PDSCHEDULER: PERSISTENT DYNAMIC POLICY DECOMPOSITION FOR FLEXIBLE JOB-SHOP SCHEDULING

Anonymous authors
Paper under double-blind review

ABSTRACT

Efficient neural scheduling for the Flexible Job Shop Scheduling Problem (FJSP) requires coordinating global assignment guidance with decisions that adapt as the schedule evolves. Assignment-first approaches can restrict subsequent scheduling through early machine commitments, while autoregressive policies repeatedly process structural information throughout schedule construction. We propose PDScheduler, a neural scheduling framework based on Persistent Dynamic Policy Decomposition (PDPD). Rather than separating assignment from sequencing, PDPD decomposes scheduling guidance into a persistent prior derived from the problem instance and a dynamic residual that adapts to the evolving partial schedule, while preserving joint decisions over feasible operation and machine pairs. A masked discrete diffusion model generates a reusable assignment heatmap once per instance, providing soft guidance without fixing machine assignments. A lightweight residual actor then reweights this guidance over feasible candidates according to the evolving partial schedule. This design moves global prediction outside the sequential decision loop while preserving state-adaptive schedule construction. We further characterize the optimal persistent score under an idealized squared error projection. Experiments show that PDScheduler outperforms classical dispatching rules and state-of-the-art neural baselines on synthetic FJSP instances under both in-distribution and out-of-distribution evaluations, and remains competitive with the strongest baselines on public benchmarks while running 1.8× faster, demonstrating its superior solution quality and computational efficiency.

1 INTRODUCTION

The Flexible Job Shop Scheduling Problem (FJSP) is a classical combinatorial optimization problem with important applications in manufacturing and production planning (Brandimarte, 1993; Pezzella et al., 2008; Dauzère-Pérès et al., 2024). Each job consists of a sequence of operations linked by precedence constraints, and each operation can be processed by any machine in its eligible set (Brucker & Schlie, 1990; Dauzère-Pérès et al., 2024). Solving an instance therefore requires jointly determining the machine assigned to each operation and the processing order on each machine. These decisions are tightly coupled: machine assignments shape resource competition, while processing orders determine when the assigned machines become available. Effective scheduling must account for both the structure of the problem instance and the consequences of the evolving partial schedule.

Two common design choices illustrate this challenge. Some methods first assign operations to machines and then optimize processing orders, dividing the solution process into two stages (Brandimarte, 1993; Chen et al., 2023). For each fixed set of assignments, the downstream scheduler can optimize processing orders but cannot revisit those machine choices. Consequently, alternatives that would lead to better schedules may be excluded unless additional assignments are explored. Figure 1(a) illustrates this separation of assignment and sequencing. Autoregressive FJSP methods instead interleave operation selection and machine assignment during schedule construction, retaining the coupling between assignment and sequencing (Lei et al., 2022; Song et al., 2023; Wang et al., 2024; Zhao et al., 2025; Xiao et al., 2026).When these policies recompute the current state representation at every step, they repeatedly process structural and temporal information together. This supports adaptation to the current schedule but can incur substantial repeated computation as instances grow, as illustrated in Figure 1(b).

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.

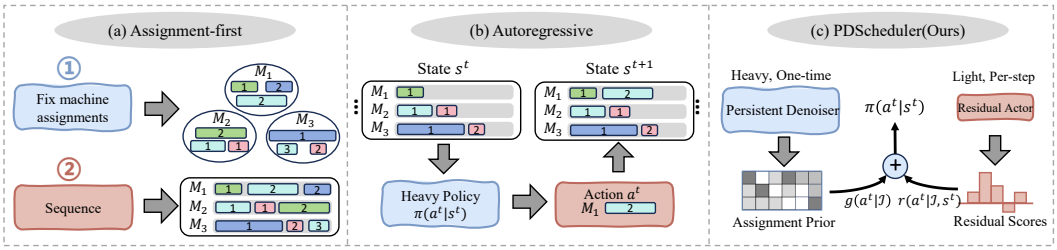


Figure 1: Comparison of scheduling designs. (a) Assignment-first decomposition fixes machine assignments before optimizing processing orders, restricting the decisions available to the downstream scheduler. (b) State-reencoded construction jointly selects operation–machine pairs but repeatedly processes the evolving scheduling state. (c) PDScheduler decomposes decision guidance into a persistent assignment prior generated once per instance and a dynamic residual updated at each step. The two components jointly score feasible operation–machine pairs, preserving state-dependent decisions without advance commitment to machine assignments or repeated global generation.

These limitations motivate a different question: *how can global scheduling guidance be reused without committing to decisions that should depend on the evolving schedule?* The processing times, machine eligibility, and precedence constraints of the original instance remain fixed, whereas machine availability and competition among feasible actions change during construction. The fixed structure can support a reusable assignment prior, even though the best machine choice for an operation may change with the partial schedule. This suggests decomposing scheduling guidance into a persistent component derived from the problem instance and a dynamic residual that adapts the guidance to the current partial schedule. Crucially, the decomposition concerns scheduling guidance rather than the underlying decision variables: assignment and sequencing remain coupled through the joint selection of operations and machines.

We propose PDScheduler, a neural scheduling framework built on Persistent Dynamic Policy Decomposition (PDPD), as illustrated in Figure 1(c). Its persistent component is a masked discrete diffusion model (Austin et al., 2021; Sahoo et al., 2024; Wang et al., 2026), that operates on a heterogeneous graph of operations and machines and generates an assignment heatmap during a preprocessing phase performed once per instance. The heatmap represents soft preferences over all eligible machines for each operation, rather than committing to fixed assignments or a partial schedule. Its dynamic component is a lightweight residual actor that uses the current partial schedule to compute corrections for feasible candidates. Self-attention over the candidate set and information about earliest finish times allow these corrections to reflect the current scheduling context. Together, the two components determine the next operation and its machine, retaining joint scheduling decisions while keeping global prediction outside the sequential construction loop.

Reusable encodings of problem instances and solution pipelines that combine global prediction with local decision making provide important precedents for this design (Kool et al., 2019; Sun & Yang, 2023; Li et al., 2023; 2024; Sanokowski et al., 2024). COExpander further integrates global prediction with information from partial solutions through adaptive decision granularity (Ma et al., 2025). Our focus is instead an explicit decomposition of scheduling guidance into a persistent assignment prior and residual corrections that account for the current scheduling state. This prior explicitly represents assignment preferences and contributes directly to action scoring, rather than serving only as a reusable latent representation. Its role is to guide subsequent decisions, not to determine them in advance.

We also formalize this decomposition at the level of idealized decision scores. With an unrestricted dynamic residual, the decomposed form can represent the original decision score. Under a fixed reference distribution over valid scheduling states, the conditional mean of the idealized score given the instance provides the optimal persistent approximation in terms of squared error. The residual captures deviations from this approximation. This analysis characterizes the roles of the two components; the effectiveness and efficiency of the lightweight neural instantiation are evaluated empirically.

Our main contributions are as follows:

- We introduce PDScheduler based on PDPD, separating reusable assignment guidance from dynamic corrections while retaining joint assignment and sequencing decisions. We characterize the decomposition and its optimal persistent projection at the level of idealized decision scores.
- We instantiate this decomposition with a masked discrete diffusion generator and a lightweight residual actor. The generator produces a persistent soft assignment prior, while the actor adapts its influence at each construction step without requiring repeated global generation or advance commitment to machine assignments.
- We demonstrate that PDScheduler achieves state-of-the-art makespan with lower inference cost on synthetic FJSP instances and generalizes to larger scales and public benchmark suites without further training, and that separately trained variants remain competitive on the job shop scheduling problem (JSSP) and the flexible flow shop scheduling problem (FFSP).

2 Related Work

Neural constructive scheduling. Priority dispatching rules are widely studied in job-shop scheduling (Haupt, 1989; Sels et al., 2012). Deep reinforcement learning extends this approach by learning constructive policies: L2D learns dispatching policies for JSSP (Zhang et al., 2020), while FJSP methods learn operation selection and machine assignment using graph representations and attention mechanisms (Lei et al., 2022; Song et al., 2023; Wang et al., 2024; Zhao et al., 2025). RESCHED uses a simplified state representation and a Transformer policy (Xiao et al., 2026). For FFSP, MatNet encodes processing-time matrices and reuses the resulting representations during autoregressive construction (Kwon et al., 2021). These policies construct schedules sequentially, but differ in how they encode and reuse instance information. Another approach separates machine assignment from sequencing: fixing the assignments yields a JSSP, enabling hierarchical optimization (Brandimarte, 1993). Learned two-stage models also exploit this structure (Chen et al., 2023). For each fixed assignment, the sequencing stage optimizes processing orders; exploring other machine choices requires considering additional assignments.

Diffusion-based priors and persistent guidance. DIFUSCO generates solution heatmaps through graph-based diffusion and uses a separate decoder, optionally combined with local or tree search (Sun & Yang, 2023). T2T learns instance-conditioned solution distributions and uses objective-guided gradient search at test time (Li et al., 2023). StruDiCO introduces variable-absorption noising to construct structured trajectories of partial solutions (Wang et al., 2025). Supervised graph networks also predict heatmaps for combinatorial optimization (Joshi et al., 2019). PDScheduler uses a masked-diffusion denoiser to predict soft machine assignment preferences. A single evaluation at the fully masked state produces guidance that is reused throughout schedule construction, while the actor computes state-dependent corrections. Proposition 1 characterizes the optimal fixed approximation to prescribed logits under expected squared error; an unrestricted residual can recover the target logits exactly. In the implemented policy, the learned persistent scores enter as additive logits over feasible operation–machine pairs, allowing the actor to adapt machine choices and processing orders to the evolving schedule. The assignment prior does not prune the feasible action set.

