SCHEDGPT: TOWARDS THE FIRST AUTO-REGRESSIVE FOUNDATION MODEL FOR LIST SCHEDULING

Anonymous authors
Paper under double-blind review

ABSTRACT

List scheduling underlies many combinatorial problems in manufacturing and distributed computing, yet existing learning-based schedulers are typically specialized to one problem family and one architecture. We introduce SchedGPT, an autoregressive foundation model with a graph encoder that handles diverse list-scheduling variants within a single architecture. The key insight is a unified heterogeneous graph representation over task and processor nodes, connected by precedence and eligibility relations, plus a unified priority-list encoding that specifies task ordering and per-task processor assignment and is deterministically decoded into a schedule. SchedGPT combines a relation-aware GATv2 encoder with a nanoGPT autoregressive decoder, factorizing each construction step into task sequencing and nested processor assignment via two pointer heads. Trained from scratch with a rollout-baseline policy-gradient objective, SchedGPT outperforms specialized state-of-the-art algorithms on manufacturing scheduling (JSP and FJSP) and workflow scheduling (five variants), demonstrating that list-scheduling problems can be unified under one architecture.

1 INTRODUCTION

List scheduling is a fundamental abstraction for allocating resources to precedence-constrained tasks and underlies a wide range of combinatorial problems in manufacturing and distributed computing (Graham, 1966). Despite this shared structure, existing learning-based schedulers are typically specialized to a single problem family, each relying on a dedicated architecture (Smit et al., 2025). This fragmentation obscures their shared combinatorial core and prevents models from reusing experience across domains.

Motivated by this gap, we introduce SchedGPT, the first autoregressive foundation model with a graph encoder for list scheduling that handles diverse problem instances within a single architecture. The core difficulty is that list scheduling problems differ in constraints, objectives, and solution spaces, involving heterogeneous nodes and relations, joint task ordering and processor assignment, and hard feasibility constraints, which precludes direct adaptation of routing foundation models such as RouteFinder (Berto et al., 2025). We show this fragmentation is not inherent: list scheduling problems can be cast as a single heterogeneous graph over task and processor nodes with precedence and eligibility relations. Solutions are encoded as a unified priority list (Shutler, 2003) specifying task ordering and per-task processor assignment, deterministically decoded into a schedule. This recasts list scheduling as sequence generation, enabling one autoregressive model to serve as a general-purpose policy.

SchedGPT uses a static encoder and dynamic decoder. A relation-aware GATv2 encoder (Brody et al., 2022) embeds the heterogeneous graph once, and an autoregressive decoder built on a nanoGPT backbone (Karpathy, 2022) builds the schedule through masked task and processor selections, factorizing each step into task sequencing and nested processor assignment via two pointer heads. We train it purely from reward feedback to minimize makespan, without optimal-solution labels or imitation, using REINFORCE (Williams, 1992). To reduce gradient variance during training without a learned critic, we use a frozen copy of the current policy that greedily decodes each instance, and the makespan of the resulting schedule serves as the baseline (Kool et al., 2019). This frozen copy is periodically refreshed as the policy improves. Because makespans and trajectory lengths vary by orders of magnitude across heterogeneous scheduling problems, a naive shared objective would be

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.

dominated by large instances and long horizons. We therefore adopt two scale-invariant designs, i.e., a relative advantage that measures each trajectory's improvement over the baseline as a fraction of the baseline cost, and per-trajectory normalization of log-probability and entropy by the number of tasks. Together with entropy regularization and temperature annealing, these choices allow us to train the same architecture on heterogeneous list scheduling instances separately, without per-problem reward shaping.

