable_pair()` to recompute now that the clock has caught up with the timestamp already recorded in `ready_ts`. The event, in short, exists to trigger cache invalidation at the correct simulated time, not to perform the state transition itself.

This mechanism enforces a clean boundary between successive calls to `step()`. Every event due by the moment a new actionable pair appears is fully processed, through `_process_job_completions()` and `_process_pending_events()`, before `MetaSelectorEnv` extracts the

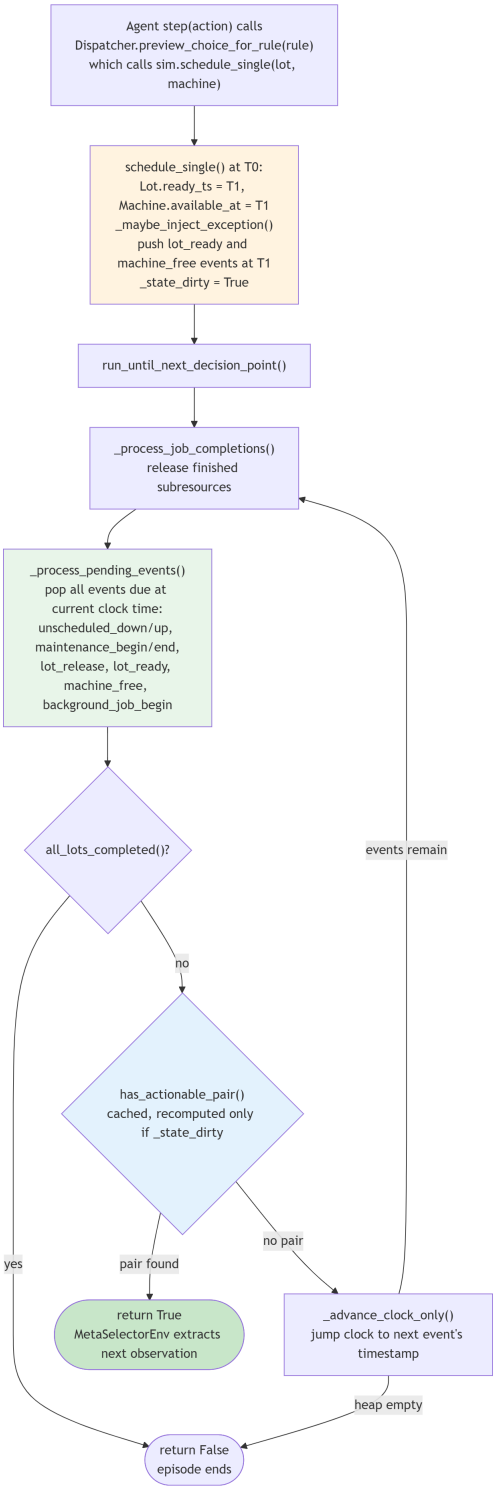

Figure F.2: Event calendar and decision-point loop. `run_until_next_decision_point()` processes all due events, checks for termination, and tests the cached actionable-pair flag before advancing the clock, repeating until a new decision point is reached.