3 Problem formulation

An instance $\mathcal{I}$ of the flexible job shop scheduling problem consists of jobs $\mathcal{J}$, machines $\mathcal{M}$, and operations $\mathcal{O}$. Each job is an ordered chain of operations that must be processed in sequence; we write $i \prec i'$ when operation i must complete before operation i' of the same job can start. Operation $i \in \mathcal{O}$ can be processed by any machine in its eligible set $\mathcal{M}_i \subseteq \mathcal{M}$, with processing time $p_{ij} > 0$ on machine $M_j \in \mathcal{M}_i$. Processing is non-preemptive, and each machine processes at most one operation at a time. A feasible schedule assigns every operation to one eligible machine and fixes its start time subject to the precedence and machine-capacity constraints. Letting C_i denote the completion time of operation i, the objective is to minimize the makespan

$$C_{\max} = \max_{i \in \mathcal{O}} C_i. \tag{1}$$

FJSP reduces to the JSSP when $|\mathcal{M}_i| = 1$ for every operation, and to the FFSP when all jobs share a common routing, i.e., all job chains have the same length and $\mathcal{M}_i = \mathcal{M}_{i'}$ whenever operations i and i' occupy the same position in their respective chains.

Constructive scheduling. We build schedules constructively over decision steps $t = 1, \ldots, |\mathcal{O}|$. The partial schedule at step t records the machine and start time of every scheduled operation. An operation becomes ready once its predecessor in the job chain has been scheduled, and any pairing of a ready operation with an eligible machine is a feasible action. Committing an action fixes the operation's machine and starts it at the earliest time consistent with the partial schedule. After $|\mathcal{O}|$ steps the schedule is complete, and its cost is the makespan.

4 METHODOLOGY

Figure 2 summarizes how PDScheduler combines persistent assignment guidance with dynamic residual corrections. Once per instance, a persistent denoiser processes the initial instance graph with fully masked assignment inputs and produces an assignment prior together with structural embeddings. These outputs remain fixed throughout schedule construction. At each decision step, a recurrent residual actor combines the persistent outputs with dynamic candidate features to predict corrections for feasible pairs of operations and machines. The corrections are added to the persistent assignment scores, and the resulting policy selects an action that updates both the partial schedule and the actor's memory.

We use i to index operations and j to index machines. The initial graph $G_{\mathcal{I}} = (V_O, V_M, E_{\mathrm{OM}}, E_{\mathrm{prec}})$ contains operation and machine nodes, compatibility edges, and direct precedence edges within each job. The denoiser F_ϕ produces operation, machine, edge, and graph embeddings, denoted by $\mathbf{o}_i$, $\mathbf{m}_j$, $\mathbf{e}_{ij}$, and $\mathbf{g}$. The actor, parameterized by θ, uses the augmented state $\bar{s}_t = (s_t, \mathbf{c}_t)$, where s_t is the partial scheduling state and $\mathbf{c}_t$ is its recurrent memory.

Persistent dynamic decomposition. Persistent Dynamic Policy Decomposition (PDPD) combines reusable instance-level preferences with state-dependent corrections. Rather than separating assignment from sequencing, it decomposes the action score itself. At decision step t, for each feasible pair $a = (i, j) \in \mathcal{A}_t$,

$$\pi(a \mid \mathcal{I}, \bar{s}_t) = \mathrm{softmax}_{a \in \mathcal{A}_t} \Big(\underbrace{g(a \mid \mathcal{I})}_{\text{persistent}} + \underbrace{r(a \mid \mathcal{I}, \bar{s}_t)}_{\text{dynamic}} \Big), \tag{2}$$

where the persistent score g is computed once per instance and the residual r is recomputed at every step (Figure 1(c)); equivalently, the residual multiplicatively reweights the prior, $\pi \propto e^g \cdot e^r$. The prior guides machine choices without fixing them: assignment and sequencing remain jointly determined when an action is selected.

Proposition 1 (Optimal persistent approximation)**.** *Fix an instance $\mathcal{I}$ and a reference distribution $d(\cdot \mid \mathcal{I})$ over augmented scheduling states. Let $\mathbf{w}^*(\mathcal{I}, \bar{s}) \in \mathbb{R}^{|E_{\mathrm{OM}}|}$ be a prescribed target logit vector defined before feasibility masking, with finite second moments. The unique fixed vector minimizing the expected squared approximation error is*

$$\begin{aligned} \mathbf{g}^*(\mathcal{I}) &= \operatorname*{arg\,min}_{\mathbf{u} \in \mathbb{R}^{|E_{\mathrm{OM}}|}} \mathbb{E}_d \left[\|\mathbf{w}^*(\mathcal{I}, \bar{s}) - \mathbf{u}\|_2^2 \right] \\ &= \mathbb{E}_d \left[\mathbf{w}^*(\mathcal{I}, \bar{s}) \right]. \end{aligned} \tag{3}$$

The corresponding residual $\mathbf{r}^*(\mathcal{I}, \bar{s}) = \mathbf{w}^*(\mathcal{I}, \bar{s}) - \mathbf{g}^*(\mathcal{I})$ has zero mean.

This result identifies the persistent approximation that minimizes the expected squared magnitude of the remaining corrections. It motivates reusing assignment guidance across construction steps while adapting its influence through a dynamic residual. The proof is provided in Appendix A.4. The following sections describe how PDScheduler learns the assignment prior, computes the residual, and optimizes the resulting scheduling policy.

4.1 PERSISTENT ASSIGNMENT GUIDANCE VIA MASKED DIFFUSION

The persistent component learns assignment preferences from the complete instance graph before schedule construction. It produces two complementary outputs: an assignment prior that contributes

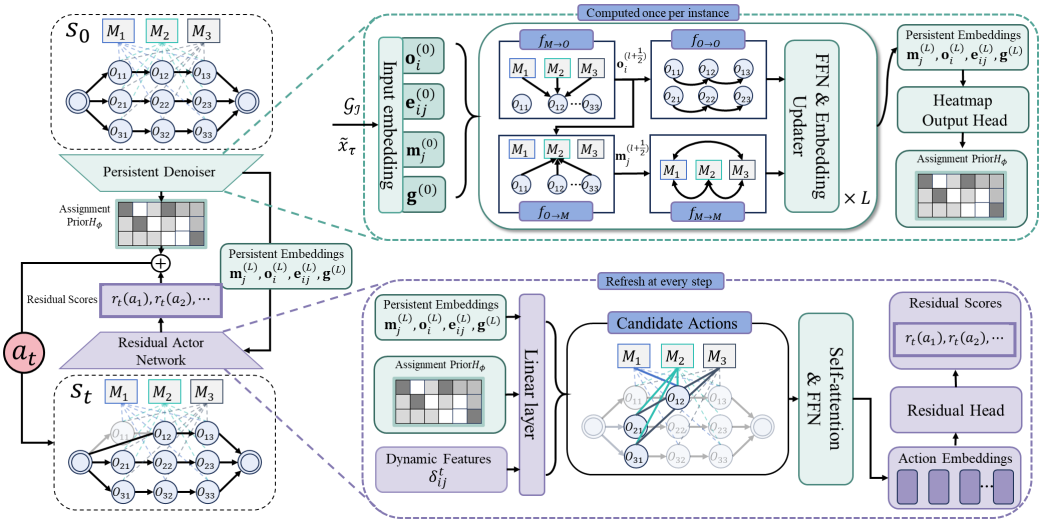


Figure 2: Overview of the PDScheduler. Left: schedule construction using a persistent assignment prior and a dynamic residual actor. Top right: the denoiser processes the fully masked initial graph once per instance and exports assignment probabilities and structural embeddings. Bottom right: the actor combines cached outputs with dynamic candidate features to score feasible actions.

directly to action scoring, and structural embeddings that provide context for the residual decision network. We learn these outputs through masked denoising, using the graph structure and partially observed assignments to predict the hidden machine choices.

Assignment representation and corruption. For each operation i, let $\mathcal{M}_i$ denote its compatible machines and let y_i be its assigned machine in a greedy reference schedule produced by RESCHED (Xiao et al., 2026). These extracted assignments serve as the sole supervision for denoiser pretraining: they are cheap to obtain at scale, yet provide a strong and unambiguous assignment signal. Each assignment is represented by a one-hot indicator over $\mathcal{M}_i$. We sample a diffusion step $\tau \sim \mathrm{Uniform}\{1, \ldots, T\}$ independently for each instance and mask its assignments with probability $q_\tau = \sin^2(\pi\tau/(2T))$. Given independent indicators $z_i \sim \mathrm{Bernoulli}(q_\tau)$, the corrupted edge value is $x_{ij,\tau} = (1 - z_i)\mathbf{1}[y_i = j]$. Masking therefore hides the machine choice for an entire operation without deleting its compatibility edges. Visible assignments provide context for predicting hidden ones, while larger corruption levels require the predictor to rely more on the instance structure. At $\tau = T$, all assignments are masked.

Heterogeneous graph denoising. The denoiser conditions its predictions on the initial instance graph $G_\mathcal{I}$, the corrupted assignments $\mathbf{x}_\tau$, and the diffusion step τ. Operation and machine features are projected into node embeddings, while processing information and corrupted-assignment indicators are embedded into the compatibility edges; the encoded diffusion step is injected into the graph embedding to indicate the corruption level.

The network applies four heterogeneous graph layers. Within each layer, compatibility edges are first updated from their endpoint representations and the graph embedding. Messages then flow from machines to their compatible operations and from the updated operations back to the machines. Attention over direct precedence edges captures job dependencies, while attention among machines provides cross-resource context, and a graph-level update aggregates the resulting node representations. This design allows each assignment prediction to draw on both local compatibility information and the global structure of jobs and resources.

A shared output head maps the final representations to edge logits, normalized separately over the compatible machines of each operation. Thus, although every operation emits its own categorical distribution, all predictions are informed by shared context from graph message passing, and all compatible edges participate in this computation, including those of operations that are not yet ready for scheduling. Appendix A.1 details the input features, the ordered layer updates, and the output head.