We evaluate SchedGPT on two broad categories of list scheduling problems: manufacturing scheduling and workflow scheduling. The former includes the classical JSP (Xiong et al., 2022) and FJSP (Dauzère-Pérès et al., 2024), while the latter comprises five problem variants (Katz et al., 2005; Deelman et al., 2006; Juve et al., 2009; Livny et al., 2008; Fajardo et al., 2021) that differ in constraints and resource characteristics. Despite using a single architecture and a shared training recipe, SchedGPT consistently outperforms specialized state-of-the-art algorithms that are designed and tuned for each problem individually. These results demonstrate that heterogeneous list scheduling problems, which have traditionally been addressed in isolation, can be effectively unified under one autoregressive foundation model. More broadly, they suggest a path toward general-purpose learned schedulers that transfer across domains without problem-specific architectural engineering.

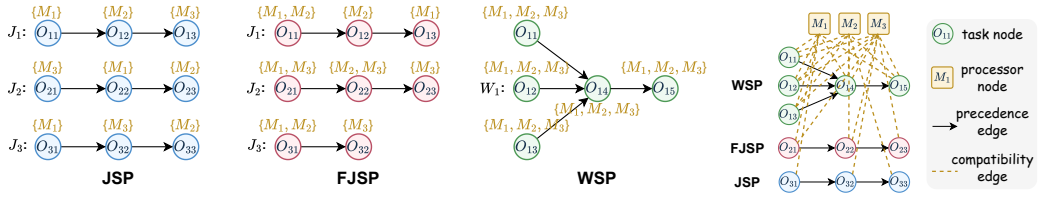
2 Related Work

Learning-based manufacturing schedulers typically use problem-specific representations and decision processes. L2D (Zhang et al., 2020) learns dispatching rules from JSP disjunctive graphs, while GNN-based improvement heuristics (Zhang et al., 2024) iteratively refine complete schedules. For FJSP, HGNN (Song et al., 2023) models operation–machine compatibility through heterogeneous graphs, and MIStar (Wang et al., 2026) combines heterogeneous graph representations with memory-enhanced improvement search. RESCHED (Xiao et al., 2026) uses simplified states and a Transformer architecture, extending from FJSP to JSP and FFSP. Li et al. (Li et al., 2025) address long-horizon FJSP through learning-guided rolling horizon optimization. Choi et al. (Choi et al., 2025) develop self-evolution-based multi-policy learning and evaluate it across six job-scheduling variants. These studies advance scheduling representations and learning strategies, but do not establish a common task–processor representation spanning manufacturing and workflow scheduling. For static workflow scheduling, Jeon et al. (Jeon et al., 2023) generate node priorities in one network pass and construct schedules through list scheduling. Qi et al. (Qi et al., 2025) improve one-shot priority learning through comparable-antichain identification and dense node-level rewards. We include both as makespan-oriented WSP baselines. In contrast, GOODRL (Yang et al., 2025) combines graph representations with offline-online learning to minimize mean flowtime under dynamic workflow arrivals. DEFT (Shen et al., 2026) uses deadline-aware mixtures of experts to minimize VM rental costs and deadline penalties for dynamically arriving workflows. We do not include these dynamic schedulers as direct baselines because their objectives and online decision protocols differ from our static makespan setting. Attention-based combinatorial models (Kool et al., 2019; Kwon et al., 2020) provide autoregressive construction strategies, while RouteFinder (Berto et al., 2025) unifies 48 VRP variants through shared representations and attribute-based encoding. Applying such models to list scheduling additionally requires representing precedence, processor eligibility, and coupled ordering and assignment decisions. SchedGPT addresses these requirements through a unified heterogeneous graph and priority-list representation, with autoregressive task selection and conditional processor assignment across JSP, FJSP, and WSP.

3 Preliminary

We first briefly introduce three representative scheduling problems (see Figure 1(a)) using their native terminology, and then define the unified terminology for our SchedGPT.