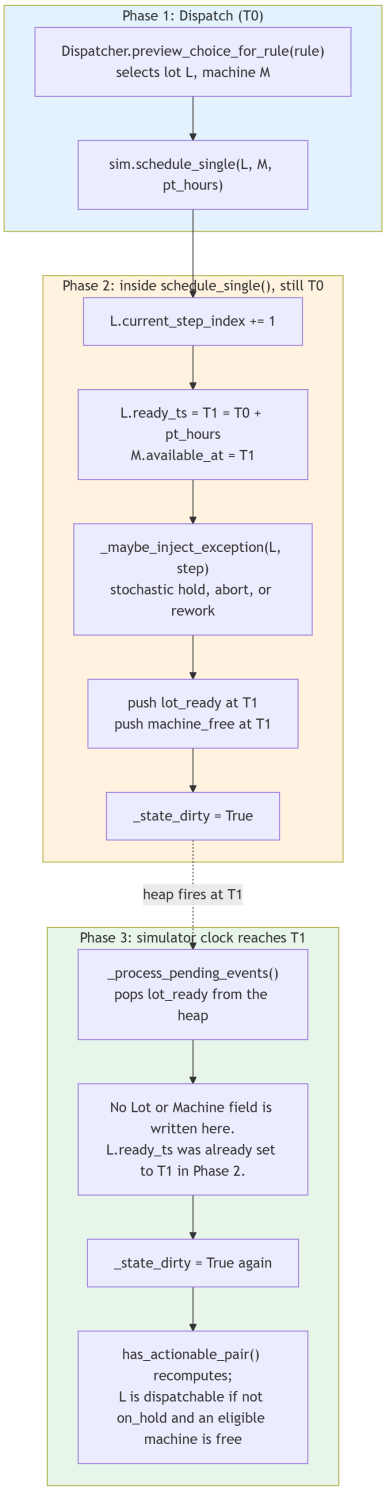

Figure F.3: `lot_ready` event lifecycle across its three phases. All state relevant to future dispatchability is written in Phase 2, at T_0. The event that fires later, in Phase 3, only invalidates the actionable-pair cache.

next observation and returns control to the agent. This guarantees that decision points are never presented out of order and that a partially applied event never leaks into the observation the agent receives. It is a property of decision-point sequencing internal to the simulator, and should not be conflated with the observability question addressed in Section **??**: the event calendar guarantees *when* the agent is allowed to act, not that the resulting observation o_{t+1} is a sufficient statistic of the full, unobserved simulator state.