Pretraining reconstructs the reference assignments at sampled corruption levels. Following the absorbing-state discrete diffusion objective (Austin et al., 2021; Sahoo et al., 2024), we optimize

$$\mathcal{L}_{\mathrm{rec}} = -\mathbb{E}_{\mathcal{I},\tau,\mathbf{z}}\left[\frac{w(\tau)}{N}\sum_{i=1}^{N} z_i \log p_\phi\left(y_i \mid i, G_{\mathcal{I}}, \mathbf{x}_\tau, \tau\right)\right], \tag{4}$$

where $N = |\mathcal{O}|$ and $w(\tau) = T(q_\tau - q_{\tau-1})/q_\tau$, with $q_0 = 0$. Only masked operations contribute to the reconstruction objective, while visible assignments remain available as conditioning information. The loss is normalized by the total number of operations in the instance.

Guidance extraction and inference budget. Although trained with a horizon of T corruption steps, the denoiser is not unrolled at inference. Let K denote the number of network evaluations used to obtain the persistent guidance. By default we use $K = 1$: once per instance, a single forward pass from the fully masked input yields

$$\begin{aligned} H_{ij} &= p_\phi\left(j \mid i, G_{\mathcal{I}}, \mathbf{0}, T\right), \\ b_{ij} &= \log \max\{H_{ij}, 10^{-12}\}. \end{aligned} \tag{5}$$

The numerical floor is used for actor inputs and action scoring; reconstruction uses unclipped log probabilities. The same forward pass produces the operation, machine, edge, and graph embeddings. All persistent outputs are computed once per instance, shared across its G rollouts, and retained throughout construction.

Additional evaluations can instead be used for iterative refinement, where confident assignment predictions provide context for subsequent denoiser calls. We use direct prediction as the default because the output serves as soft guidance for a learned decision network, rather than as a complete assignment that must be fixed before scheduling. This avoids repeated global prediction while leaving the actor free to revise machine preferences during construction.

4.2 Dynamic residual decision network

The decision network adapts persistent assignment guidance to the current partial schedule. Rather than treating the prior as a fixed machine choice, it evaluates feasible pairs of operations and machines jointly and predicts corrections to their persistent scores. This retains the coupling between assignment and sequencing while avoiding repeated processing of the full instance graph.

Candidate representation. At step t, each candidate $(i, j) \in \mathcal{A}_t$ is represented using its cached operation, machine, edge, and graph embeddings, together with the persistent score b_{ij} and dynamic scheduling features $\boldsymbol{\delta}_t(i, j)$. The dynamic features describe processing times, availability, scheduling progress, and resource competition. Separate linear projections map the persistent inputs into a shared space, while a multilayer perceptron embeds the dynamic features. These contributions are added together with a projected recurrent summary and normalized to form a candidate token. We denote the augmented actor state by $\bar{s}_t = (s_t, \mathbf{c}_t)$, where $\mathbf{c}_t$ summarizes previous decisions. The resulting representation exposes both the prior preference and the current scheduling context to the decision network.

Interaction among candidates. A single block of self-attention and feedforward transformations processes the feasible candidate tokens jointly, using residual connections and layer normalization. Each candidate can incorporate information from alternative machine choices for the same operation and from other operations competing for the same machine. The network can therefore evaluate an action relative to the current candidate set, rather than score each pair independently. This interaction allows the learned correction to reflect changes in resource competition even when the persistent prior remains unchanged.

Residual action scoring. A shared scalar head maps each updated candidate representation to a learned score correction. Together with a timing bias derived from the normalized earliest finish time and scaled by a learned weight, this forms the dynamic residual $r_\theta(\bar{s}_t, i, j)$. The final logits

and action probabilities are

$$\ell_t(i,j) = b_{ij} + r_\theta(\bar{s}_t, i, j),$$
$$\pi_{\phi,\theta}(a \mid \mathcal{I}, \bar{s}_t) = \frac{\exp(\ell_t(a)/\vartheta)}{\sum_{a' \in \mathcal{A}_t} \exp(\ell_t(a')/\vartheta)}. \tag{6}$$

Actions outside $\mathcal{A}_t$ have zero probability. The temperature is $\vartheta = 2$ for sampled training rollouts and $\vartheta = 1$ for greedy validation and evaluation.

The prior enters the policy through two complementary paths: it conditions the learned correction through the candidate representation and contributes directly to the final action score. The residual can change the ranking induced by the prior, so machine assignments remain undecided until their operations are selected. After each action, the partial schedule and recurrent summary are updated, while the persistent scores and embeddings remain unchanged throughout the rollout. Appendix A.1.2 provides the feature definitions, network equations, timing bias, and recurrent update.

5 EXPERIMENTS

We evaluate PDScheduler to answer three questions: (i) does persistent guidance improve scheduling quality, (ii) how well does the policy generalize to larger instances and public benchmarks, and (iii) which components contribute most to performance.

Setup. We train on SD1 and SD2 instances at four sizes, 10×5, 20×5, 15×10, and 20×10. The denoiser is pretrained on machine assignments extracted from RESCHED reference schedules, followed by policy optimization. Table 1(a) reports a paired synthetic comparison on 100 shared instances per setting, with OR-Tools (Perron & Furnon, 2025) solutions (1800-second budget) as references; the gap metric is defined in Appendix A.5 (Eq. equation 20). Table 1(b) evaluates 20×10 checkpoints on 30×10 and 40×10 instances without further training, and Table 1(c) transfers SD1 20×10 checkpoints to 130 public instances from the Brandimarte (Brandimarte, 1993) and Hurink edata/rdata/vdata (Hurink et al., 1994) benchmarks. Full training and dataset configurations are given in Appendix A.5, and the main conclusions are stable across three training seeds (Appendix A.6). Code and trained checkpoints are available at `https://anonymous.4open.science/r/PDScheduler-AF3D5742`.

Baselines and decoding. We compare against four priority dispatching rules (FIFO, SPT, MOPNR, MWKR) and three neural schedulers: HGNN (Song et al., 2023), DANIEL (Wang et al., 2024), and RESCHED (Xiao et al., 2026). We additionally include Fixed-SPT variants paired with FIFO, MOPNR, and MWKR; see Appendix A.5 for details. For neural methods, greedy decoding selects the highest-scoring feasible action, while Sampling-100 returns the best schedule among 100 sampled trajectories.

5.1 SYNTHETIC FJSP EVALUATION

Performance at the training sizes. Table 1(a) shows that PDScheduler achieves the lowest greedy gap in all eight distribution–size settings, reducing RESCHED's mean gap from 9.18% to 6.96% despite its assignment supervision being extracted from RESCHED's own schedules. This indicates that the scheduler does not merely reproduce the teacher: the learned assignment heatmap provides guidance that the downstream actor exploits beyond the teacher's greedy performance. With Sampling-100, PDScheduler improves on RESCHED in six of the eight settings; on SD1 20×10, sampling pushes the gap from -1.24% to -3.10%, i.e., below the OR-Tools reference obtained with an 1800-second budget.

Table 1: FJSP results: (a) paired synthetic development comparison; (b) cross-size generalization; (c) public benchmarks. Lower gaps are better. All learned models in (b) and (c) use 20×10 checkpoints.

(a) Synthetic development comparison: mean gap (%).

Dataset	Size	PDRs				Fixed-SPT			Greedy				Sampling-100		OR-Tools $C_{\max}$
		FIFO	SPT	MOPNR	MWKR	FIFO	MOPNR	MWKR	HGNN	DANIEL	RESCHED	Ours	RESCHED	Ours	
SD1	10×5	25.14	36.53	19.83	17.01	52.94	70.02	72.18	16.59	11.17	10.97	**6.16**	3.77	**3.28**	95.84
	20×5	16.07	22.77	14.81	12.28	33.46	44.63	50.52	13.89	5.88	4.42	**2.35**	1.94	**0.94**	185.08
	15×10	29.41	38.40	21.62	20.54	64.52	91.11	94.64	16.24	13.08	5.06	**4.64**	1.72	**1.07**	142.25
	20×10	23.14	31.21	13.46	11.24	51.48	76.81	81.25	11.61	1.95	0.51	**-1.24**	-0.88	**-3.10**	184.16
SD2	10×5	71.61	49.51	69.54	65.01	44.14	59.64	72.50	64.40	23.31	14.25	**11.70**	**6.07**	6.67	69.35
	20×5	68.90	38.71	69.00	67.01	34.80	46.36	59.05	69.46	11.00	7.30	**4.39**	2.70	**2.21**	130.99
	15×10	121.64	79.61	114.23	111.26	60.70	88.96	107.88	101.84	49.06	16.54	**15.39**	**9.89**	10.03	82.15
	20×10	127.67	74.29	121.49	119.20	53.12	83.74	104.91	116.46	29.03	14.38	**12.31**	8.83	**7.16**	100.64

(b) Cross-size generalization from 20×10: mean gap (%).

Dataset	Test	N	Greedy				Sampling-100			
			HGNN	DANIEL	RESCHED	Ours	HGNN	DANIEL	RESCHED	Ours
SD1	30×10	100	14.01	2.50	1.79	**1.04**	13.64	1.70	0.92	**-0.14**
	40×10	100	13.75	1.52	1.44	**0.96**	13.55	1.14	1.07	**-0.05**
	Weighted avg.	200	13.88	2.01	1.62	**1.00**	13.60	1.42	1.00	**-0.10**
SD2	30×10	100	121.79	11.95	4.03	**3.14**	112.73	4.89	1.01	**-0.24**
	40×10	100	108.51	-1.67	**-6.02**	-5.24	100.17	-6.57	**-8.25**	-8.22
	Weighted avg.	200	115.15	5.14	-1.00	**-1.05**	106.45	-0.84	-3.62	**-4.23**
All	Weighted avg.	400	64.52	3.58	0.31	**-0.03**	60.02	0.29	-1.31	**-2.16**

(c) Public FJSP benchmarks: mean gap to the reference upper bound (%).