Job Shop Scheduling (JSP). A JSP instance consists of a set $\mathcal{J}$ of *jobs* and a set $\mathcal{M}$ of *machines* (Zhang et al., 2020). Each job $J_i \in \mathcal{J}$ contains an ordered sequence of *operations* $\{O_{i1} \to \cdots \to O_{i\ell_i}\}$. Each operation O_{ij} runs on a *designated* machine $M_k \in \mathcal{M}$ with a fixed processing time pt_{ij}. An operation can start only after its predecessor is completed. Each machine processes at most one operation at a time, and preemption is not allowed. The objective is to determine start times $\{st_{ij}\}$ that minimize the *makespan* $C_{\max} = \max_{i,j}\{ft_{ij} = st_{ij} + pt_{ij}\}$, subject to *precedence* constraints.



(a) Instances of three representative scheduling problems. (b) Unifed heterogeneous graph.

Figure 1: Representative problems and their unified heterogeneous graph representation.

Flexible Job Shop Scheduling (FJSP). FJSP extends JSP by allowing each operation O_{ij} to be processed on any machine in a *compatible* set $\mathcal{M}_{ij} \subseteq \mathcal{M}$ (Song et al., 2023). Its processing time pt_{ij}^k depends on the selected machine $M_k \in \mathcal{M}_{ij}$. Each operation must be processed non-preemptively on exactly one compatible machine. A complete schedule/solution requires all start times $\{st_{ij}^k\}$ of selected operation-machine pairs, under the operation *precedence* and machine *compatibility* constraints. The objective remains to minimize $C_{\max} = \max_{i,j,k}\{ft_{ij}^k = st_{ij}^k + pt_{ij}^k\}$.

Workflow Scheduling (WSP). A workflow consists of a set of *tasks* $\mathcal{O}$ and a set of *processors* $\mathcal{M}$ (Qi et al., 2025). These tasks are connected by directed edges that represent *precedence* constraints, forming a directed acyclic graph (DAG), i.e., a task may have multiple predecessors and multiple successors. A task can start only after all of its predecessors have been completed. Each task $O_{ij} \in \mathcal{O}$ can be assigned to any processor $M_k \in \mathcal{M}$. Given the workload tw_{ij} of task O_{ij} and the processing speed ms_k of assigned processor M_k, its processing time is $pt_{ij}^k = tw_{ij}/ms_k$. A schedule/solution determines the *assigned processor* and *start time* of each task, with the objective of minimizing the makespan $C_{\max} = \max_{i,j,k}\{ft_{ij}^k = st_{ij}^k + pt_{ij}^k\}$.

Common List Scheduling Formulation. To describe these scheduling problems within SchedGPT, we use ***task*** for the object being assigned and ***processor*** for a processing resource (Topcuoglu et al., 2002). Let $O_{ij} \in \mathcal{O}$ denote the j-th task in job J_i or workflow W_i, and $M_k \in \mathcal{M}$ denotes the k-th processor. The objective is to construct a schedule/solution (i.e., determining the assigned processor M_k and start time for each task st_{ij}^k) that minimizes the *makespan* $C_{\max}$, while satisfying the *precedence* and *compatibility* constraints. Solution construction in SchedGPT is separated into: (i) ***task prioritizing***, which chooses the next ready task with the highest priority; and (ii) ***processor selection***, which chooses a suitable compatible processor for that task. Full mathematical formulations are provided in Appendix A.

4 Proposed Method

Figure 2 illustrates the overall framework of **SchedGPT**, a GPT-style foundation model that learns to autoregressively construct schedules/solutions through *task prioritizing* and *processor selection*. **First**, we represent different scheduling instances as *unified heterogeneous graphs* $\mathcal{G}$ and their solutions as *two aligned lists* (i.e., task list $\mathbf{L}_t^O$ and processor list $\mathbf{L}_t^M$). **Second**, the foundation model architecture consists of the GATv2-based encoder and the nanoGPT-based decoder. The *encoder* extracts task and processor embeddings from graph $\mathcal{G}$ once, which are then reused throughout autoregressive decoding. The causal *decoder* receives input tokens (i.e., the encoded task and processor embeddings, the aligned lists from the previous step, and dynamic features at step t), and provides the selected task $O^{(t)}$ and processor $M^{(t)}$. The two aligned lists $(\mathbf{L}_t^O, \mathbf{L}_t^M)$ are constructed autoregressively by the decoder until $t = |\mathcal{O}|$. **Finally**, the schedule builder maps the complete two lists into a feasible schedule, whose *makespan* provides the training signal for the encoder and decoder.

