ANY-SCALE FLEXIBLE JOB SHOP SCHEDULING VIA DUAL-ATTENTION PROXIMAL POLICY OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Flexible job shop scheduling (FJSS) plays a critical role in modern intelligent manufacturing systems. With the rapid development of industrial production environments, learning-based scheduling has emerged as a promising paradigm for learning effective scheduling policies from data and enabling fast decision-making. However, existing methods struggle to generalize across variable-sized instances, learn heterogeneous interactions among operations, machines, and processing-time information, and generate reliable decisions under dynamic scheduling constraints. To address these challenges, we propose a dual-attention proximal policy optimization framework for pre-training a generalizable neural scheduler across FJSS scales. Specifically, we introduce a dual-attention representation mechanism that explicitly models the heterogeneous interactions among operations, machines, and dynamically updated completion-time information. An online mask further incorporates scheduling information during sequential decision-making by masking infeasible actions, thereby improving decision reliability and training efficiency. The proposed framework is pre-trained on a large collection of synthetic FJSS instances with diverse job-machine configurations, using high-quality Google Operations Research Tools solutions to construct informative reward and advantage signals. Extensive experiments on 217 FJSS instances from six benchmark families and 129 large-scale synthetic instances, covering problem sizes from 3×2 to 100×100 (job by machine), demonstrate strong cross-family and cross-scale generalization, with the proposed framework consistently outperforming representative heuristics and state-of-the-art learning-based methods on unseen instances.

1 INTRODUCTION

Flexible job shop scheduling (FJSS) is a fundamental combinatorial optimization problem that arises widely in modern manufacturing systems such as smart manufacturing (Zhang et al., 2024), cloud computing (Sun et al., 2026), and semiconductor production (Liu et al., 2024). Unlike classical job shop scheduling, FJSS allows each operation to be processed by multiple alternative machines, resulting in a significantly larger solution space and more complicated operation-machine assignment decisions. Efficiently solving FJSS is therefore essential for improving production efficiency and resource utilization in modern manufacturing systems (Dauzère-Pérès et al., 2024).

The rapid development of smart manufacturing systems has introduced increasingly large-scale tasks and diverse scheduling environments (Li et al., 2025b), where production systems from different scenarios involve significantly different numbers of jobs, machines, and operations. Modern FJSS systems across different scenarios are expected to handle varying numbers of jobs, machines, and operations while efficiently responding to production conditions with scheduling solutions. Consequently, scalable FJSS models with strong cross-instance generalization and high efficiency have become increasingly important (Wang et al., 2025).

Recent advances in learning-based scheduling have opened a promising direction for FJSS by learning scheduling policies directly from historical instances instead of relying on manually designed dispatching rules. In particular, neural combinatorial optimization (NCO) and deep reinforcement learning (DRL) enable end-to-end policy learning and rapid response, in which deep neural networks automatically learn task representations from complex scheduling states (Chung et al., 2025). More

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

recently, Transformer architectures have further improved representation learning through global attention mechanisms, demonstrating promising scalability compared with conventional recurrent, convolutional, or graph architectures (Xiao et al., 2026).

However, existing learning-based methods still exhibit limited scalability in solving scheduling instances with diverse job-machine configurations. Moreover, learning-based methods, especially transformer-based ones, employ deep neural networks with homogeneous self-attention or cross-attention mechanisms. They might be insufficient for modeling the heterogeneous interactions among operations, machines, and the changing completion information during in-step inference, which are important for improving the scheduling quality. Recent studies have also explored large language models (LLMs) for scheduling and generating dispatching rules or heuristics. Despite their impressive reasoning capability, LLM-generated decisions are prone to hallucination and may violate scheduling constraints, limiting their trustworthiness and applicability to safety-critical manufacturing environments. Therefore, learning-based scheduling approaches still face three fundamental challenges: 1) scalability across variable-sized scheduling instances; 2) expressive modeling of heterogeneous interactions between operation-machine features and processing time information; 3) reliable decision generation under strict scheduling constraints.

To address these challenges, we propose a dual-attention proximal policy optimization (PPO) framework as a pre-training architecture for any-scale FJSS. We propose a dual-attention mechanism that aims to enhance representation for operations, machines, and the online operation-machine processing time as the state. In addition, the in-step scheduling information is considered by an online mask mechanism to reduce hallucination of the Transformer-based large model. The main contributions can be summarized as follows.

1. We develop AnyFJS, a Transformer-based neural scheduling framework for any-scale FJSS. By pre-training on a large and diverse distribution of instances with varying job-machine configurations, AnyFJS learns a transferable dispatching policy that generalizes to FJSS with previously unseen numbers of jobs and machines.
2. We propose a dual-attention mechanism that combines cross-attention and bilinear attention to jointly capture heterogeneous operation-machine interactions and high-order dependencies involving dynamically updated completion-time information.
3. We introduce an online mask that incorporates scheduling information during policy execution to filter infeasible actions, improving the reliability and efficiency of sequential dispatching decisions during training.

2 Related Work

We review the learning-based FJSS studies in NCO and DRL-assisted FJSS most closely related to our work, with a particular focus on state representation and policy optimization.

State Representation: Learning-based FJSS methods have evolved from handcrafted features to graph/attention-based representations. Song et al. (2023) introduce heterogeneous graph representation and GNN to capture complex operation-machine relationships. DANIEL (Wang et al., 2024) further employs dual attention to learn production-adaptive representations. Wang et al. (2025) propose a heterogeneous disjunctive graph and a memory-enhanced heterogeneous GNN to exploit historical trajectories. More recently, ReSched (Xiao et al., 2026) adopts Transformer blocks for scalable state representation. However, existing methods primarily focus on operation-machine relationships or general global dependencies, leaving the joint modeling of operations, machines, and dynamically updated completion-time information insufficiently explored.

Policy Optimization: DRL-based FJSS methods have explored various policy optimization paradigms, ranging from value-based learning (Du et al., 2024) to policy-gradient methods such as PPO (Li et al., 2025a). Meta-DRL has further been investigated to improve the adaptability of learned policies across changing scheduling environments (Wu et al., 2025). Beyond direct policy learning, L-RHO (Li et al., 2025b) employs a learned model to guide rolling-horizon optimization by identifying variables that need not be re-optimized, demonstrating the potential of learning to improve conventional optimization procedures. Nevertheless, these studies demonstrate the effectiveness of DRL in learning scheduling policies, but existing approaches mainly focus on improving

policy optimization or adaptation, with limited attention to scalable policy learning across diverse problem sizes and configurations.

In contrast to existing studies, our work integrates Transformer-based representation learning with PPO and online information feedback in a unified framework, targeting scalable policy learning and reliable decision-making across variable-sized FJSS problems.

3 METHODOLOGY

3.1 OVERALL WORKFLOW

The workflow of the proposed framework is illustrated in Fig. 6 in the appendix, and an overview is presented in Fig. 1. Given a flexible job shop scheduling problem (FJSP) instance, it iteratively constructs a complete schedule through the state-decision-execution loop. The overall scheduling process consists of four main steps.

First, it takes the FJSP information as input, including the jobs and their operations, the available machines, and the flexibility matrix that specifies the feasible machine assignments and corresponding processing times. Second, at each scheduling step t, the current scheduling status is transformed into an online state S_t. The state summarizes the dynamically changing information of the environment, such as machine availability, job readiness, completed operations, and the valid operation-machine (O-M) assignments. Third, the resulting state S_t is fed into the neural dispatcher, which determines the next dispatching action according to the learned scheduling policy by PPO. Finally, the selected action is executed in the FJSP environment, which updates the corresponding machine and job status, advances the scheduling process, and generates the next state S_{t+1} together with the associated reward. An example in Step 4 in Fig. 6 illustrates the changes in the scheduling state and the calculation of the reward after executing an action.

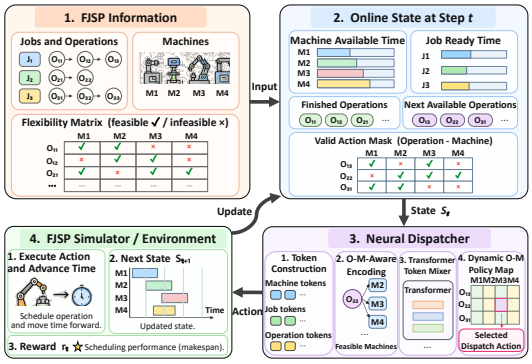

Figure 1: An overview of the workflow of the proposed AnyFJS framework.