Dataset	N	PDR	Fixed-SPT			Greedy				Sampling-100			
		MWKR	FIFO	MOPNR	MWKR	HGNN	DANIEL	RESCHED	Ours	HGNN	DANIEL	RESCHED	Ours
Brandimarte	10	27.82	69.75	78.43	80.15	29.53	13.52	**10.54**	15.61	20.40	8.65	**6.42**	8.82
Hurink (edata)	40	18.56	27.60	50.38	53.61	17.24	16.82	16.12	**13.73**	9.94	9.47	7.89	**7.38**
Hurink (rdata)	40	13.23	45.68	68.70	71.96	12.85	11.71	**11.57**	12.42	7.32	**5.30**	6.04	6.20
Hurink (vdata)	40	3.89	49.29	72.95	77.35	3.96	**2.98**	4.71	7.65	1.79	**0.66**	1.62	3.50
Weighted avg.	130	13.12	43.08	65.12	68.60	12.75	**10.74**	10.78	11.60	7.43	5.41	**5.28**	5.93

Notes. Gaps are instance means (%). In (a), all methods use the same 100 instances per setting and OR-Tools 1800-s references; the last column gives reference mean makespans. Fixed-SPT averages five seeded tie-breaking runs per instance, without best-of-K selection. Sampling-100 takes the best of 100 trajectories. Bold/underline mark the best/second-best learned result within each setting and decoding strategy. In (b), checkpoints are trained on the corresponding SD1/SD2 distribution; each test set has 100 instances. In (c), checkpoints are trained on SD1. Weighted averages use instance counts: 400 in (b), and $10 + 40 + 40 + 40 = 130$ in (c).

Table 2: Ablation on SD1 10×5. Δ is relative to the full model.

Variant	Mean $C_{\max}$	Δ (%)
Full model	**101.218**	**0.000**
Direct GNN	103.716	+2.546
Static only	133.384	+32.041
Dynamic only	113.710	+12.492
w/o static embeddings	113.368	+12.150
w/o heatmap scores	103.004	+1.786

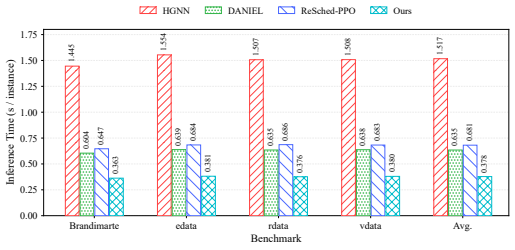


Figure 3: Greedy runtime on public FJSP benchmarks (s/instance), using SD1 20×10 checkpoints. Avg. is weighted over 130 instances.

Generalization to larger instances. Table 1(b) transfers the 20×10 checkpoints to 30×10 and 40×10 instances without further training. PDScheduler attains the lowest gap in six of the eight size–strategy combinations; both exceptions occur at SD2 40×10, where RESCHED is stronger (the Sampling-100 margin is only 0.03 percentage points). Its weighted mean gap over all 400 instances nevertheless improves over RESCHED under both greedy decoding (-0.03% vs. 0.31%) and Sampling-100 (-2.16% vs. -1.31%). Transfer to larger instances is therefore essentially lossless in aggregate, and the sampling budget converts into additional quality rather than compensating for weak greedy decisions.

Solution quality on public benchmarks. Table 1(c) shows that transfer to public benchmarks is distribution-dependent. PDScheduler achieves the best neural result on Hurink edata under both decoding strategies, and remains within roughly one percentage point of the strongest baseline in the 130-instance aggregate (11.60% vs. 10.74% greedy; 5.93% vs. 5.28% under Sampling-100), while DANIEL and RESCHED stay stronger on Brandimarte, rdata, and vdata. Since all checkpoints are trained only on SD1, this gap reflects the distribution shift between the synthetic training data and these suites rather than overfitting to the training sizes. We therefore assess quality jointly with inference cost below.

Greedy inference time. Figure 3 shows that PDScheduler is the fastest neural method on every public benchmark set: 0.378 s per instance on average, about $1.8\times$ faster than RESCHED and $4.0\times$ faster than HGNN. The same ordering holds on the synthetic training sizes and the larger test sets (Appendix A.6). Combined with the results above, PDScheduler is well suited to settings where schedules must be produced quickly: its aggregate gap stays within one percentage point of the best neural baseline, at roughly half the inference cost.

Ablation. Table 2 shows that both persistent outputs contribute: removing static embeddings degrades the mean makespan by 12.2% and removing heatmap scores by 1.8%, indicating that the embeddings carry most of the structural context while the heatmap provides complementary assignment preferences. The Static only and Dynamic only variants degrade by 32.0% and 12.5%, respectively, confirming that persistent guidance and dynamic decisions are complementary rather than redundant. Replacing the conditioned actor with a direct GNN also degrades quality ($+2.5\%$), supporting the complete design. Enlarging the refinement budget beyond $K{=}1$ brings no consistent quality gain while inference cost grows steadily (Appendix A.6), confirming the single-step persistent inference used throughout as the right operating point.

Generality on JSSP and FFSP. We evaluate problem-specific variants of PDScheduler on JSSP and FFSP, with full results in Appendices A.7 and A.8. For JSSP, only the actor is trained, using synthetic 10×10 instances without denoiser pretraining. This actor is evaluated directly on the Taillard (Taillard, 1993) and DMU (Demirkol et al., 1998) benchmarks, reaching 100×20 and 50×20, respectively, without further training. PDScheduler achieves the best or tied-best neural greedy gap in five of the sixteen size groups and ranks second among the reported neural methods in the aggregate results of both suites. Its average gaps are 20.65% on Taillard and 27.58% on DMU; the latter is 0.73 percentage points above RESCHED. On Taillard, its average gap is lower than those of L2D (Zhang et al., 2020) and RL-GNN (Park et al., 2021). For FFSP, a separate PDScheduler model is trained on FFSP20 and evaluated on FFSP20, FFSP50, and FFSP100 without further training, where the suffix denotes the number of jobs. Under the evaluation protocol described in Appendix A.8, it achieves the best greedy makespan at the training size and the shortest greedy inference time at all three sizes, with speedups of up to $2.3\times$ over RESCHED. Its makespan remains higher than RESCHED's on FFSP50 and FFSP100, showing a trade-off between cross-size solution quality and inference time.

6 Discussion

We have presented PDScheduler, a scheduling framework that reinterprets machine assignment as a persistent prior rather than an early commitment or a repeatedly recomputed representation. Under Persistent Dynamic Policy Decomposition, a diffusion-generated assignment heatmap provides soft, reusable guidance, while a residual actor adapts this guidance to the evolving partial schedule; our analysis characterizes the optimal persistent score as the best state-independent approximation to the prescribed logits. Empirically, PDScheduler improves solution quality over strong neural baselines on synthetic FJSP instances, transfers to larger instances without retraining, and remains within a small margin of the strongest baselines on public FJSP, JSSP, and FFSP benchmarks while reducing inference time by up to $4\times$. More broadly, PDPD offers a template for combining global learned priors with sequential decision-making in combinatorial optimization. The prior is currently distilled from reference schedules of existing solver and remains fixed at inference; removing this dependence through end-to-end training, and closing the residual quality gap on distribution-shifted benchmarks, are natural next steps.

AI USE STATEMENT

We used large language models (e.g., Kimi) as general-purpose assistants for (i) language polishing of the manuscript (grammar, wording, and clarity), (ii) suggesting occasional non-substantive code snippets, (iii) code review of the PDSCHEDULER implementation, and (iv) checking the potential overlap between the ideas in this work and the existing literature. LLMs were not used for problem formulation, algorithm or model design, experimental design or analysis, data generation, or drawing conclusions. All core code and technical content was implemented and verified by the authors, and all LLM-assisted text and code was reviewed and edited by the authors. We take full responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, pp. 17981–17993, 2021.

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993.

Peter Brucker and Rainer Schlie. Job-shop scheduling with multi-purpose machines. *Computing*, 45(4):369–375, 1990. doi: 10.1007/BF02238804.

Wenbo Chen, Reem Khir, and Pascal Van Hentenryck. Two-stage learning for the flexible job shop scheduling problem. arXiv preprint arXiv:2301.09703, 2023.

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. doi: 10.1016/j.ejor.2023.05.017.

Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

R. Haupt. A survey of priority rule-based scheduling. *OR Spektrum*, 11(1):3–16, 1989.

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.

Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. In *NeurIPS Workshop on Graph Representation Learning*, 2019.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

Yeong-Dae Kwon, Jinho Choo, Iljoo Yoon, Minah Park, Duwon Park, and Youngjune Gwon. Matrix encoding networks for neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 34, pp. 5138–5149, 2021.

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.

Yang Li, Jinpei Guo, Runzhong Wang, and Junchi Yan. T2T: From distribution learning in training to gradient search in testing for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 36, pp. 50020–50040, 2023.

Yang Li, Jinpei Guo, Runzhong Wang, Hongyuan Zha, and Junchi Yan. Fast T2T: Optimization consistency speeds up diffusion-based training-to-testing solving for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 37, pp. 30179–30206, 2024.

Jiale Ma, Wenzheng Pan, Yang Li, and Junchi Yan. COExpander: Adaptive solution expansion for combinatorial optimization. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 42130–42164. PMLR, 2025.

Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: Representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021. doi: 10.1080/00207543.2020.1870013.

Laurent Perron and Vincent Furnon. OR-Tools. `https://developers.google.com/optimization/`, 2025. Version 9.12.

Ferdinando Pezzella, Gianluca Morganti, and Giampiero Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, 35(10):3202–3212, 2008. doi: 10.1016/j.cor.2007.02.014.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184, 2024.