4.1 Unified Instance and Solution Representations

Instance Representation. For an instance $\mathcal{I}$ consists of a set $\mathcal{O}$ of *tasks* and a set $\mathcal{M}$ of *processors*, let $n = |\mathcal{O}|$ and $m = |\mathcal{M}|$. As shown in Figure 1(b), we construct a heterogeneous graph

$$\mathcal{G} = (\mathcal{O} \cup \mathcal{M}, \mathcal{E}_{\text{pre}} \cup \mathcal{E}_{\text{comp}}), \tag{1}$$

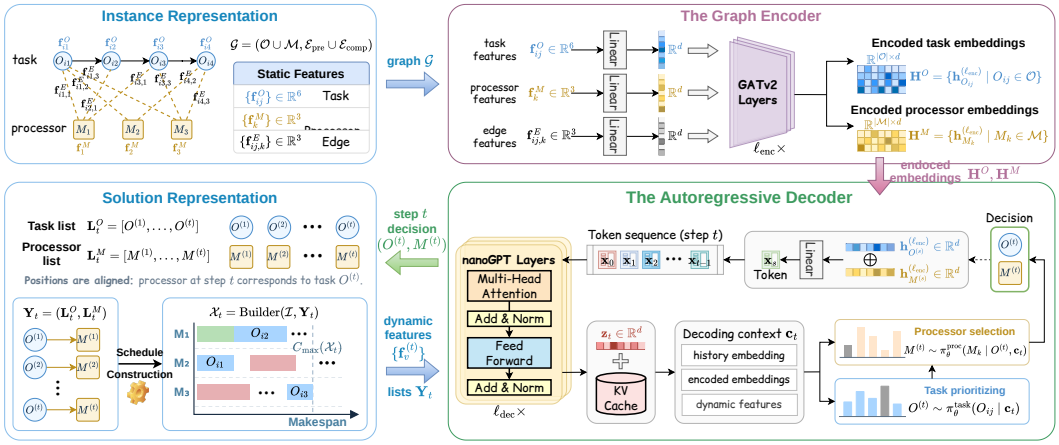


Figure 2: The overall framework of SchedGPT.

where $(O_{ij}, O_{pq}) \in \mathcal{E}_{\text{pre}}$ denotes a *precedence* constraint, and $\{(O_{ij}, M_k), (M_k, O_{ij})\} \in \mathcal{E}_{\text{comp}}$ represents a bidirectional *compatibility* relation whenever $M_k \in \mathcal{M}_{ij}$. This graph represents different problem instances using a unified representation. For example, JSP and FJSP contain job-specific precedence chains, whereas WSP allows general DAG dependencies. Each task O_{ij} connects to one designated processor $M_k \in \mathcal{M}$ in JSP, a compatible subset $\mathcal{M}_{ij} \subseteq \mathcal{M}$ in FJSP, and all processors $\mathcal{M}$ in WSP, as formulated in Section 3.

Each task O_{ij} is represented by a feature vector $\mathbf{f}_{ij}^O \in \mathbb{R}^6$, encoding its processing requirements, precedence-related information, and processor compatibility. Each processor M_k has a feature vector $\mathbf{f}_k^M \in \mathbb{R}^3$ summarizing its compatible tasks and their processing times. Each compatibility edge is associated with $\mathbf{f}_{ij,k}^E \in \mathbb{R}^3$ capturing three processing-time features. Full feature definitions are provided in Appendix B.