Steps 2-4 form an iterative scheduling loop, where the updated state is continuously fed back to the neural dispatcher to obtain the next dispatching decision. This process is repeated until all operations are completed and a feasible schedule is obtained. The PPO-based training procedure used to learn the scheduling policy is described separately later. The same workflow is used for both training and testing. During training, the collected state-action-reward trajectories are used to update the scheduling policy through PPO, whereas during testing, the learned policy is applied directly without further parameter updates.

3.2 NEURAL DISPATCHER

The neural dispatcher maps the current online scheduling state to a dispatching action by jointly considering the interactions among operations and machines. As illustrated in Fig. 2, the dispatcher consists of four components: token construction, O-M-aware encoding, Transformer token mixing, and a dynamic O-M policy head.

Token Construction: The scheduling environment of an FJSP represents machines, jobs, and operations using corresponding tokens. Specifically, a job token is constructed for each job, an operation token is constructed for each operation, and a machine token is constructed for each machine. Let $\{ \mathbf{J} = \mathbf{j} * 1, \ldots, \mathbf{j} * N_J \}$, $\{ \mathbf{O} = \mathbf{o} * 1, \ldots, \mathbf{o} * N_O \}$, and $\{ \mathbf{M} = \mathbf{m} * 1, \ldots, \mathbf{m} * N_M \}$ denote the job, operation, and machine tokens, respectively. The operation tokens additionally encode the current scheduling status of the corresponding operations, allowing the dispatcher to distinguish completed operations from those that are required for dispatch. The machine tokens characterize

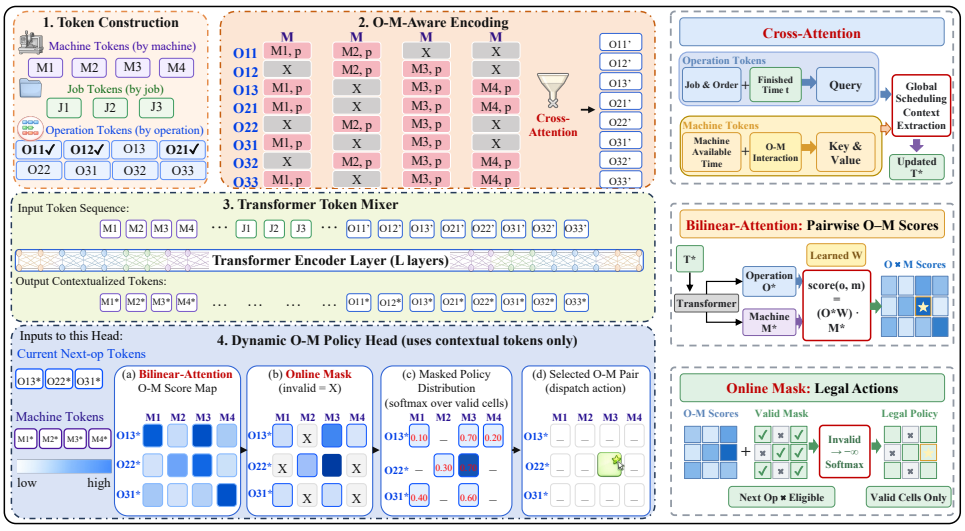

Figure 2: The proposed neural dispatcher workflow.

the current states of the machines, including their availability and completion time. These heterogeneous tokens provide a unified representation of the scheduling environment and serve as the inputs to the subsequent encoding module.

Operation-Machine-Aware Encoding: A key challenge in FJSP is that the scheduling decision is determined by the joint relationship between an operation and its feasible machines. To explicitly capture such relationships, we introduce an O-M-aware encoding based on a cross-attention mechanism. For each operation, its representation is used as the query, while the machine-related representations provide the keys and values. The machine-related representations incorporate both the machine states and the operation-specific processing-time information. Specifically, for an O-M pair (o_i, m_j), we construct its machine-related representation as

$$\mathbf{z} * ij = \phi(\mathbf{m} * j, p * ij, r_j^t),\ \mathbf{z} * ij \in \mathbf{Z}, \tag{1}$$

where $p * ij$ denotes the processing time of operation o_i on machine m_j, r_j^t denotes the available time of machine m_j at scheduling step t, and $\phi(\cdot)$ denotes the corresponding feature transformation. The query, key, and value representations are then obtained as $\mathbf{Q}_O = \mathbf{O}\mathbf{W}_O^Q$, $\mathbf{K}_Z = \mathbf{Z}\mathbf{W}_Z^K$, and $\mathbf{V}_Z = \mathbf{Z}\mathbf{W}_Z^V$, respectively, where $\mathbf{W}_O^Q$, $\mathbf{W}_Z^K$, and $\mathbf{W} * Z^V$ are learnable projection matrices. The cross-attention weights are computed as $\mathbf{A} * OZ = \mathrm{softmax}(\mathbf{Q}_O\mathbf{K} * Z^\top/\sqrt{d})$, and the updated operation representations are obtained as $\mathbf{O}' = \mathbf{O} + \mathbf{A} * OZ\mathbf{V}_Z$. Through this interaction, each operation selectively aggregates machine-side information according to the current scheduling state and the corresponding O-M processing characteristics. The resulting operation representations therefore incorporate both operation-level scheduling information and machine-dependent contextual information, yielding updated operation tokens denoted by $\mathbf{O}'$. This enables the dispatcher to explicitly learn interactions between operations and machines rather than modeling operations and machines independently. The detailed cross-attention mechanism is illustrated in the first part of the rightmost column in Fig. 2.

Transformer Token Mixer: The O-M-aware tokens are then fed into a multi-layer Transformer encoder to further capture global scheduling dependencies. Specifically, the updated operation tokens are combined with the machine and job tokens to form the input $\mathbf{T}^{(0)} = [\mathbf{M}; \mathbf{J}; \mathbf{O}']$. Through multiple Transformer layers, the contextualized token sequence is obtained as $\mathbf{T}^{(l+1)} = \mathrm{TransformerLayer}_l(\mathbf{T}^{(l)})$, $l = 0, \ldots, L-1$, yielding $\mathbf{T}^* = \mathrm{TransformerEncoder}(\mathbf{T}^{(0)})$. The resulting representations encode both local O-M interactions and global scheduling context. These contextualized tokens are subsequently used by the dynamic O-M policy head to generate the dispatching action.

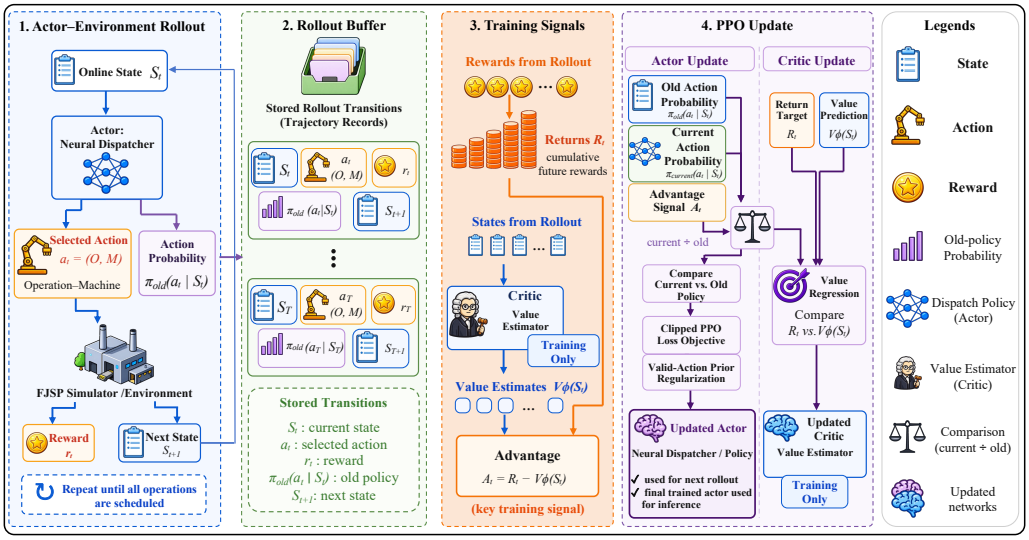

Figure 3: Illustration of the PPO training in the proposed framework.