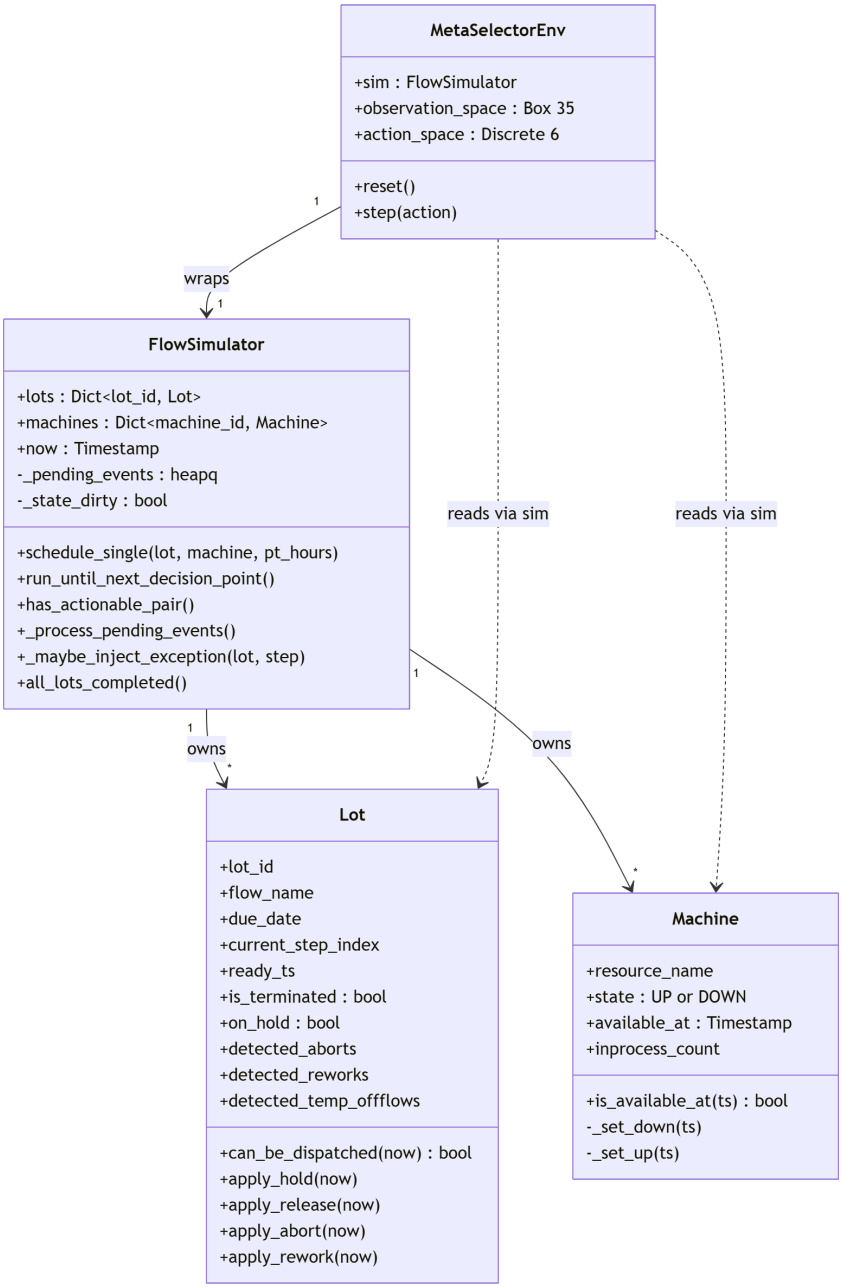

Figure F.4: Class interactions among `FlowSimulator`, `Lot`, `Machine`, and `MetaSelectorEnv`. `FlowSimulator` owns the lot and machine collections and the event calendar; `MetaSelectorEnv` wraps a single `FlowSimulator` instance and never manipulates `Lot` or `Machine` objects directly.

Appendix G

Hyperparameters and Configuration Tables

This appendix collects the full hyperparameter and configuration values referenced but not fully tabulated in Chapter **??**, to support exact reproduction of the reported experiments.

G.1 PPO Hyperparameters (Full)

Table **??** expands Table **??** with parameters omitted from the main chapter for brevity.

G.2 Scenario Configuration Values

Table **??** lists the exact parameter values for each of the ten training scenarios sampled by ScenarioMixEnv (Section **??**), plus the unmodified normal scenario held out for validation. Setup scale and due-date factor are multiplicative factors applied to the corresponding base simulator parameter (lower due-date factor means tighter deadlines); breakdown scale multiplies the machine failure rate.

The due-date factor scales the anchor slack window used to compute each lot's due date. Hot lots (fab-disturbance scenarios only) instead receive a due date at 3.5× their remaining processing time, versus 20× for non-hot lots in the same episode, and the two hot-lot scenarios plus *Maintenance + hot-lots* additionally disable stochastic MES exception injection so the urgency signal is not diluted by unrelated noise. †*Scheduled maintenance* doubles planned-maintenance density rather than the unscheduled breakdown rate reported in the Breakdown scale column.

Table G.1: PPO hyperparameters and training configuration (full).

Parameter	**Value**	**Notes**
Learning rate α	10^{-4}	Constant schedule (no decay)
Discount factor γ	0.99	
GAE λ	0.95	
Clip range ε	0.2	
Rollout per env.	512	PPO horizon per environment (`n_steps`)
Mini-batch size	64	
Epochs per rollout	10	
Entropy coeff.	0.03	Elevated to prevent premature collapse onto SPT
Value function coeff.	0.25	Set explicitly, below the Stable-Baselines3 default of 0.5
Max gradient norm	0.5	Gradient clipping
Parallel envs.	4	CPU-bound (`DummyVecEnv`)
Total timesteps	200,000	
Random seed	0	Fixed across reported runs
Device	CPU	No GPU used for training
VecNormalize clip	± 10	Observation and reward clipping
Eval frequency	$\sim$4096 steps	$\max(\texttt{n\_steps} \times \texttt{n\_envs}, 4096)$ steps; approximately once per rollout
Checkpoint frequency	$\sim$512 steps	$\max(1000/\texttt{n\_envs}, \texttt{n\_steps})$ steps; approximately once per rollout, best-model selection by validation OTR

Table G.2: Training scenario configuration values.

Scenario	**Setup scale**	**Due-date factor**	**Hot-lot fraction**	**Breakdown scale**
Normal (validation)	1.0	1.0	0.0	1.0
Rule-regime scenarios				
Setup-heavy	5.0	1.0	0.0	1.0
Tight-deadlines	1.0	0.50	0.0	1.0
High-urgency	1.0	0.30	0.0	1.0
Setup-and-slack	4.0	0.60	0.0	1.0
Fab-disturbance scenarios				
Hot-lots (15%)	1.0	1.0	0.15	1.0
Hot-lots (30%)	1.0	1.0	0.30	1.0
Maintenance + hot-lots	1.0	1.0	0.30	3.0
High-mix pressure	2.5	0.70	0.0	1.0
Bottleneck failures	1.0	1.0	0.0	3.0
Scheduled maintenance†	1.0	0.80	0.0	1.0