Solution Representation. At construction step t, let $O^{(t)} = O_{i_t j_t}$ and $M^{(t)} = M_{k_t}$ denote the selected task and processor. Their decisions extend two aligned lists:

$$\mathbf{L}_t^O = [O^{(1)}, \ldots, O^{(t)}], \qquad \mathbf{L}_t^M = [M^{(1)}, \ldots, M^{(t)}]. \tag{2}$$

$$\mathbf{Y}_t = (\mathbf{L}_t^O, \mathbf{L}_t^M) = \{(O^{(s)}, M^{(s)})\}_{s=1}^t, \tag{3}$$

The two lists are positionally aligned, i.e., $M^{(t)}$ is the processor assigned to $O^{(t)}$, to specify task–processor assignments. Every task appears *once* in the complete task list $\mathbf{L}_n^O$, whereas a processor may appear *multiple times* in the complete processor list $\mathbf{L}_n^M$. The task list records the construction priority, not necessarily the chronological order of task start times in the schedule.

From Lists to Schedule. The partial task and processor lists determine a partial schedule through

$$\mathcal{X}_t = \text{Builder}(\mathcal{I}, \mathbf{Y}_t), \tag{4}$$

where $\mathcal{X}_t$ denotes the partial schedule generated for instance $\mathcal{I}$ at step t. The schedule builder processes task–processor assignments in order and computes their start and finish times, subject to precedence and compatibility constraints. Replaying the lists $\mathbf{Y}_t$ with the same builder recovers the same schedule $\mathcal{X}_t$ and its objective $C_{\max}(\mathcal{X}_t)$. Detailed descriptions are provided in Appendix C.

4.2 The SchedGPT Foundation Model

4.2.1 The Graph Encoder

As shown in Figure 2, separate linear projections first embed task, processor, and edge features (i.e., $\mathbf{f}_{ij}^O \in \mathbb{R}^6, \mathbf{f}_k^M \in \mathbb{R}^3$, and $\mathbf{f}_{ij,k}^E \in \mathbb{R}^3$) into a common hidden space with dimension as d. For message passing, we consider three relation types: precedence edges point from predecessors to successors, task self-loops preserve local information, and each compatibility edge is represented in both directions. After ℓ_{enc} layers of GATv2 (Brody et al., 2022) networks, we obtain the encoded task and processor embeddings

$$\mathbf{H}^O = \{\mathbf{h}_{O_{ij}}^{(\ell_{\text{enc}})} \mid O_{ij} \in \mathcal{O}\}, \qquad \mathbf{H}^M = \{\mathbf{h}_{M_k}^{(\ell_{\text{enc}})} \mid M_k \in \mathcal{M}\}. \tag{5}$$

The static embeddings $\mathbf{H}^O \in \mathbb{R}^{n\times d}$ and $\mathbf{H}^M \in \mathbb{R}^{m\times d}$ are produced once per instance. The decoder reuses them throughout a rollout. Subsequent changes of task-processor assignments in the evolving partial schedule are maintained separately through decoder-used dynamic features.

4.2.2 THE AUTOREGRESSIVE DECODER

The decoder uses nanoGPT[1] (Karpathy, 2022) as the backbone to autoregressively construct the aligned lists $\mathbf{Y}_t = (\mathbf{L}_t^O, \mathbf{L}_t^M)$ until $t = |\mathcal{O}|$. Within the decoder, two pointer heads are used to perform *task prioritizing* and *processor selection*, respectively.

List-conditioned Decoding. At step t, the decoder conditions on the instance embeddings $(\mathbf{H}^O, \mathbf{H}^M)$, the last-step lists $\mathbf{Y}_{t-1}$, and the dynamic features $\{\mathbf{f}_v^{(t)}\}_{v\in\mathcal{O}\cup\mathcal{M}}$. Specifically, each previous task–processor assignment $(O^{(s)}, M^{(s)})$, $s < t$, is represented by projecting the concatenation of their encoded embeddings:

$$\mathbf{x}_s = \mathrm{Linear}_\theta\Big([\mathbf{h}_{O^{(s)}}^{(\ell_{\mathrm{enc}})} \| \mathbf{h}_{M^{(s)}}^{(\ell_{\mathrm{enc}})}]\Big). \tag{6}$$