Dynamic O-M Policy Head: The dynamic O-M policy head uses only the contextualized tokens to construct the dispatching policy. At each scheduling step, the head first selects the tokens corresponding to the currently available next operations and the machine tokens. Let $\mathbf{O}^*$ and $\mathbf{M}^*$ denote the corresponding contextualized operation and machine tokens. A bilinear-attention mechanism is then employed to compute pairwise O-M scores. Specifically, the complete O-M score map is obtained as

$$\mathbf{S}_t = \mathbf{O}_t^* \mathbf{W}_B (\mathbf{M}_t^*)^\top, \qquad s_{ij}^t = [\mathbf{S}_t]_{ij} = (\mathbf{o}_{i,t}^* \mathbf{W}_B) \cdot \mathbf{m}_{j,t}^*, \tag{2}$$

where $\mathbf{W} * B$ is a learnable bilinear projection matrix, and $s * ij^t$ measures the compatibility between operation o_i and machine m_j at scheduling step t. Applying this operation to all current next operations and machines produces an $O \times M$ score map, where each entry represents the preference for dispatching a particular operation to a particular machine.

However, not all O-M pairs constitute valid scheduling actions. To explicitly enforce the current feasibility constraints, an online mask $\mathbf{A} * t \in 0, 1^{O \times M}$ is constructed from the scheduling state at step t, where $A * ij^t = 1$ indicates that (o_i, m_j) is a valid dispatching action. The masked score and resulting policy distribution are defined as

$$\hat{s}_{ij}^t \begin{cases} s_{ij}^t, & A_{ij}^t = 1, \\ -\infty, & A_{ij}^t = 0, \end{cases} \qquad \pi_\theta(o_i, m_j \mid S_t) \frac{\exp(\hat{s} * ij^t)}{\sum *(i', j') \in \mathcal{A} * t \exp(\hat{s} * i'j'^t)}. \tag{3}$$

where the dynamic action space at scheduling step t is $\mathcal{A} * t = (o_i, m_j) \mid A * ij^t = 1$. Consequently, the policy distribution is normalized only over valid O-M pairs. This online masking mechanism not only guarantees feasible dispatching decisions but also reduces the effective action space during both training and inference, thereby improving computational efficiency. Finally, the dispatcher samples or selects an O-M pair from the masked policy distribution as the dispatching action.

3.3 Model Training

We train the proposed neural dispatcher using PPO. During each rollout, the actor interacts with the FJSP simulator by sequentially selecting an O-M pair $a_t = (O, M)$ according to the current policy $\pi_{\text{old}}(a_t|s_t)$. The simulator executes the selected dispatching action and returns the corresponding reward r_t, producing a trajectory until all operations are scheduled.

The collected rewards are first converted into return targets R_t, which are used to train the critic $V_\phi(s_t)$ through regression. The advantage is then computed as

$$A_t = R_t - V_\phi(s_t), \tag{4}$$

whcih provides the training signal for policy improvement. To obtain informative advantages, we align the rollout rewards with high-quality scheduling solutions generated by OR-Tools in the training data, such that the resulting advantage reflects the relative quality of the learned scheduling decisions with respect to strong reference solutions.

For the actor, the makespan of the completed schedule provides the primary optimization signal. Specifically, the makespan is converted into a reward, which is further combined with the reference solution quality to construct the advantage signal. A shorter makespan therefore, leads to a stronger positive training signal, encouraging the policy to increase the probability of dispatching decisions that contribute to high-quality schedules. The actor is then optimized using the clipped PPO objective, where the advantage determines the direction and magnitude of the policy update through the ratio between the current and old action probabilities. This alternating actor-critic procedure progressively improves the dispatching policy while maintaining stable policy updates.

4 EXPERIMENTS

We conduct extensive experiments on FJSPs from different scenarios with substantially different scales to demonstrate the competitiveness of AnyFJS against different types of peer methods. The brief introduction to the settings is as follows, while a detailed introduction is presented in the Appendix.

Training Data and Test Instances: AnyFJS is trained on a large collection of synthetic FJSP instances with a fixed number of jobs (10) and varying numbers of machines (4-20), operations (3-15), and flexibility levels (0.07-1), covering a broad range of problem configurations. We adopt 217 benchmark FJSP instances organized into 41 family-size groups from six benchmark families, ranging from 3×2 to 30×10 in size. The *Chambers–Barnes* set (Chambers & Barnes, 1996), the *Fattahi* set (Fattahi et al., 2007), and the *Hurink* variants (*edata*, *rdata*, *sdata*, and *vdata* (Hurink et al., 1994)).

Model Configuration Details: AnyFJS adopts a Transformer-based Actor-Critic architecture, with 8-layer Transformer encoders for both the Actor and Critic. The model uses 16-head self-attention with a hidden dimension of 1024, resulting in approximately 212M parameters in total. It is trained using PPO with BF16 mixed precision on 8 NVIDIA H20 GPUs, totaling nearly 482 GPU-hours.

Baselines: We compare AnyFJS against eight peer methods in three categories, and all baselines are evaluated with their original publicly released implementations or checkpoints, without retraining or fine-tuning on our test instances. *1) Priority dispatching rules (PDRs) (Lei et al., 2022).* FIFO, MOPNR, and MWKR. These priority rules originate from classical scheduling theory (Panwalkar & Iskander, 1977) and require no training. *2) NCO methods.* MPGN (Lei et al., 2022), HGNN (Song et al., 2023), and DANIEL (Wang et al., 2024). Since these models are originally trained on fixed small-scale instance distributions, evaluating them on our benchmark suite constitutes a stringent zero-shot generalization setting. *3) LLM-based schedulers.* SeEvo (Huang et al., 2026) and ReSched (Xiao et al., 2026).

In addition, for the extremely large-scale synthetic instances where most learned baselines become impractical to apply, we employ Google OR-Tools CP-SAT (Perron & Furnon, 2025) as the reference solver to obtain high-quality upper bounds (UBs) with a per-instance time limit of 1,800 s.

Evaluation Metrics: The primary objective is the makespan $C_{\max}$, defined as the completion time of the last operation, equivalently, the maximum completion time over all machines. For each benchmark instance i, we report the instance-level gap (%) to the best-known solution (BKS). Each family-size row averages the instance-level gaps of its N instances. For the large-scale synthetic instances, no published BKS exists; we therefore take the best feasible solution found by OR-Tools within the 1,800 s limit as the reference upper bound (UB) and report the gap. We additionally report the mean inference time per instance, measured end-to-end including model loading, decoding, and schedule reconstruction; processing times are normalized during inference and restored to the original integer scale when reporting makespans.

Table 1: Comparisons on the representative FJSP benchmark instances with different scales. Each row reports the instance-level gap (%) ↓ to the BKS, where N denotes the number of instances in this family-size group. The final row reports each method's mean runtime (s) ↓ of inference over all 41 family-size groups (217 instances). Every family-size row averages the instance-level gaps of all instances. Bold marks the gaps of the best method on each instance, while underlining denotes the strongest baseline in each row.

Family	Size	N	PDRs			NCO			LLM		**Ours**
			FIFO	MOPNR	MWKR	MPGN	HGNN	DANIEL	SeEvo	ReSched	AnyFJS
Chambers–Barnes	15×12	3	27.57	25.57	28.90	24.74	16.00	22.80	27.03	20.60	**8.88**
	15×16	2	25.08	23.30	26.33	22.84	15.68	17.87	26.50	18.34	**8.28**
	15×18	2	31.32	22.30	29.08	22.51	14.37	20.70	29.22	20.07	**9.93**
Hurink edata	10×5	5	20.80	26.46	25.71	18.87	14.66	18.93	31.92	19.55	**7.03**
	10×10	18	27.79	33.27	27.44	23.24	18.66	22.48	30.51	19.85	**11.91**
	15×5	5	17.28	15.80	20.80	12.36	13.36	13.00	18.51	9.83	**4.28**
Hurink sdata	10×5	5	20.06	21.54	21.28	16.68	14.55	21.35	22.65	12.42	**6.65**
	15×5	5	8.97	9.01	9.28	7.48	2.86	6.57	7.93	3.77	**1.05**
	15×15	5	26.01	27.62	21.32	21.16	19.21	24.26	21.19	19.80	**11.93**
Hurink vdata	10×10	18	8.22	14.00	9.36	5.13	5.49	7.12	265.90	16.24	**0.34**
	15×15	5	4.77	14.48	6.17	2.99	3.92	5.64	452.29	15.42	**0.21**
	30×10	5	3.11	5.32	5.36	3.30	2.73	6.42	327.30	3.65	**1.41**
Mean inference time (s)			0.06	0.09	0.11	4.84	0.69	0.64	0.01	7.80	15.35

4.1 Comparisons of AnyFJS and Baselines on Representative Scaled FJSPs

We evaluate AnyFJS against all baselines on 41 representative family-size groups covering diverse FJSP scales. Specifically, after large-scale training, AnyFJS is directly applied to the test instances. For the baselines, we use the implementations provided in the corresponding toolboxes for heuristic methods, while the original publicly released models from the authors are directly adopted for NCO and LLM-based methods without additional training.