Sebastian Sanokowski, Sepp Hochreiter, and Sebastian Lehner. A diffusion model framework for unsupervised neural combinatorial optimization. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 43346–43367, 2024.

Veronique Sels, Nele Gheysen, and Mario Vanhoucke. A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions. *International Journal of Production Research*, 50(15):4255–4270, 2012.

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.

Zhiqing Sun and Yiming Yang. DIFUSCO: Graph-based diffusion solvers for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 36, pp. 3706–3731, 2023.

Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

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.

Yu Wang, Yang Li, Junchi Yan, and Yi Chang. StruDiCO: Structured denoising diffusion with gradient-free inference-stage boosting for memory and time efficient combinatorial optimization. In *Advances in Neural Information Processing Systems*, 2025.

Yu Wang, Yang Li, Jiale Ma, Junchi Yan, and Yi Chang. NExCO: Native solution expansion for diffusion-based combinatorial optimization. In *International Conference on Learning Representations*, 2026.

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

Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Chi Xu. Learning to dispatch for job shop scheduling via deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1621–1632, 2020.

Peng Zhao, You Zhou, Di Wang, Zhiguang Cao, Yubin Xiao, Xuan Wu, Yuanshu Li, Hongjia Liu, Wei Du, Yuan Jiang, and Liupu Wang. Dual operation aggregation graph neural networks for solving flexible job-shop scheduling problem with reinforcement learning. In *Proceedings of the ACM Web Conference (WWW)*, pp. 4089–4100, 2025.

A Appendix

A.1 Architecture Details

We use i to index operations and j to index machines, following Section 4; τ denotes the diffusion timestep. All time features are normalized by $s_{\mathcal{I}} = \max\{1, \max_{(i,j)\in E_{\mathrm{OM}}} p_{ij}\}$, which is fixed for each instance. The compatible-machine neighborhood of operation i is $\mathcal{N}_M(i) = \{j : (i,j) \in E_{\mathrm{OM}}\}$, and the compatible-operation neighborhood of machine j is $\mathcal{N}_O(j) = \{i : (i,j) \in E_{\mathrm{OM}}\}$.

A.1.1 Persistent graph denoiser

Inputs and capacity. The denoiser uses four layers, width $d = 192$, eight attention heads of width $d_h = 24$, and zero dropout. Layers have separate parameters. Operation, machine, edge, and graph features have dimensions 10, 3, 5, and 5, respectively; the corrupted assignment indicator supplies one additional edge channel. Operation features summarize initial readiness, job positions, processing-time statistics, eligibility, and suffix workloads. Machine features summarize coverage and processing times; edge features include processing time, two compatibility ranks, a fastest-machine indicator, and relative earliest finish time. Ranks count strictly smaller processing times, so ties receive the same rank. Graph features are $[J/100, M/100, N/1000, N_{\mathrm{ready}}/N, \widehat{C}_0/s_{\mathcal{I}}]$, where $\widehat{C}_0$ is the maximum job workload obtained using minimum processing times. These features do not use teacher assignments.

The initial graph embedding is the sum of a graph-feature projection and a timestep projection:

$$\mathbf{g}^{(0)} = \mathrm{MLP}_G(\mathbf{f}^G) + \mathrm{MLP}_\tau(\mathrm{SinEnc}_{128}(\tau)). \tag{7}$$

The timestep MLP has widths $128 \to 192 \to 192$ and a SiLU activation. Node and edge inputs are embedded into the same hidden width.

Ordered layer updates. A layer first updates the edge embeddings using the current endpoints and graph embedding:

$$\mathbf{e}'_{ij} = \mathrm{LN}(\mathbf{e}_{ij} + \mathrm{MLP}_E([\mathrm{LN}(\mathbf{m}_j); \mathrm{LN}(\mathbf{o}_i); \mathrm{LN}(\mathbf{e}_{ij}); \mathrm{LN}(\mathbf{g})])\,. \tag{8}$$

The MLP has widths $4d \to 2d \to d$ with SiLU; the layer normalizations have separate parameters.

Next, machines send messages to operations. For each head,

$$\begin{aligned} \mathbf{q}_{ij} &= W_q\mathbf{o}_i + W_{qe}\mathbf{e}'_{ij}, \qquad \mathbf{k}_{ij} = W_k\mathbf{m}_j + W_{ke}\mathbf{e}'_{ij}, \\ \mathbf{u}_{ij} &= W_v\mathbf{m}_j + W_{ve}\mathbf{e}'_{ij}, \quad \alpha_{ij} = \mathrm{softmax}_{j\in\mathcal{N}_M(i)}(\mathbf{q}_{ij}^\top\mathbf{k}_{ij}/\sqrt{d_h}). \end{aligned} \tag{9}$$

Head outputs $\sum_j \alpha_{ij}\mathbf{u}_{ij}$ are concatenated and projected to obtain $\mathrm{Attn}_{M\to O}(i)$. The operation update is

$$\widetilde{\mathbf{o}}_i = \mathrm{LN}(\mathbf{o}_i + \mathrm{Attn}_{M\to O}(i)), \qquad \mathbf{o}'_i = \mathrm{LN}(\widetilde{\mathbf{o}}_i + \mathrm{FFN}_O(\widetilde{\mathbf{o}}_i)). \tag{10}$$

The reverse branch has separate parameters and analogous machine updates, normalizing over $\mathcal{N}_O(j)$. It uses the already updated $\mathbf{o}'_i$ and the same $\mathbf{e}'_{ij}$: the two directions are sequential, not parallel.

Operation attention then follows direct precedence edges from predecessors to successors, without reverse edges or attention self loops. Machine attention is dense within an instance and includes self attention. For each branch the update has the form $\mathbf{h} \leftarrow \mathbf{h} + \mathrm{Attn}(\mathrm{LN}(\mathbf{h}))$ followed by $\mathbf{h} \leftarrow \mathrm{LN}(\mathbf{h} + \mathrm{FFN}(\mathbf{h}))$. Each node FFN is SwiGLU with gated intermediate width 384:

$$\mathrm{FFN}(\mathbf{h}) = W_3\,[\mathrm{SiLU}(W_a\mathbf{h} + \mathbf{b}_a) \odot (W_b\mathbf{h} + \mathbf{b}_b)] + \mathbf{b}_3.$$

Finally, the graph embedding is updated by

$$\mathbf{g}' = \mathbf{g} + \mathrm{MLP}_{3d\to d\to d}\left([\mathbf{g}; \mathrm{mean}_i\, \mathbf{o}_i; \mathrm{mean}_j\, \mathbf{m}_j]\right). \tag{11}$$

The node embeddings in the global update are the final node states of the current layer. All compatible edges participate, including edges of operations that are not initially ready. Diffusion masking hides the assignment indicator but does not remove graph edges.

Assignment head. After the last layer, let $\bar{\mathbf{e}}_{ij} = \mathrm{LN}(\mathbf{e}_{ij}^{(L)})$. The output MLP receives

$$[\mathbf{m}_j^{(L)}; \mathbf{o}_i^{(L)}; \bar{\mathbf{e}}_{ij}; \mathbf{g}^{(L)}; x_{ij,\tau}]. \tag{12}$$

The resulting logits are normalized over compatible machines separately for each operation, as in Section 4.1.

A.1.2 Residual actor interface

Effective dynamic correction. Let $\rho_{t,ij}$ denote the neural residual-head output. The correction used in Equation equation 6 is

$$r_\theta(\bar{s}_t, i, j) = \rho_{t,ij} - \lambda \frac{\mathrm{EFT}_t(i,j)}{s_{\mathcal{I}}}, \qquad \lambda = \mathrm{softplus}(\eta_\lambda).$$

The coefficient is initialized at 3 and optimized with the actor.

Residual and memory. Using the terminal prior H_{ij} from Equation equation 5, we set $b_{ij} = \max\{\log H_{ij}, \log(10^{-12})\}$. The candidate token is

$$\begin{aligned}\mathbf{z}_{t,ij}^{(0)} = \mathrm{LN}\big(&W_O \mathbf{o}_i + W_M \mathbf{m}_j + W_E \mathbf{e}_{ij} + W_G \mathbf{g} + W_b b_{ij} \\ &+ \mathrm{MLP}_{12\to 96\to 96}(\boldsymbol{\delta}_t(i,j)) + W_c \mathbf{c}_t\big).\end{aligned} \tag{13}$$

Biases are omitted for readability. One four-head attention block and a SiLU FFN of widths $96 \to 192 \to 96$ follow, each with a residual connection and subsequent layer normalization; dropout is zero. The final token $\mathbf{z}_{t,ij}^{(2)}$ feeds a linear residual head $\rho_{t,ij} = \mathbf{w}_r^\top \mathbf{z}_{t,ij}^{(2)} + b_r$, whose weights and bias are initialized to zero. The memory is initialized independently for every rollout and updated using the selected candidate token:

$$\mathbf{c}_0 = \mathbf{0} \in \mathbb{R}^{96}, \qquad \mathbf{c}_{t+1} = \mathrm{GRUCell}(\mathbf{z}_{t,a_t}^{(2)}, \mathbf{c}_t).$$

The memory is not detached between decision steps.

Action scores. The implemented temporal utility and scores are

$$\begin{aligned} u_t(i,j) &= -\,\mathrm{EFT}_t(i,j)/s_{\mathcal{I}}, \\ \ell_t(i,j) &= b_{ij} + \rho_{t,ij} + \lambda u_t(i,j), \\ \pi_{\phi,\theta}(a_t = (i,j)) &= \mathrm{softmax}_{\mathcal{A}_t}(\ell_t(i,j)/\vartheta). \end{aligned} \tag{14}$$

The dynamic residual in the decomposition is $\rho_{t,ij} + \lambda u_t(i,j)$. The scalar λ follows the initialization described above. The temperature ϑ is 2 for sampled training rollouts and 1 for greedy validation and evaluation. Padded candidates are excluded from attention and action selection.