Together with an initial token $\mathbf{x}_0$, the aligned lists $\mathbf{Y}_{t-1} = (\mathbf{L}_{t-1}^O, \mathbf{L}_{t-1}^M)$ are therefore represented as a sequence $[\mathbf{x}_0, \ldots, \mathbf{x}_{t-1}]$ of task–processor *tokens*. Then, the *token sequence* is processed by ℓ_{dec} causal Transformer layers to obtain a history embedding $\mathbf{z}_t$:

$$\mathbf{z}_t = [\mathrm{GPT}_\theta([\mathbf{x}_0, \ldots, \mathbf{x}_{t-1}])]_{t-1}, \tag{7}$$

where $\mathbf{z}_t \in \mathbb{R}^d$ summarizes all task–processor assignments generated before step t. Moreover, a key-value (KV) cache is adopted to store previous self-attention keys and values for efficient autoregressive decoding. Subsequently, $\mathbf{z}_t$ is combined with $(\mathbf{H}^O, \mathbf{H}^M)$ and $\{\mathbf{f}_v^{(t)}\}_{v\in\mathcal{O}\cup\mathcal{M}}$ to form the decoding context $\mathbf{c}_t$, which is used to make the next task and processor decisions.

Task Prioritizing Decision. Given the history embedding $\mathbf{z}_t$, the task pointer first attends to the encoded task embeddings:

$$\mathbf{q}_{\bar{O}}^{(t)} = \mathrm{MHA}_\theta^O(\mathbf{z}_t, \mathbf{H}^O, \mathbf{H}^O), \tag{8}$$

where $\mathbf{q}_{\bar{O}}^{(t)} \in \mathbb{R}^d$ summarizes the task information at step t. For each task O_{ij}, its static encoded embedding is concatenated with the current dynamic features $\mathbf{f}_{O_{ij}}^{(t)}$, processed by

$$\mathbf{k}_{O_{ij}}^{(t)} = \mathrm{MLP}_\theta^O\left([\mathbf{h}_{O_{ij}}^{(\ell_{\mathrm{enc}})} \| \mathbf{f}_{O_{ij}}^{(t)}]\right), \tag{9}$$

where $\mathbf{f}_{O_{ij}}^{(t)} \in \mathbb{R}^7$ describes its current scheduling status (see Appendix B). The priority score of O_{ij} is then computed by

$$u_{O_{ij}}^{(t)} = \frac{(\mathbf{q}_{\bar{O}}^{(t)})^\top \mathbf{k}_{O_{ij}}^{(t)}}{\sqrt{d}}. \tag{10}$$

Let $\mathcal{A}_t^O \subseteq \mathcal{O}$ denote the set of ready tasks at step t. The task selection probability is

$$\pi_\theta^{\mathrm{task}}(O_{ij} \mid \mathbf{c}_t) = \frac{\exp(u_{O_{ij}}^{(t)})}{\sum_{O_{pq}\in\mathcal{A}_t^O} \exp(u_{O_{pq}}^{(t)})}, \quad O_{ij} \in \mathcal{A}_t^O, \tag{11}$$

and zero otherwise. Thus, the next task $O^{(t)}$ is selected according to $\pi_\theta^{\mathrm{task}}(\cdot \mid \mathbf{c}_t)$.

Processor Selection Decision. Given the selected task $O^{(t)} = O_{i_t j_t}$, the processor pointer conditions on both its encoded embedding and the history embedding $\mathbf{z}_t$. For each processor M_k, its static embedding is combined with the task-dependent dynamic features $\mathbf{f}_{M_k}^{(t)}(O^{(t)})$:

$$\mathbf{k}_{M_k}^{(t)} = \mathrm{MLP}_\theta^M\left([\mathbf{h}_{M_k}^{(\ell_{\mathrm{enc}})} \| \mathbf{f}_{M_k}^{(t)}(O^{(t)})]\right), \tag{12}$$

$$\mathbf{q}_{\bar{M}}^{(t)} = \mathrm{MHA}_\theta^M\left(\mathrm{Linear}_\theta^M[\mathbf{z}_t \| \mathbf{h}_{O^{(t)}}^{(\ell_{\mathrm{enc}})}], \mathbf{K}_{\bar{M}}^{(t)}, \mathbf{K}_{\bar{M}}^{(t)}\right), \tag{13}$$