Overall Performance. Table 1 presents the comparisons on several representative families with different scales for better comparisons and analyses. Among the baselines, HGNN is the strongest competitor on the largest number of groups, while ReSched and MPGN are competitive on several Hurink variants. However, AnyFJS consistently outperforms all baselines across diverse benchmark families, demonstrating robust solution quality and model generalization ability.

Zero-shot Generalization. The evaluated instances cover substantially different numbers of jobs and machines. Importantly, AnyFJS maintains strong performance across these scales without scale-specific redesign or retraining, indicating effective zero-shot generalization to problem sizes. For example, it achieves gaps of 0.34% and 1.41% on the 10×10 and 30×10 Hurink vdata groups, respectively, both substantially below all other baselines. This suggests that AnyFJS learns transferable scheduling strategies rather than relying on fixed-size patterns.

Inference Efficiency. AnyFJS requires an average inference time of 15.35 seconds over 217 instances. Although this is slightly higher than peer methods, especially the lightweight dispatching

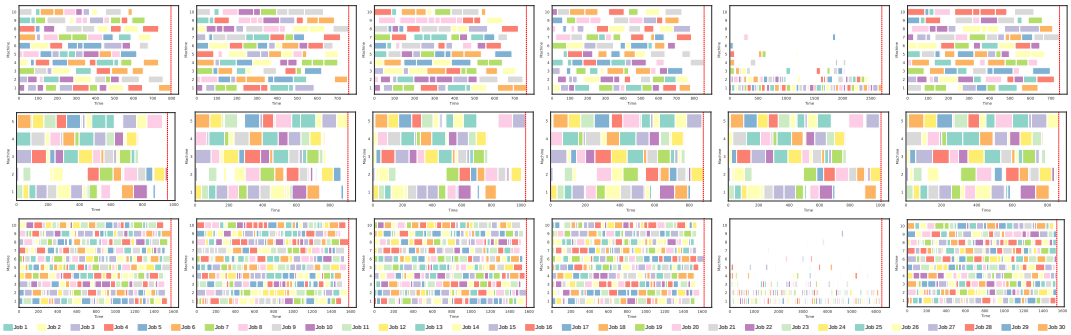

Figure 4: The Gantt charts of AnyFJS and peer methods on three representative instances, where the legends are listed at the bottom.

Table 2: Comparisons of AnyFJS and the best-performing baseline on all 41 family-size groups. Each row is an average over N instances of the family-size group. Bold values indicate the lower gap between the best baseline and AnyFJS. Time indicates the inference time of AnyFJS.

Family	Size	N	Best baseline	Base gap	AnyFJS gap	Time (s)
Chambers–Barnes	10×11	2	HGNN	13.76	**13.71**	12.32
	15×11	2	HGNN	14.47	**10.77**	16.94
	15×12	3	HGNN	16.00	**8.88**	16.81
	15×13	2	HGNN	15.14	**10.03**	17.53
	15×16	2	HGNN	15.68	**8.28**	24.49
	15×17	3	HGNN	12.65	**10.37**	25.14
	15×18	2	HGNN	14.37	**9.93**	25.61
Fattahi et al.	3×2	3	MPGN	1.12	**0.00**	9.41
	3×3	2	FIFO+EET	8.63	**2.38**	8.31
	5×6	1	ReSched	6.83	**5.77**	8.37
	6×7	1	ReSched	18.88	**3.86**	8.59
	7×7	2	ReSched	16.52	**10.58**	8.50
	12×8	1	ReSched	20.73	**19.73**	10.06
Hurink edata	10×5	5	HGNN	14.66	**7.03**	9.85
	10×10	18	HGNN	18.66	**11.91**	12.19
	15×5	5	ReSched	9.83	**4.28**	11.83
	15×10	5	HGNN	17.43	**14.13**	15.83
	15×15	5	HGNN	16.46	**12.41**	24.59
	20×5	6	MPGN	15.10	**12.90**	14.13
	20×10	5	HGNN	22.88	**18.00**	20.75
Hurink rdata	6×6	1	MPGN	**6.38**	**6.38**	9.35
	10×5	5	HGNN	10.80	**4.35**	9.61
	10×10	18	HGNN	18.57	**11.03**	12.42
	15×5	5	HGNN	4.60	**2.25**	10.79
	15×10	5	HGNN	20.42	**16.56**	15.91
	15×15	5	HGNN	18.86	**17.20**	24.95
	20×5	6	MPGN	6.89	**4.37**	12.87
Hurink sdata	10×5	5	ReSched	12.42	**6.65**	9.64
	10×10	18	HGNN	19.52	**10.94**	12.09
	15×5	5	HGNN	2.86	**1.05**	10.52
	15×10	5	HGNN	15.20	**10.85**	15.83
	15×15	5	HGNN	19.21	**11.93**	24.03
	20×10	5	HGNN	16.11	**13.86**	20.88
Hurink vdata	10×5	5	HGNN	6.97	**3.00**	9.46
	10×10	18	MPGN	5.13	**0.34**	12.22
	15×5	5	HGNN	4.02	**1.71**	10.76
	15×10	5	HGNN	5.86	**3.30**	15.64
	15×15	5	MPGN	2.99	**0.21**	24.10
	20×5	6	MPGN	3.07	**1.39**	12.39
	20×10	5	HGNN	4.03	**1.36**	20.55
	30×10	5	HGNN	2.73	**1.41**	33.99

rules, the overhead remains moderate compared with its substantial improvement in solution quality. The results therefore suggest that the additional inference cost of AnyFJS is acceptable for obtaining consistently high-quality results across different problem types and scales.

Visualization of the Gantt Charts. The Gantt charts in Fig. 4 further illustrate the scheduling behaviors of different methods. AnyFJS produces high-quality schedules. In particular, the operations are organized in a more structured manner across machines, with reduced unnecessary fragmentation and improved utilization of available processing time. The resulting schedules also demonstrate a clear coordination between operation sequencing and machine allocation, which helps avoid excessive idle periods and alleviates bottlenecks on critical machines. Such advantages become increasingly clear as the problem scale grows.

4.2 Performance of AnyFJS on Different Scaled FJSP

The proposed AnyFJS and all peer methods are also evaluated on all 41 family-size groups, covering six benchmark families and problem sizes ranging from 3×2 to 30×10. Compared with the training distribution, these test instances span a substantially broader range of problem scales and exhibit more diverse and complex combinations of jobs, machines, and routing structures, providing a challenging evaluation under realistic scheduling scenarios. Across all 41 groups, AnyFJS achieves a mean gap of 7.93%, a median gap of 8.28%, and is always the best, outperforming the strongest baseline on 40 groups and tying on the remaining group. The inference time remains efficient across different scales, with only a moderate increase as the instance size grows. Among the baselines, HGNN is the strongest overall, achieving the lowest gap on the largest number of groups, followed by ReSched and MPGN. These results demonstrate strong cross-family and cross-scale generalization, indicating that AnyFJS learns transferable scheduling strategies rather than relying on family- or size-specific patterns.