A.2 Denoiser pretraining

The settings in this appendix describe the configured SD1/SD2 FJSP protocol. The protocol uses 50,000 labelled instances per training setting. Reference machine assignments are extracted from greedy schedules of the corresponding RESCHED teacher (Xiao et al., 2026); neither teacher action sequences nor teacher weights initialize the actor. For each graph n in a batch, sample τ_n independently and mask its operations independently conditional on τ_n:

$$\begin{aligned} \tau_n &\sim \mathrm{Uniform}\{1, \ldots, T\}, \quad T = 1000, \\ q_\tau &= \sin^2(\pi\tau/(2T)), \quad z_i \sim \mathrm{Bernoulli}(q_{\tau_n}), \\ x_{ij,\tau_n} &= (1 - z_i)\mathbf{1}[y_i = j]. \end{aligned} \tag{15}$$

Algorithm 1 Pretraining the persistent denoiser

Require: Reference dataset $\mathcal{D}_{\text{ref}}$; $T = 1000$;
1: Initialize ϕ
2: **for** each shuffled mini-batch $\mathcal{B} \subset \mathcal{D}_{\text{ref}}$ **do**
3: Sample $\tau_n \sim \text{Uniform}\{1, \dots, T\}$ and form $\mathbf{x}_{\tau_n}$ (Eq. equation 15) for each $n \in \mathcal{B}$
4: Evaluate $F_\phi(G_n, \mathbf{x}_{\tau_n}, \tau_n)$ in a batch; compute $\mathcal{L}_{\text{rec}}$ (Eq. equation 16)
5: Update ϕ with AdamW and gradient clipping
6: **end for**
7: **return** checkpoint with best validation accuracy on fully masked inputs

There is no random machine replacement or additional forced-terminal mixture. Fully masked examples can nevertheless arise under this sampler.

For graph n with N_n operations, define

$$\ell_n = -\frac{w(\tau_n)}{N_n} \sum_{i \in \mathcal{O}_n} z_i \log p_\phi(y_i \mid i, G_n, \mathbf{x}_{\tau_n}, \tau_n),$$
$$w(\tau) = T\frac{q_\tau - q_{\tau-1}}{q_\tau}, \qquad \mathcal{L}_{\text{rec}} = \frac{1}{B}\sum_{n=1}^{B} \ell_n. \tag{16}$$

The denominator is the total operation count, not the masked count. Graphs with no masked operations contribute zero and remain in the batch mean; they are not resampled. The loss uses unclipped log probabilities.

A.3 Actor optimization

For FJSP, the actor is optimized for 80 epochs with the denoiser frozen throughout; the persistent network receives no gradients in this stage and no reconstruction loss is added. The actor and its GRU are initialized from scratch. Each epoch generates 1024 new base instances, with $G = 8$ sampled rollouts per instance. For each base instance, the denoiser is evaluated once on its fully masked initial graph. The resulting persistent assignment scores and structural embeddings are cached and shared across all G rollouts, and reused at every decision step. Each rollout maintains its own partial schedule and recurrent memory.

For rollout costs C_{ng}, let $\bar{C}_n$ and σ_n be the mean and population standard deviation within the group. The implemented cost advantage and loss are

$$A_{ng} = \text{clip}\left(\frac{C_{ng} - \bar{C}_n}{\max\{\sigma_n, 1\}}, -3, 3\right), \tag{17}$$

$$\mathcal{L}_{\text{actor}} = \frac{1}{BG}\sum_{n,g}\left[\text{sg}(A_{ng})\sum_{t=1}^{N_n}\log \pi_{\phi,\theta}(a_t^{ng} \mid \mathcal{I}_n, \bar{s}_t^{ng}) - \frac{10^{-3}}{N_n}\sum_{t=1}^{N_n}\mathcal{H}(\pi_t^{ng})\right]. \tag{18}$$

The positive sign uses a cost advantage: smaller makespans receive negative advantages. Log probabilities are summed along a rollout, whereas entropy is averaged. The baseline includes all eight rollouts.

Optimization settings. Both stages use AdamW with weight decay 10^{-6} and gradient-norm clipping at 1. The pretraining learning rate is 3×10^{-4}. During actor optimization, the actor and λ use 10^{-4} and the denoiser remains frozen (learning rate zero). There is no additional learning-rate scheduler. The configured batch sizes are listed in Table 3. The actor checkpoint is selected by the lowest greedy mean makespan on a fixed 500-instance validation set over all 80 epochs.

A.4 Proof of the Persistent Projection

Fix an instance $\mathcal{I}$ and a distribution $d(\cdot \mid \mathcal{I})$ over nonterminal scheduling states. Let $\mathbf{w}^*(\mathcal{I}, s) \in \mathbb{R}^{|E_{\text{OM}}|}$ be a prescribed vector of finite logits before feasibility masking, with $\mathbb{E}_d\|\mathbf{w}^*\|_2^2 < \infty$. The superscript $*$ denotes a reference target, not necessarily an optimal policy. The state may include the actor's recurrent memory.

Table 3: Configured batch sizes per GPU for both SD1 and SD2. Actor batchsizes count base instances before the eight rollouts are generated.

Scale	Denoiser batch	Actor base batch
10×5	128	16
20×5	64	16
15×10	32	8
20×10	32	8

Proposition 1 (restated) (Persistent projection of prescribed logits)**.** *For the fixed instance, state distribution, and logit representation above,*

$$\mathbf{g}^* = \mathbb{E}_d[\mathbf{w}^*] = \arg\min_{\mathbf{g}\in\mathbb{R}^{|E_{\mathrm{OM}}|}} \mathbb{E}_d\|\mathbf{w}^* - \mathbf{g}\|_2^2.$$

The minimizer is unique. The exact residual $\mathbf{r}^* = \mathbf{w}^* - \mathbf{g}^*$ *has zero mean.*

Proof. Write $\boldsymbol{\mu} = \mathbb{E}_d[\mathbf{w}^*]$. The second-moment assumption ensures that this mean is well defined. For any fixed vector $\mathbf{g}$,

$$\begin{aligned}\mathbb{E}_d\|\mathbf{w}^* - \mathbf{g}\|_2^2 &= \mathbb{E}_d\|\mathbf{w}^* - \boldsymbol{\mu}\|_2^2 + \|\boldsymbol{\mu} - \mathbf{g}\|_2^2 \\ &\quad + 2\,\mathbb{E}_d\left[(\mathbf{w}^* - \boldsymbol{\mu})^\top(\boldsymbol{\mu} - \mathbf{g})\right] \\ &= \mathbb{E}_d\|\mathbf{w}^* - \boldsymbol{\mu}\|_2^2 + \|\boldsymbol{\mu} - \mathbf{g}\|_2^2. \end{aligned} \tag{19}$$

The cross term is zero because $\mathbb{E}_d[\mathbf{w}^* - \boldsymbol{\mu}] = \mathbf{0}$. The first term is independent of $\mathbf{g}$, and the second is uniquely minimized at $\mathbf{g} = \boldsymbol{\mu}$. Thus $\boldsymbol{\mu}$ is the unique optimal persistent approximation. Defining $\mathbf{r}^* = \mathbf{w}^* - \mathbf{g}^*$ gives $\mathbb{E}_d[\mathbf{r}^*] = \mathbf{0}$ and

$$\mathbb{E}_d\|\mathbf{w}^* - \mathbf{g}\|_2^2 = \mathbb{E}_d\|\mathbf{r}^*\|_2^2 + \|\mathbf{g} - \mathbf{g}^*\|_2^2.$$

This also establishes the minimum expected squared magnitude of an exact residual. □

Connection to the policy. For any fixed $\mathbf{g}$, setting $\mathbf{r} = \mathbf{w}^* - \mathbf{g}$ recovers the target logits exactly. Therefore, on any nonempty feasible action set,

$$\mathrm{softmax}_{\mathcal{A}}(\mathbf{g} + \mathbf{r}) = \mathrm{softmax}_{\mathcal{A}}(\mathbf{w}^*), \qquad e^{g(a)+r(a)} = e^{g(a)}e^{r(a)}.$$

This concerns prescribed logits and an unrestricted residual. It neither bounds the required actor capacity nor identifies the learned log heatmap with the ideal projection. The latter depends on the reference state distribution and logit representation, including the scores assigned to temporarily infeasible actions.

A.5 Experimental details

Datasets and evaluation splits. We evaluate PDScheduler on synthetic FJSP instances from SD1 and SD2, following the data-generation settings associated with HGNN (Song et al., 2023) and DANIEL (Wang et al., 2024). Each distribution includes four training sizes: 10×5, 20×5, 15×10, and 20×10, where the two dimensions denote the numbers of jobs and machines. For SD1, the number of operations per job is sampled between $\lfloor 0.8m \rfloor$ and $\lfloor 1.2m \rfloor$, where m is the number of machines. Each operation has between one and m eligible machines, and processing times lie in $[1, 20]$. The SD1 generator first samples a nominal processing time and then samples machine-specific times within a clipped 20% interval around it. SD2 uses processing times in $[1, 99]$. We retain the supplied instance files, including the mixed SD2 sets, throughout the paired comparisons.

At each training size, all methods are evaluated on the same 100 instances with the same reference solutions. Cross-size evaluation applies the 20×10 checkpoint trained on each distribution to 100 instances at each of 30×10 and 40×10, giving 400 instances across the two distributions. Public-benchmark evaluation uses SD1 20×10 checkpoints on 10 Brandimarte (Brandimarte, 1993) instances and 40 instances from each of the Hurink edata, rdata, and vdata suites (Hurink et al., 1994), for a total of 130 instances. Neither cross-size nor public-benchmark evaluation involves additional training.

Optimization and model selection. PDScheduler is trained in PyTorch using FP32 arithmetic and distributed data parallelism on two NVIDIA GeForce RTX 5090 GPUs, each with 32 GB of memory. The configured denoiser and actor batch sizes depend on the training size and are listed in Table 3. Each actor batch expands by a factor of eight through the per-instance rollouts.