[1] https://github.com/karpathy/nanoGPT

where $\mathbf{K}_M^{(t)} = \{\mathbf{k}_{M_k}^{(t)} \mid M_k \in \mathcal{M}\}$. The score of assigning $O^{(t)}$ to processor M_k is obtained by

$$u_{M_k}^{(t)} = \frac{(\mathbf{q}_M^{(t)})^\top \mathbf{k}_{M_k}^{(t)}}{\sqrt{d}}. \tag{14}$$

Then, the processor selection probability is

$$\pi_\theta^{\text{proc}}(M_k \mid O^{(t)}, \mathbf{c}_t) = \frac{\exp(u_{M_k}^{(t)})}{\sum_{M_p \in \mathcal{M}_{i_t j_t}} \exp(u_{M_p}^{(t)})}, \quad M_k \in \mathcal{M}_{i_t j_t}, \tag{15}$$

and zero incompatible processors. The processor $M^{(t)}$ is selected according to $\pi_\theta^{\text{proc}}(\cdot \mid O^{(t)}, \mathbf{c}_t)$. For JSP, $|\mathcal{M}_{i_t j_t}| = 1$, making this decision deterministic.

Autoregressive Construction. After selecting $(O^{(t)}, M^{(t)})$, the two aligned lists are extended as

$$\mathbf{L}_t^O = \mathbf{L}_{t-1}^O \oplus O^{(t)}, \qquad \mathbf{L}_t^M = \mathbf{L}_{t-1}^M \oplus M^{(t)}, \tag{16}$$

where $\oplus$ denotes the append operation. The selected task–processor pair forms the next decoder token $\mathbf{x}_t$ following Eq. (6), which becomes part of the subsequent GPT input in Eq. (7) to obtain $\mathbf{z}_{t+1}$. The above process in Eq. (3)–(16) is repeated until $t = n$. Finally, the complete lists $\mathbf{Y} = (\mathbf{L}_n^O, \mathbf{L}_n^M)$ are mapped to the complete schedule $\mathcal{X}$ according to Eq. (3)–(4).

4.3 Training SchedGPT

SchedGPT is trained with policy gradient directly from the makespan $C_{\max}(\mathcal{X})$ of the constructed complete schedule $\mathcal{X}$. Given $\mathbf{Y} = (\mathbf{L}_n^O, \mathbf{L}_n^M)$ and $\mathcal{X} = \text{Builder}(\mathcal{I}, \mathbf{Y})$, the learning objective is

$$\min_\theta \mathbb{E}_{\substack{\mathcal{I}\sim\mathcal{D} \\ \mathbf{Y}\sim\pi_\theta(\cdot|\mathcal{I})}} \left[C_{\max}(\mathcal{X})\right], \tag{17}$$

where $\mathcal{D}$ denotes the training-instance distribution.

Policy-Gradient Objective. For an instance $\mathcal{I}$, the probability of the complete task and processor lists follows the two-stage decisions in Eq. (11) and Eq. (15):

$$\pi_\theta(\mathbf{Y} \mid \mathcal{I}) = \prod_{t=1}^{n} \pi_\theta^{\text{task}}(O^{(t)} \mid \mathbf{c}_t)\, \pi_\theta^{\text{proc}}(M^{(t)} \mid O^{(t)}, \mathbf{c}_t), \tag{18}$$

$$\log \pi_\theta(\mathbf{Y} \mid \mathcal{I}) = \sum_{t=1}^{n} \left[\log \pi_\theta^{\text{task}}(O^{(t)} \mid \mathbf{c}_t) + \log \pi_\theta^{\text{proc}}(M^{(t)} \mid O^{(t)}, \mathbf{c}_t)\right]. \tag{19}$$