Table 3: Results of AnyFJS and OR-Tools on 128 synthetic 100×20 instances grouped by different number of eligible machines k per operation.

k	N	Mean makespan		Gap		Mean time
		OR-Tools UB	**AnyFJS**	Mean (%)	Std. (pp)	(s)
1–5	33	5139.55	5375.45	4.5816	1.2036	161.39
6–10	32	4973.56	5172.91	4.0083	0.8243	166.69
11–15	39	4980.62	5201.18	4.4297	1.0003	161.93
16–20	24	4937.67	5155.50	4.4152	0.7983	162.26

4.3 Extending to Extremly Large-scaled FJSP

We further evaluate AnyFJS on 128 synthetic 100×20 FJSP instances, where each instance contains 2,000 operations and represents a substantially larger scale than the benchmark instances considered

above. Notably, there is a substantial scale gap between the training distribution and these test instances: AnyFJS is trained on relatively small-scale FJSPs, while the test instances are considerably larger in terms of problem size and contain 2,000 operations each. This setting therefore provides a stringent evaluation of cross-scale generalization, as the model is directly applied to substantially larger problems without retraining or fine-tuning. At this scale, existing NCO and LLM-based methods either become impractical to apply or exhibit substantially degraded solution quality. We therefore compare AnyFJS with OR-Tools, which is given sufficient solving time to obtain a high-quality upper bound (UB).

As shown in Table 3, AnyFJS consistently achieves low gaps across different ranges of eligible machines k, with mean gaps between 4.01% and 4.58%. The overall mean gap is 4.36%, while the inference time remains around 161–163 seconds across all groups. These results demonstrate that AnyFJS can effectively generalize from the small-scale training distribution to substantially larger FJSPs, highlighting the scalability and cross-scale generalization capability of the proposed framework. The detailed experimental results are reported in the Appendix.

Table 4: Results of makespan and run time on a synthetic 100×100 FJSP instance.

Method	Makespan	Time	Method	Makespan	Time
FIFO-EET	5088	67.87	MWKR-EET	5139	72.54
MPGN	–	–	HGNN	5042	4159.09
DANIEL	4975	1855.37	SeEvo	22204	5.30
ReSched	4904	2417.02	**AnyFJS**	**4652**	**2257.14**

We further evaluate different methods on an extremely large-scale 100×100 FJSP instance. As shown in Table 4, AnyFJS achieves the best makespan of 4652, outperforming all competing methods, while maintaining a reasonable inference time of 2257.14 s. These results demonstrate the effectiveness of the proposed model in handling large-scale FJSP instances.

4.4 Convergence of the Training Loss of Actor

Figure 5 illustrates the training convergence of AnyFJS under the sampling-based and greedy-based action selection strategies. Both strategies exhibit a clear decreasing trend in normalized makespan as the training epochs increase, indicating that the learned scheduling policy progressively improves and converges during training. Compared with random sampling, the greedy strategy converges more rapidly and reaches a lower final normalized makespan, demonstrating more effective exploitation of the learned policy and higher scheduling accuracy. These results confirm the convergence of the proposed training procedure and the effectiveness of greedy action selection for obtaining high-quality schedules.

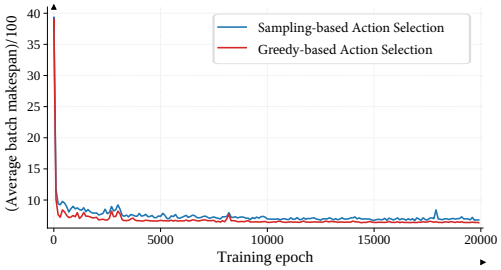

Figure 5: The training loss curves using a random sample and a greedy strategy.

5 Conclusions

This paper presents a novel neural dispatch framework that enables training a single model to generalize across FJSPs of different scales. The proposed dual-attention PPO achieves substantially lower makespan and competitive computational efficiency compared with a diverse set of heuristic, NCO, and LLM-based methods on 217 benchmark instances and 129 extremely large-scale synthetic instances, covering problem sizes from 3 jobs $\times$ 2 machines to 100 jobs $\times$ 100 machines. These results demonstrate the potential of the proposed framework for solving complex FJSPs and its broader applicability to real-world scheduling problems. More importantly, the proposed framework provides a promising foundation for training generalizable neural solvers for other job shop scheduling and related scheduling problems. In the future, we will explore integrating the proposed model into the agentic AI systems as a specialized scheduling skill, with the goal of enhancing the problem-solving capabilities of LLMs. The source code and model for peer review can be obtained from ***https://github.com/Anonymous-cloud-paper/ICLR27-AnyFJS/releases/tag/V1.0***

AI USAGE STATEMENTS

In this work, we used generative AI tools to assist with writing and presentation. Specifically, these tools were used to improve grammatical correctness, sentence fluency, clarity, and consistency of academic expression, as well as to polish the wording of selected passages without altering their intended technical meaning. Generative AI tools were also used to assist with the formatting and organization of large tables, including adjusting table layouts, aligning entries, refining column structures, and improving their readability and presentation. All AI-assisted content and formatting were subsequently reviewed, verified, and revised by the authors, who take full responsibility for the final manuscript.

REFERENCES

Richard L. Chambers and J. Wesley Barnes. New results in flexible job-shop scheduling. Technical Report ORP96-10, Graduate Program in Operations Research and Industrial Engineering, The University of Texas at Austin, 1996.

Kwok Tung Chung, Carman KM Lee, and Yung Po Tsang. Neural combinatorial optimization with reinforcement learning in industrial engineering: a survey. *Artificial Intelligence Review*, 58(5): 130, 2025.

Stéphane Dauzère-Pérès, Junwen Ding, Liji Shen, and Karim Tamssaouet. The flexible job shop scheduling problem: A review. *European Journal of Operational Research*, 314(2):409–432, 2024.

Yu Du, Junqing Li, Chengdong Li, and Peiyong Duan. A reinforcement learning approach for flexible job shop scheduling problem with crane transportation and setup times. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4):5695–5709, 2024.

Parviz Fattahi, Mohammad Saidi Mehrabad, and Fariborz Jolai. Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. *Journal of Intelligent Manufacturing*, 18 (3):331–342, 2007. doi: 10.1007/s10845-007-0026-8.

Jin Huang, Qihao Liu, Xinyu Li, Liang Gao, and Yue Teng. Automatic programming via large language models with population self-evolution for dynamic fuzzy job shop scheduling problem. *IEEE Transactions on Fuzzy Systems*, 34(3):896–908, 2026. doi: 10.1109/TFUZZ.2025.3650586.

Johann Hurink, Bernd Jurisch, and Monika Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spektrum*, 15(4):205–215, 1994. doi: 10.1007/BF01719451.

Kun Lei, Peng Guo, Wenchao Zhao, Yi Wang, Linmao Qian, Xiangyin Meng, and Liansheng Tang. A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem. *Expert Systems with Applications*, 205:117796, 2022. doi: 10.1016/j.eswa.2022.117796.

Chen Li, Xiyan Zhao, Lin Lin, Wenqiang Zhang, Mitsuo Gen, and Qing Zhang. An evolutionary knowledge training-based proximal policy optimization algorithm for job shop scheduling in flexible intelligent manufacturing. *Computers & Industrial Engineering*, 210:111533, 2025a.

Sirui Li, Wenbin Ouyang, Yining Ma, and Cathy Wu. Learning-guided rolling horizon optimization for long-horizon flexible job-shop scheduling. In *International Conference on Learning Representations*, 2025b.

Mingju Liu, Yingjie Li, Jiaqi Yin, Zhiru Zhang, and Cunxi Yu. Differentiable combinatorial scheduling at scale. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 31464–31476. PMLR, 2024.

S. S. Panwalkar and Wafik Iskander. A survey of scheduling rules. *Operations Research*, 25(1): 45–61, 1977. doi: 10.1287/opre.25.1.45.

Laurent Perron and Vincent Furnon. Or-tools: Google's operations research suite. `https://github.com/google/or-tools`, 2025.

Wen Song, Xinyang Chen, Qiqiang Li, and Zhiguang Cao. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 19(2):1600–1610, 2023.

Zaixing Sun, Fangfang Zhang, Yi Mei, Hejiao Huang, Chonglin Gu, Bin Wang, and Mengjie Zhang. Cooperative coevolution genetic programming for dynamic joint workflow scheduling and container scaling in cloud-fog computing. *IEEE Transactions on Services Computing*, 19(1):225–239, 2026.

Jiaqi Wang, Zhiguang Cao, Peng Zhao, Rui Cao, Yubin Xiao, Yuan Jiang, and You Zhou. Learning memory-enhanced improvement heuristics for flexible job shop scheduling. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Runqing Wang, Gang Wang, Jian Sun, Fang Deng, and Jie Chen. Flexible job shop scheduling via dual attention network-based reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, 35(3):3091–3102, 2024.

Lincong Wu, Xiaoxia Li, Xin Lu, Zaiwen Feng, and Yanguo Jing. An adaptive meta-reinforcement learning framework for dynamic flexible job shop scheduling. *IEEE Transactions on Automation Science and Engineering*, 22:24036–24052, 2025.