Both stages use AdamW with weight decay 10^{-6} and gradient-norm clipping at 1. The denoiser pretraining learning rate is 3×10^{-4}, and the actor learning rate is 10^{-4}. The denoiser learning rate remains zero throughout actor training. Dropout is set to zero. We use a training sampling temperature of 2.0 and an entropy coefficient of 10^{-3}. Actor validation uses greedy decoding on a fixed set of 500 instances, with an evaluation batch size of 50 instances per GPU. We select the actor checkpoint with the lowest validation mean makespan. Sampling-100 evaluation uses temperature 1.0 and reports the smallest makespan among 100 complete trajectories.

Model configuration and training. PDScheduler is implemented in PyTorch. The persistent denoiser has four graph layers with hidden dimension 192 and eight attention heads. The residual actor has hidden dimension 96, four attention heads, and a GRU memory. The FJSP training configuration uses $50{,}000$ labelled instances per distribution–size setting. Machine-assignment labels are extracted from greedy schedules produced by the corresponding RESCHED teacher (Xiao et al., 2026). The denoiser is pretrained for 60 epochs. We then freeze the denoiser and optimize the actor for 80 epochs. The denoiser learning rate is 3×10^{-4} during pretraining and zero during actor optimization; the actor learning rate is 10^{-4}. Both stages use AdamW with weight decay 10^{-6} and gradient-norm clipping at 1. Actor training samples eight rollouts per generated instance. The denoiser checkpoint is selected by the highest validation assignment accuracy, and the actor checkpoint by the lowest greedy mean makespan on a fixed 500-instance validation set.

Metrics and aggregation. For evaluation instance n, let C_n be the resulting makespan and C_n^{ref} its reference value. For a set of N_{test} instances, we report

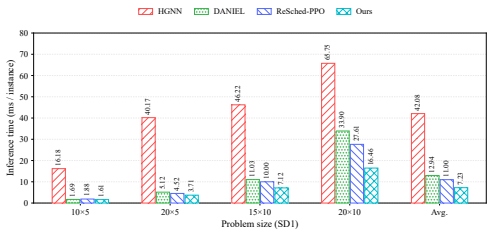
$$\mathrm{Gap} = \frac{100}{N_{\mathrm{test}}} \sum_{n=1}^{N_{\mathrm{test}}} \frac{C_n - C_n^{\mathrm{ref}}}{C_n^{\mathrm{ref}}}. \tag{20}$$

Synthetic FJSP references are OR-Tools solutions obtained with an 1800-second budget per instance. Public-benchmark gaps use the reference upper bounds associated with each benchmark. A negative gap indicates a makespan below the reference value. In particular, a time-limited OR-Tools solution need not be optimal. Gaps are averaged over instances, rather than computed from the ratio of aggregate makespans. Cross-suite averages are weighted by instance counts: 400 for cross-size FJSP evaluation and 130 for public FJSP benchmarks. Greedy and sampling results are ranked separately.

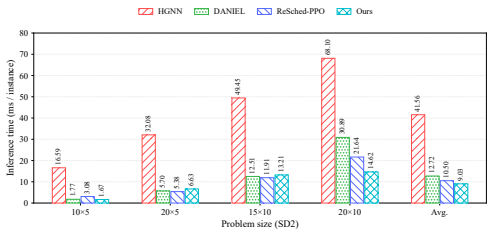
FJSP baselines. The rule-based baselines comprise FIFO, SPT, MOPNR, and MWKR. They prioritize arrival order, short processing times, the number of remaining operations, and remaining processing workload, respectively. The neural baselines are HGNN (Song et al., 2023), DANIEL (Wang et al., 2024), and RESCHED (Xiao et al., 2026). All comparisons in Table 1 use the shared evaluation sets described above.

Fixed-SPT variants. We also evaluate a two-stage heuristic that separates machine assignment from operation sequencing. First, every operation is assigned to an eligible machine with the shortest processing time. These assignments remain fixed throughout schedule construction. Second, FIFO, MOPNR, or MWKR selects among the next unscheduled operations of the jobs. FIFO prioritizes the earliest job-ready time, MOPNR the largest number of remaining operations, and MWKR the largest remaining workload computed using the fixed assignments. The selected operation starts at the maximum of its job-ready time and the availability time of its assigned machine. Equal-priority choices and equal-time machine assignments are resolved using seeded random tie-breaking. For each instance, we average five runs without best-of-K selection. This produces the Fixed-SPT-FIFO, Fixed-SPT-MOPNR, and Fixed-SPT-MWKR baselines.

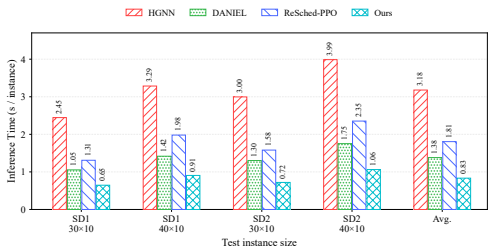
Decoding and computational budgets. Greedy decoding selects the feasible action with the highest score at every decision step. Sampling-100 generates 100 complete trajectories per instance and returns the schedule with the smallest makespan. PDScheduler computes persistent guidance from



(a) SD1 training sizes (ms/instance).



(b) SD2 training sizes (ms/instance).



(c) Larger instances from 20×10 checkpoints (s/instance).

Figure 4: Greedy inference time per instance across evaluation sets. (a,b) Training sizes of SD1 and SD2, 500 instances per size, with checkpoints matching each setting. (c) 30×10 and 40×10 instances, 100 per setting, using 20×10 checkpoints from the corresponding distribution. *Avg.* is weighted by instance count: 2,000 instances in (a,b) and 400 in (c).

the fully masked initial graph and reuses it throughout schedule construction. Reported inference times are in seconds per instance, and sampling times include all sampled trajectories. FFSP uses a separate budget of 24 trajectories, as specified below.

A.6 Additional FJSP evaluation

Additional inference-time results. Figures 4a–4c supplement Figure 3 with greedy inference times on the synthetic training sizes (500 instances per size) and the large-instance generalization sets (100 instances per setting). All measurements are amortized wall-clock time per instance on the RTX 5090 setup described above, excluding model loading, data parsing, and warmup; the *Avg.* bar is weighted by instance count. PDScheduler has the lowest average time in every figure, and the advantage grows with problem size: it is the fastest method in all four large-instance settings, averaging 0.83 s per instance versus 1.38 s for the second-best method ($1.7\times$), and leads at 20×10 on both distributions, while at the smallest sizes all methods run within a few milliseconds per instance and the ordering is mixed, as fixed per-instance overhead dominates.

Sensitivity to the refinement budget. For $K > 1$, the persistent heatmap is refined with K denoising network evaluations following the masked reverse process of Section 4.1, starting from the fully masked input; $K{=}1$ is the single-step default used in all other experiments. All runs use greedy decoding on the same 100 instances per setting and the same checkpoints as Table 1(a).

Table 4 shows that enlarging the refinement budget does not improve schedule quality: $K{=}1$ attains the best mean makespan in seven of the eight settings; the sole exception (SD2 20×10) improves by only 0.17% while costing 20% more inference time. The mean makespan change across settings stays within $\pm0.18\%$ for every K, whereas the mean inference time grows from 22.8 ms at $K{=}1$ to 30.5 ms at $K{=}8$ ($+34\%$). Unlike diffusion-based solvers that decode the final denoised sample directly as a solution, PDScheduler uses the denoiser only as a persistent global prior and delegates state-dependent decisions to the residual actor, so little quality remains to be recovered by additional refinement. Moreover, the denoiser checkpoint is selected by assignment accuracy on fully masked inputs (Appendix A.5), i.e., training and model selection are explicitly optimized for the single-step operating point. We therefore use $K{=}1$ throughout.

Table 4: Sensitivity to the persistent refinement budget K (denoising network evaluations per instance). $C_{\max}$ is the mean makespan and Time is the greedy inference time in milliseconds per instance. All entries use the same 100 instances per setting and the same checkpoints as Table 1(a). Bold marks the best makespan in each row. Lower is better.

	K=1		K=2		K=4		K=8	
Setting	$C_{\max}$ (↓)	Time (↓)	$C_{\max}$ (↓)	Time (↓)	$C_{\max}$ (↓)	Time (↓)	$C_{\max}$ (↓)	Time (↓)
SD1 10 × 5	**101.218**	8.97	101.348	9.43	101.360	11.78	101.526	15.92
SD1 20 × 5	**182.390**	16.42	182.456	17.01	182.536	21.47	182.494	23.80
SD1 15 × 10	**141.548**	24.48	141.796	25.45	141.720	27.81	142.060	32.22
SD1 20 × 10	**181.958**	38.52	181.980	39.65	182.100	42.08	182.108	47.09
SD2 10 × 5	**78.108**	8.71	78.328	9.26	78.228	11.54	78.204	16.38
SD2 20 × 5	**135.712**	17.20	135.916	18.05	135.798	20.35	135.768	24.72
SD2 15 × 10	**94.174**	27.65	94.400	28.63	94.316	30.82	94.202	35.22
SD2 20 × 10	113.258	40.74	113.428	41.72	113.064	44.10	**113.000**	48.87
Mean time (ms)		22.84		23.65		26.24		30.53

Table 5: Greedy gap (%) of PDScheduler trained on SD1 20×10 across three training seeds, evaluated on the in-distribution test set, the scale-generalization sets, and the open FJSP benchmarks. Seed 42 is the checkpoint reported in Table 1. The RESCHED column repeats the re-evaluated results from Table 1 as a reference. Lower is better.