Xiangjie Xiao, Zhiguang Cao, Cong Zhang, and Wen Song. RESCHED: Rethinking flexible job shop scheduling from a transformer-based architecture with simplified states. In *The Fourteenth International Conference on Learning Representations*, 2026.

Cong Zhang, Zhiguang Cao, Wen Song, Yaoxin Wu, and Jie Zhang. Deep reinforcement learning guided improvement heuristic for job shop scheduling. In *International Conference on Learning Representations*, 2024.

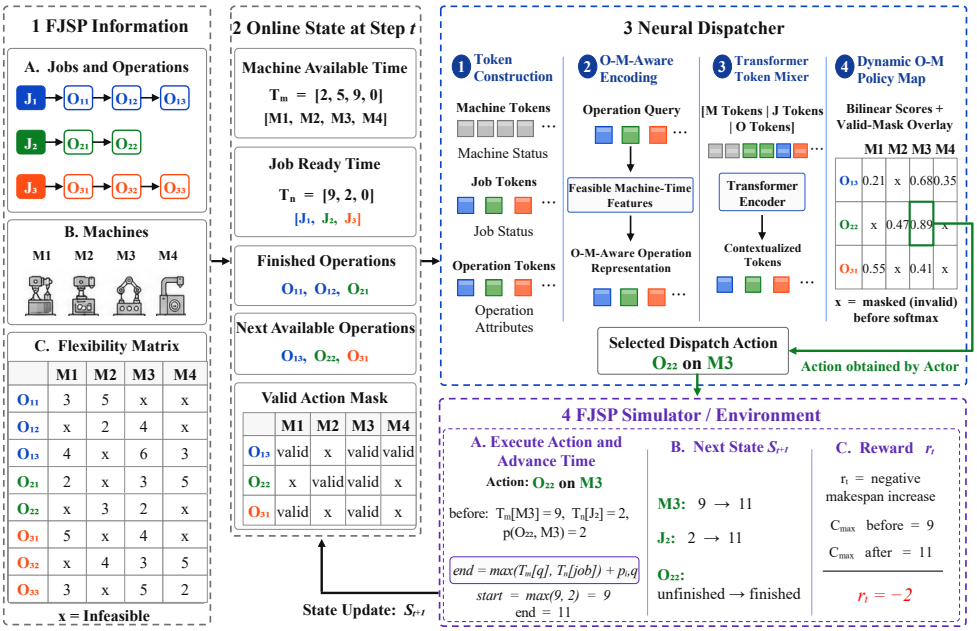

Figure 6: The overall workflow of the proposed framework.

A Appendix

A.1 Detailed Explanations of our Proposed Framework

This section provides a detailed description of the complete scheduling workflow illustrated in Fig. 6. The workflow consists of four main stages: FJSP information, online state construction, neural dispatching, and FJSP simulation. At each scheduling step, the current scheduling state is first constructed from the partially completed schedule. The neural dispatcher then evaluates the feasible operation–machine (O-M) assignments and selects a dispatching action. The selected action is subsequently executed by the FJSP simulator, which updates the machine, job, and operation states and calculates the corresponding reward. The updated state is then fed back to the neural dispatcher, forming an iterative state–decision–execution loop.

A.1.1 FJSP Information

The first stage provides the basic information of an FJSP instance, including the jobs and their operations, the available machines, and the flexibility matrix. As illustrated in Fig. 6, the example instance contains three jobs and four machines. Specifically, the three jobs are

$$J_1 = \{O_{11}, O_{12}, O_{13}\}, \quad J_2 = \{O_{21}, O_{22}\}, \quad J_3 = \{O_{31}, O_{32}, O_{33}\}, \tag{5}$$

and the available machines are

$$\mathcal{M} = \{M_1, M_2, M_3, M_4\}. \tag{6}$$

Each operation can be processed by one or more eligible machines. The corresponding processing times are specified by the flexibility matrix shown in Fig. 6. For an operation–machine pair (o_i, m_j), let p_{ij} denote its processing time. An infeasible assignment is denoted by x in the flexibility matrix. The feasibility of an O-M assignment can therefore be represented by

$$a_{ij} = \begin{cases} 1, & \text{if } o_i \text{ can be processed on } m_j, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

For example, O_{22} can be processed on M_2 and M_3, with processing times of 3 and 2, respectively, while its assignment to M_1 and M_4 is infeasible. This flexibility information provides the basic constraints for subsequent scheduling decisions.

A.1.2 Online State at Step t

During scheduling, the complete FJSP instance is progressively transformed into a partial schedule. At each scheduling step t, the simulator maintains the current online scheduling state. The state contains the dynamically changing machine availability, job readiness, operation completion status, and feasible O-M assignments.

In the example shown in Fig. 6, the machine available times at scheduling step t are

$$\mathbf{T}_m^t = [2, 5, 9, 0], \tag{8}$$

where the four entries correspond to M_1, M_2, M_3, and M_4, respectively. For example, $T_m^t[M_3] = 9$ means that machine M_3 is available for a newly assigned operation from time 9 onward.

Similarly, the current job ready times are

$$\mathbf{T}_n^t = [9, 2, 0], \tag{9}$$

where the entries correspond to J_1, J_2, and J_3. A job ready time indicates the earliest time at which its next unscheduled operation can start, given the completion of its preceding operations.

At this scheduling step, the finished operations are

$$\mathcal{O}_t^{\text{finished}} = \{O_{11}, O_{12}, O_{21}\}, \tag{10}$$

while the next available operations are

$$\mathcal{O}_t^{\text{next}} = \{O_{13}, O_{22}, O_{31}\}. \tag{11}$$

Only these next available operations can participate in the current dispatching decision. Combining the operation readiness with the flexibility matrix produces the valid O-M action mask. In particular, the valid assignments at step t are

$$\mathcal{A}_t = \left\{(o_i, m_j) \mid o_i \in \mathcal{O}_t^{\text{next}},\ a_{ij} = 1\right\}. \tag{12}$$

For the example in Fig. 6, this produces the following valid assignments: O_{13} can be assigned to M_1, M_3, or M_4; O_{22} can be assigned to M_2 or M_3; and O_{31} can be assigned to M_1 or M_3. All other O-M combinations are masked as invalid.

Therefore, the online state at step t can be conceptually summarized as

$$S_t = \left(\mathbf{T}_m^t, \mathbf{T}_n^t, \mathcal{O}_t^{\text{finished}}, \mathcal{O}_t^{\text{next}}, \mathcal{A}_t\right). \tag{13}$$

This state representation changes after every dispatching action and provides the complete scheduling context required by the neural dispatcher.

A.1.3 Neural Dispatcher

Given the online state S_t, the neural dispatcher determines the next O-M dispatching action. As shown in Fig. 6, the dispatcher consists of four components: token construction, O-M-aware encoding, Transformer token mixing, and the dynamic O-M policy head.

First, the scheduling state is converted into machine tokens, job tokens, and operation tokens. The machine tokens encode machine status, the job tokens encode job status, and the operation tokens

encode operation attributes. These heterogeneous tokens provide a unified representation of the current scheduling environment.

Second, the operation tokens interact with feasible machine–time information through the O-M-aware encoding module. For each operation–machine pair, the corresponding machine representation incorporates the machine state and the processing-time information. Cross-attention is then used to learn the interaction between operation queries and machine-related features, producing O-M-aware operation representations.

Third, the O-M-aware operation representations are combined with the machine and job tokens and processed by the Transformer token mixer. Multiple Transformer encoder layers are used to extract the global scheduling context and generate contextualized tokens. These representations jointly capture the relationships among operations, jobs, machines, and their current scheduling status.

Finally, the dynamic O-M policy head evaluates the currently valid operation–machine combinations. Bilinear attention is first used to calculate an O-M score map. The online valid-action mask is then applied to remove infeasible O-M pairs from the policy distribution. For the example in Fig. 6, the resulting scores before softmax are

$$\mathbf{S}_t = \begin{bmatrix} 0.21 & -\infty & 0.68 & 0.35 \\ -\infty & 0.47 & 0.89 & -\infty \\ 0.55 & -\infty & 0.41 & -\infty \end{bmatrix}, \tag{14}$$

where the rows correspond to O_{13}, O_{22}, and O_{31}, respectively, and the columns correspond to M_1–M_4. The entries marked by $-\infty$ correspond to infeasible O-M assignments and therefore receive zero probability after softmax.

The policy distribution is consequently defined only over the valid action set $\mathcal{A}_t$,

$$\pi_\theta(o_i, m_j \mid S_t) = \frac{\exp(\hat{s}^t_{ij})}{\sum_{(i',j')\in\mathcal{A}_t} \exp(\hat{s}^t_{i'j'})}. \tag{15}$$

In the illustrated example, the largest score among the valid O-M pairs is 0.89, corresponding to the assignment of O_{22} to M_3. Thus, under the greedy action-selection strategy, the dispatcher selects

$$a_t = (O_{22}, M_3). \tag{16}$$

This selected O-M pair is then passed to the FJSP simulator for execution.

A.1.4 FJSP Simulator / Environment

The FJSP simulator is responsible for executing the selected dispatching action, advancing the scheduling state, and calculating the reward. It acts as the interaction environment of the neural dispatcher. After receiving an action from the actor, the simulator updates the corresponding machine and job availability times, changes the status of the dispatched operation, and produces the next online state.

A. Execute Action and Advance Time Consider the action selected in the example above, namely assigning operation O_{22} to machine M_3. From the current online state, the available time of M_3 is 9, while the ready time of job J_2 is 2. The processing time of O_{22} on M_3 is 2. Therefore, the operation cannot start before both the selected machine and its corresponding job are ready.

The start time is determined by taking the later of the machine available time and the job ready time:

$$s^t_{22} = \max\left(T^t_m[M_3], T^t_n[J_2]\right). \tag{17}$$

Substituting the values from the current state gives

$$s_{22}^t = \max(9, 2) = 9. \tag{18}$$

Therefore, although job J_2 is already ready at time 2, the operation has to wait until machine M_3 becomes available at time 9. Since the processing time of O_{22} on M_3 is 2, its completion time is

$$c_{22}^t = s_{22}^t + p_{22,3} = 9 + 2 = 11. \tag{19}$$

After executing the operation, the availability time of the selected machine is updated from 9 to 11. Thus,

$$T_m^{t+1}[M_3] = c_{22}^t = 11. \tag{20}$$

At the same time, the progress of job J_2 is updated. Since O_{22} is the final operation of J_2 in this example, its completion changes the ready time of J_2 from 2 to 11:

$$T_n^{t+1}[J_2] = 11. \tag{21}$$

The operation status is also updated from unfinished to finished. Consequently, O_{22} is removed from the set of next available operations. The simulator then determines whether the completion of O_{22} makes any subsequent operation newly available. In this particular example, no subsequent operation exists for J_2, so the set of candidate operations is updated accordingly.

This example illustrates that one dispatching action simultaneously affects several components of the scheduling environment. The selected machine becomes occupied until the operation finishes, the corresponding job advances to its new state, and the operation itself becomes completed. These changes jointly determine the next scheduling state.

B. Next State S_{t+1} After executing $a_t = (O_{22}, M_3)$, all affected scheduling information is updated to form the next state. In particular, the machine available-time vector changes from

$$[2, 5, 9, 0]$$

to a vector whose M_3 entry is 11, while the job ready-time vector changes from

$$[9, 2, 0]$$

to a vector whose J_2 entry is 11. The finished-operation set additionally contains O_{22}, and the set of next available operations is updated by removing O_{22} and adding any newly released operations if their precedence constraints have been satisfied.

The state transition can therefore be represented as

$$S_{t+1} = \mathcal{T}(S_t, a_t), \tag{22}$$

where $\mathcal{T}(\cdot)$ denotes the transition function of the FJSP simulator. The updated state S_{t+1} is then returned to the neural dispatcher and becomes the input for the next scheduling decision.

C. Reward r_t The simulator also calculates a reward after executing each action. In the example shown in Fig. 6, the reward is defined according to the increase in the current makespan. Before executing O_{22} on M_3, the current makespan is 9, corresponding to the latest machine availability time. After the operation is completed at time 11, the current makespan becomes 11. Therefore, the makespan increase caused by this action is

$$\Delta C_{\max}^t = C_{\max}^{t+1} - C_{\max}^t = 11 - 9 = 2. \tag{23}$$

The corresponding reward is the negative makespan increase:

$$r_t = -\Delta C^t_{\max} = -2. \tag{24}$$

Thus, the selected action receives a negative reward of -2, indicating that the execution of this action increases the current makespan by two time units. This reward provides immediate feedback to the PPO agent and encourages the policy to favor dispatching decisions that lead to smaller increases in the final makespan.

Overall, the interaction at scheduling step t can be summarized as

$$S_t \xrightarrow{\pi_\theta} a_t \xrightarrow{\mathcal{T}} (S_{t+1}, r_t). \tag{25}$$

The updated state is subsequently returned to the neural dispatcher, which generates the next O-M dispatching action. This process is repeated until all operations are completed and the final schedule is obtained.

A.1.5 Complete Scheduling Loop

Combining the four stages above, the complete scheduling process can be summarized as follows. The FJSP instance first provides the static problem information, including jobs, operations, machines, and O-M processing characteristics. At each scheduling step, the simulator converts the current partial schedule into an online state containing machine availability, job readiness, operation status, and valid O-M assignments. The neural dispatcher then processes this state and produces a policy over the dynamically changing action space. After an O-M pair is selected, the simulator executes the operation, updates the scheduling status, and calculates the corresponding reward. The resulting state is fed back into the dispatcher to continue the scheduling process.

This iterative procedure can be expressed as

$$S_t \to \pi_\theta(a_t \mid S_t) \to a_t \to (S_{t+1}, r_t), \qquad t = 0, \ldots, T-1. \tag{26}$$

The process terminates when all operations in the instance have been completed. During training, the resulting state–action–reward trajectories are collected and used by PPO to update the parameters of the neural dispatcher. During testing, the learned policy is directly applied to unseen FJSP instances without further parameter updates.

A.2 Detailed Introduction of Experimental Settings

Training Data and Test Instances: AnyFJS is trained on a large collection of synthetic FJSP instances with a fixed number of jobs (10) and varying numbers of machines (4-20), operations (3-15), and flexibility levels (0.07-1), covering a broad range of problem configurations.

For out-of-sample evaluation, we adopt 217 benchmark instances organized into 41 family-size groups from six benchmark families, ranging from 3×2 to 30×10 in size. The *Chambers–Barnes* set (Chambers & Barnes, 1996) contributes 16 instances in seven groups (10×11 to 15×18). The *Fattahi* set (Fattahi et al., 2007) contributes ten instances in six groups (3×2 to 12×8), covering small- and medium-sized instances with low machine flexibility. The remaining 28 groups (191 instances) are provided by the four Hurink variants, *edata*, *rdata*, *sdata*, and *vdata* (Hurink et al., 1994), which extend classical job-shop instances with multi-purpose machines under different average flexibility levels (from 1.0 to 2.5 eligible machines per operation) and processing-time generation schemes, spanning sizes from 6×6 to 30×10. Together, these families cover diverse routing flexibilities and problem scales, providing a comprehensive testbed for cross-scale generalization.

Baselines: We compare AnyFJS against eight peer methods in three categories, and all baselines are evaluated with their original publicly released implementations or checkpoints, without retraining or fine-tuning on our test instances.

1) Priority dispatching rules (PDRs). Following the compound-rule construction of the multi-action DRL framework (Lei et al., 2022), we adopt three composite rules that pair the job-priority rules FIFO, MOPNR (Most Operations Number Remaining), and MWKR (Most Work Remaining) with

the machine-assignment rule EET (Earliest End Time), abbreviated as FIFO, MOPNR, and MWKR in the tables. These priority rules originate from classical scheduling theory (Panwalkar & Iskander, 1977) and require no training.

2) NCO methods. **MPGN** (Lei et al., 2022) denotes the multi-action deep reinforcement learning framework, which encodes the disjunctive graph with a GIN network and jointly decodes operation and machine actions through dual pointer networks trained by multi-actor PPO. **HGNN** (Song et al., 2023) learns a heterogeneous graph representation of the FJSP and optimizes the scheduling policy with PPO. **DANIEL** (Wang et al., 2024) employs a dual attention network to capture operation–machine interactions and generates schedules with an end-to-end learned policy. Since these models are originally trained on fixed small-scale instance distributions, evaluating them on our benchmark suite constitutes a stringent zero-shot generalization setting.