Dataset	RESCHED	Seed 42	Seed 43	Seed 44	Mean ± std
In-distribution					
SD1 20 × 10	0.51	$\mathbf{-1.238}$	−0.875	−1.010	-1.041 ± 0.183
Scale generalization					
SD1 30 × 10	1.79	**1.044**	1.613	1.254	1.304 ± 0.288
SD1 40 × 10	1.44	**0.964**	1.922	1.272	1.386 ± 0.489
Avg.	1.62	**1.004**	1.768	1.263	1.345 ± 0.388
Open benchmarks					
Brandimarte	10.54	15.614	14.930	**13.112**	14.552 ± 1.293
Hurink (edata)	16.12	**13.729**	14.514	14.790	14.344 ± 0.550
Hurink (rdata)	11.57	12.419	**12.212**	12.564	12.398 ± 0.177
Hurink (vdata)	4.71	7.646	6.892	**5.123**	6.554 ± 1.295
Avg.	10.78	11.60	11.49	**11.00**	11.36 ± 0.32

Values are mean per-instance relative gaps (%); lower is better. In-distribution and scale-generalization gaps use OR-Tools 1800-s references, and open-benchmark gaps use the reference upper bounds, as in Table 1. The Avg. rows are weighted by instance count (200 for scale generalization, 130 for the open benchmarks), matching the aggregation in Table 1. Mean ± std summarizes the three training runs (sample standard deviation). The RESCHED column is the single re-evaluated checkpoint from Table 1 and is shown for reference; bold marks the lowest gap among the three seeds in each row.

Training-seed robustness. We train three independent runs on SD1 20×10 with training seeds 42, 43, and 44, which share the same pretrained denoiser initialization, and evaluate each greedy checkpoint on the in-distribution test set, the two scale-generalization sets, and the four open benchmark suites. Seed 42 is the checkpoint reported in Table 1.

Table 5 shows that the main conclusions are not seed-specific: the mean greedy gap across seeds remains below RESCHED both in-distribution (-1.04% vs. 0.51%) and on the scale-generalization sets (1.35% vs. 1.62%), and stays within 0.6 percentage points of the strongest baseline on the open benchmarks (11.36% vs. 10.78%). Individual seeds vary—most visibly on SD1 40×10, where one seed exceeds RESCHED—but the aggregate rankings behind all main conclusions are unchanged.

Table 6: Generalization performance on the Taillard JSSP benchmark. All results are reported as the instance-wise average gap (%) with respect to the best-known upper bounds (UB). Our model is trained only on synthetic 10×10 JSSP instances and directly evaluated on all benchmark sizes without retraining. Lower is better.

Size	PDRs				DRL-based				OR-Tools	UB
	SPT	MWKR	FDD/MWKR	MOPNR	L2D	RL-GNN	RESCHED	Ours		
15×15	54.8	56.7	47.1	45.0	26.0	20.1	**14.5**	17.1	0.1	1233.9
20×15	65.2	60.7	50.6	47.7	30.0	24.9	**18.8**	18.9	0.2	1361.3
20×20	64.2	55.7	47.6	42.8	31.6	29.2	**16.0**	24.3	0.7	1617.1
30×15	61.6	52.6	45.0	45.6	33.0	24.7	22.3	**22.2**	2.1	1771.2
30×20	66.0	63.9	56.3	48.2	33.6	32.0	**21.8**	31.1	2.8	1919.4
50×15	51.4	40.9	34.8	30.1	22.4	15.9	16.0	**15.9**	0.0	2783.8
50×20	59.5	53.9	41.5	37.9	26.5	21.3	**15.7**	22.0	2.8	2834.4
100×20	41.0	32.9	23.4	20.2	13.6	9.2	**9.2**	13.8	3.9	5369.6
Avg. Gap	57.96	52.16	43.29	39.69	27.09	22.16	**16.78**	20.65	1.58	–

All baseline results are re-evaluated on our own hardware under the same evaluation pipeline as our method, using their official implementations and following the protocols established in prior work. FDD/MWKR is a composite dispatching rule that selects the operation minimizing the ratio of its flow due date to its remaining work. UB denotes the average best-known solution value of the instances in each size group. Bold marks the best gap among the DRL-based methods in each row.

Table 7: Generalization performance on the DMU JSSP benchmark. All results are reported as the instance-wise average gap (%) with respect to the best-known upper bounds (UB). The same model trained only on synthetic 10×10 JSSP instances is directly applied to all test sizes. Lower is better.

Size	PDRs				DRL-based			UB
	SPT	MWKR	FDD/MWKR	MOPNR	L2D	RESCHED	Ours	
20×15	64.1	62.1	53.6	49.2	39.0	**23.068**	23.717	3023.8
20×20	64.6	58.2	52.5	45.2	37.7	**20.664**	26.576	3472.6
30×15	62.6	60.9	54.1	47.1	41.9	**27.400**	27.809	3879.0
30×20	65.9	63.2	60.1	52.0	39.5	**29.623**	31.060	4248.4
40×15	55.9	52.9	51.4	44.7	35.4	26.239	**25.670**	4871.2
40×20	63.0	61.1	55.5	49.2	39.4	**29.851**	30.695	5240.9
50×15	50.4	48.9	52.6	40.8	36.2	26.1	**26.0**	5950.6
50×20	62.2	56.4	57.3	49.6	38.8	31.823	**29.044**	6227.3
Avg. Gap	61.09	57.97	54.63	47.22	38.49	**26.85**	27.58	–

All baseline results are re-evaluated using their official implementations on our own hardware under the same evaluation pipeline as PDScheduler, following the protocols established in prior work. UB denotes the average best-known solution value of the instances in each size group. Bold marks the best gap among the DRL-based methods in each row.

A.7 Additional JSSP evaluation

Protocol. For JSSP, we use an actor-only variant of PDScheduler. Each operation has exactly one eligible machine, so its machine assignment is fixed. The assignment-learning stage is omitted: no denoiser pretraining is performed, and only the actor parameters are optimized. The actor is trained on synthetic 10×10 JSSP instances and evaluated directly on the Taillard (Taillard, 1993) and DMU (Demirkol et al., 1998) benchmarks without further training. The evaluation covers eight size groups in each suite, reaching 100×20 on Taillard and 50×20 on DMU. Tables 6 and 7 report greedy gaps to the instance-specific best-known upper bounds. All reported baseline results, including the PDRs, L2D (Zhang et al., 2020), RL-GNN (Park et al., 2021), RESCHED (Xiao et al., 2026), and OR-Tools, are re-evaluated using their official implementations on our own hardware under the same evaluation pipeline as PDScheduler, following the protocols established in prior work. OR-Tools uses an 1800-second budget per instance.

Results. PDScheduler achieves average gaps of 20.65% on Taillard and 27.58% on DMU, compared with 16.78% and 26.85% for RESCHED. It attains the lowest neural gap in five of the 16 size groups and ranks second among the reported neural methods in the aggregate results of both suites. These results show transfer beyond the training size, with performance varying across size groups. OR-Tools results on Taillard are reported separately from the neural-method ranking.

Table 8: Cross-size generalization performance on FFSP. RESCHED20 and Ours20 are trained only on FFSP20 and directly evaluated on FFSP50 and FFSP100 without retraining. MatNet20/50/100 denote models trained on the corresponding problem sizes. Sampling uses the best solution among 24 stochastic trajectories. $C_{\max}$ denotes the average makespan, and Time is the inference time per instance. Lower is better.

Method	Decode	FFSP20		FFSP50		FFSP100	
		$C_{\max}\downarrow$	Time (s) $\downarrow$	$C_{\max}\downarrow$	Time (s) $\downarrow$	$C_{\max}\downarrow$	Time (s) $\downarrow$
MatNet20	G	31.164	0.089	55.599	0.168	96.146	0.300
	S24	27.264	0.235	52.346	0.384	93.358	**0.640**
MatNet50	G	31.085	0.086	55.243	0.163	95.334	0.294
	S24	27.052	0.240	51.546	0.389	91.911	0.650
MatNet100	G	31.110	0.090	54.855	0.168	94.553	0.301
	S24	27.106	0.242	51.397	0.384	91.518	0.643
RESCHED20	G	26.587	0.060	**51.171**	0.188	**92.392**	0.445
	S24	**25.149**	0.102	**49.686**	0.429	**90.877**	1.753
Ours20	G	**26.453**	**0.037**	51.815	**0.084**	95.230	**0.197**
	S24	25.480	**0.064**	50.856	**0.282**	95.008	1.330

G denotes greedy decoding and S24 denotes Sampling-24, i.e., the best schedule among 24 stochastic trajectories. For MatNet, the greedy result corresponds to the first-permutation (G1) evaluation in our budget-controlled re-evaluation; no augmentation is used. Bold marks the best value in each column within each decoding strategy.

A.8 Additional FFSP evaluation

Protocol. The FFSP instances have three stages with four parallel machines per stage. Machine-specific processing times are sampled uniformly from the integers in $[2, 9]$. PDScheduler and RESCHED are trained only on FFSP20 and evaluated on FFSP20, FFSP50, and FFSP100 without retraining, where the suffix denotes the number of jobs. We compare them with MatNet checkpoints trained at each of these three sizes. Greedy evaluation uses one trajectory, and Sampling-24 returns the best schedule among 24 stochastic trajectories. For MatNet, the greedy result uses the first-permutation (G1) evaluation without augmentation. Table 8 reports mean makespan and inference time per instance for each decoding strategy.

Results. On FFSP20, PDScheduler attains the lowest greedy mean makespan (26.453) and the shortest greedy inference time (0.037 s). Its greedy inference times on FFSP50 and FFSP100 are 0.084 s and 0.197 s, respectively, compared with 0.188 s and 0.445 s for RESCHED. RESCHED obtains lower makespans at both larger sizes and under Sampling-24 at all three sizes. PDScheduler has the shortest Sampling-24 inference time on FFSP20 and FFSP50, whereas MatNet20 is fastest on FFSP100. Thus, the FFSP results show a trade-off between cross-size solution quality and inference cost.