3) LLM-based and recent learned schedulers. **SeEvo** (Huang et al., 2026) automatically designs dispatching heuristics via LLM-driven population self-evolution, and we directly apply the released evolved heuristic to all instances. **ReSched** (Xiao et al., 2026) is the latest Transformer-based scheduler with simplified states, included as a representative of the most recent scalable learned dispatchers.

In addition, for the extremely large-scale synthetic instances where most learned baselines become impractical to apply, we employ Google OR-Tools CP-SAT (Perron & Furnon, 2025) as the reference solver to obtain high-quality upper bounds (UBs) with a per-instance time limit of 1,800 s.

Evaluation Metrics: The primary objective is the makespan $C_{\max}$, defined as the completion time of the last operation, equivalently the maximum completion time over all machines. For each benchmark instance i, we report the instance-level gap (%) to the best-known solution (BKS):

$$\mathrm{Gap}_i(\%) = 100 \times \frac{C_{\max,i} - C_i^{\mathrm{ref}}}{C_i^{\mathrm{ref}}},$$

where C_i^{ref} is the reference value shipped with the benchmark files, corresponding to proven optima for the Chambers–Barnes family and to the published lower bounds for the Hurink families. Each family-size row in Table 1 averages the instance-level gaps of its N instances, rather than forming a ratio from separately averaged makespans. For the large-scale synthetic instances in Section 4.3, no published BKS exists; we therefore take the best feasible solution found by OR-Tools within the 1,800 s limit as the reference upper bound (UB) and report the gap $\mathrm{Gap}_{\mathrm{UB}}(\%) = 100 \times (C_{\max}^{\mathrm{AnyFJS}} - C_{\max}^{\mathrm{OR}})/C_{\max}^{\mathrm{OR}}$. We additionally report the mean inference time per instance, measured end-to-end including model loading, decoding, and schedule reconstruction; processing times are normalized during inference and restored to the original integer scale when reporting makespans.

A.3 Detailed Training and Test Configurations

Table 5 summarizes the model architecture, training hyperparameters, training instance generation settings, and computational configurations of AnyFJS.

Model Architecture. AnyFJS adopts a Transformer-based Actor-Critic architecture. Both the actor and critic employ an 8-layer Transformer encoder. The hidden dimension is set to 1024, with 16 attention heads in each Transformer layer. The MLP expansion ratio is set to $4\times$. The actor and critic contain 106,539,009 and 105,491,457 trainable parameters, respectively, resulting in 212,030,466 parameters in total.

Training Configuration. The model is implemented using PyTorch, with HuggingFace Accelerate for distributed training. Eight distributed processes are used, corresponding to 8 NVIDIA H20 GPUs with 96 GB memory per device. BF16 mixed-precision training is adopted to improve computational and memory efficiency. The model is trained for 20,000 epochs. The learning rates of both the actor and critic are set to 8×10^{-5}, while the PPO coefficient β is set to 0.005.

Training Instance Generation. During the training process, the number of jobs n is fixed at 10. The number of machines m varies from 4 to 20, and the number of operations per job h varies from 3 to 15. The flexibility ratio $flex$ ranges from 0.07 to 1.00, covering training instances with different levels of machine flexibility. Importantly, these training instances are all of relatively low dimensionalities to facilitate the model training. However, the test instances include higher dimensionalities to verify the generalization capability of our proposed framework.

Table 5: Model architecture, training hyperparameters, instance generation settings, and computational configurations of AnyFJS.

Configuration	Setting
Model Architecture	
Model architecture	Transformer-based Actor–Critic
Actor backbone	8-layer Transformer Encoder
Critic backbone	8-layer Transformer Encoder
Hidden dimension	1024
Number of attention heads	16
MLP expansion ratio	4×
Actor parameters	106,539,009
Critic parameters	105,491,457
Total parameters	212,030,466
Training Configuration	
Training framework	PyTorch + HuggingFace Accelerate
Number of distributed processes	8
Mixed-precision training	BF16
GPU configuration	8 × NVIDIA H20
GPU memory per device	96 GB
Number of training epochs	20,000
Actor learning rate	8×10^{-5}
Critic learning rate	8×10^{-5}
PPO β	0.005
Training Instance Generation	
Number of jobs (n)	10
Number of machines (m)	4–20
Number of operations per job (h)	3–15
Flexibility ratio (*flex*)	0.07–1.00
Batch and Computational Configuration	
Environment batch size per process	32
Global environment batch size	256
PPO minibatch size per process	256
Global PPO minibatch size	2,048
Active training time	≈60.21 hours
Total computational cost	≈481.68 GPU-hours

Table 6: Overall comparison on 128 synthetic 100×20 instances.

Statistic	Value
Number of instances	128
Operations per instance	2,000
Mean OR-Tools makespan ($C_{\max}^{\mathrm{OR}}$, UB)	5,011.77
Mean AnyFJS makespan ($C_{\max}^{\mathrm{model}}$)	5,230.48
Mean $\mathrm{Gap}_{\mathrm{UB}}$	4.3608%
Std. dev. of $\mathrm{Gap}_{\mathrm{UB}}$	0.9958 pp
Median $\mathrm{Gap}_{\mathrm{UB}}$	4.3569%
Range of $\mathrm{Gap}_{\mathrm{UB}}$	[2.5814, 7.4345]%
Mean AnyFJS search time	163.04 s
Std. dev. of search time	13.34 s

Batch and Computational Configuration. Each distributed process handles 32 scheduling environments, resulting in a global environment batch size of 256. The PPO minibatch size is 256 per process and 2,048 globally. The complete training procedure requires approximately 60.21 hours of active training time, corresponding to a total computational cost of approximately 481.68 GPU-hours.

A.4 Results on Large-scale FJSP Instances

Table 6 summarizes the performance of AnyFJS on 128 large-scale synthetic 100×20 FJSP instances, where each instance contains 100 jobs, 20 machines, and 2,000 operations in total. OR-Tools is used as the reference solver, and its obtained makespan is treated as an upper bound (UB)



Figure 7: The Gantt charts of AnyFJS and peer methods on all the selected representative instances.

for comparison. The purpose of this experiment is to evaluate whether AnyFJS can maintain effective scheduling performance when the problem size substantially exceeds that of the training instances.

For these 128 instances, the mean makespan obtained by OR-Tools is 5,011.77, while AnyFJS achieves a mean makespan of 5,230.48. The resulting mean gap with respect to the OR-Tools upper bound is 4.3608%, indicating that the solutions produced by AnyFJS remain relatively close to those obtained by the reference solver even at this extremely large problem scale. The median gap is 4.3569%, which is almost identical to the mean gap, suggesting that the overall performance is stable across the test instances and is not dominated by a small number of particularly difficult cases.

The standard deviation of the gap is 0.9958 percentage points, while the gap ranges from 2.5814% to 7.4345%. This relatively narrow range further demonstrates the robustness of AnyFJS across different large-scale instances. In particular, even the largest observed gap remains below 8%, indicating that the model consistently produces competitive schedules rather than exhibiting substantial performance degradation on individual instances.

In terms of computational efficiency, AnyFJS requires an average search time of 163.04 s per instance, with a standard deviation of 13.34 s. Importantly, the model is able to directly generate high-quality schedules for instances containing 2,000 operations, demonstrating that the learned scheduling policy can be effectively transferred from the small-scaled training distribution to substantially larger problem instances.

Overall, these results demonstrate the strong scalability and generalization capability of AnyFJS on extremely large-scale FJSPs. Although OR-Tools provides a stronger reference solution in terms of solution quality, AnyFJS obtains solutions within an average gap of only 4.3608% while maintaining a bounded and short but stable search time. This result provides empirical evidence that the learned policy can scale beyond the problem sizes encountered during training and remain effective for practical large-scale scheduling scenarios.

A.5 Gantt Chart Comparison on all Selected FJSP Instances

Figure 7 provides a qualitative comparison of AnyFJS and representative baseline methods on selected FJSP instances from different families and scales. AnyFJS generally generates more compact

and well-coordinated schedules, with fewer unnecessary idle periods and more balanced machine utilization. These characteristics remain consistent across different problem families and scales, demonstrating the robustness and transferability of the learned scheduling policy. The results further suggest that AnyFJS can effectively capture global scheduling patterns and maintain competitive scheduling quality as the problem size increases.