The entropy of the two-stage decisions is

$$\mathcal{H} = \sum_{t=1}^{n} \left[\mathcal{H}(\pi_\theta^{\text{task}}) + \mathcal{H}\left(\pi_\theta^{\text{proc}}\right)\right]. \tag{20}$$

To reduce the variance of policy-gradient learning, we adopt a greedy rollout baseline to compute advantages. For each training instance $\mathcal{I}$, in addition to the sampled complete lists, we also use a frozen baseline policy $\pi_{\bar{\theta}}$ to greedily construct the reference lists:

$$\mathbf{Y} \sim \pi_\theta(\cdot \mid \mathcal{I}), \qquad \bar{\mathbf{Y}} = \text{Greedy}(\pi_{\bar{\theta}}, \mathcal{I}). \tag{21}$$

The relative cost advantage is defined as

$$C = C_{\max}\left(\text{Builder}(\mathcal{I}, \mathbf{Y})\right), \quad \bar{C} = C_{\max}\left(\text{Builder}(\mathcal{I}, \bar{\mathbf{Y}})\right), \quad A = \frac{C - \bar{C}}{\bar{C}}. \tag{22}$$

We further normalize it within each minibatch to obtain $\hat{A}$. According to Eq. (19) and Eq. (20), the *training loss* corresponding to Eq. (17) is defined as

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{\mathcal{I}\in\mathcal{B}} \left[\hat{A} \log \pi_\theta(\mathbf{Y} \mid \mathcal{I}) - \beta\mathcal{H}\right], \tag{23}$$

where β is the entropy coefficient.

Training Procedure. During training, SchedGPT alternately samples batches from JSP, FJSP, and WSP while sharing the same model parameters θ across all problem families. The rollout baseline is periodically updated as the current policy improves. The pseudo-code for training SchedGPT and other details are provided in Appendix D.

Table 1: **SchedGPT across three DAG-scheduling domains.** Lower is better; best in bold, second best underlined. JSP/FJSP report makespan gap (%) to reference upper bound; Pegasus reports the mean per-instance makespan gap (%) to HEFT, per workflow, pooling instances with 100–400 tasks (<500; 80 per workflow) and with 1000 tasks (20 per workflow) (negative: better than HEFT). SchedGPT-S is domain-specific, and SchedGPT-U is one model jointly trained on FJSP, JSP, and WSP.

JSP (Taillard), gap to UB (%)

# Tasks	# Processors	L2D	ScheduleNet	ReSched	SchedGPT-S	SchedGPT-U
225	15	25.96	15.3	15.74	**13.81**	16.88
300	15	30.04	19.4	19.70	**15.41**	20.53
400	20	31.61	17.2	16.30	**15.90**	17.97
450	15	33.01	19.1	21.50	**17.40**	22.55
600	20	33.69	23.7	22.50	**19.63**	26.63
750	15	22.38	13.9	16.10	**12.44**	18.36
1000	20	26.51	**13.5**	15.60	14.09	18.38
2000	20	13.61	**6.7**	9.60	7.42	11.16
Average		27.10	16.1	17.13	**14.51**	19.06

FJSP (Brandimarte / Hurink, 4–15 processors), gap to UB (%), greedy

Benchmark	# Tasks	HGNN	DANIEL	ReSched	SchedGPT-S	SchedGPT-U
Brandimarte	55–284	26.77	12.97	9.08	**4.33**	11.68
Hurink-edata	50–300	15.00	14.41	15.48	**12.64**	15.12
Hurink-rdata	50–300	11.14	11.42	10.18	**8.61**	11.12
Hurink-vdata	50–300	4.02	3.28	**2.55**	2.57	3.99
Average		14.23	10.52	9.32	**7.04**	10.48

Pegasus (5 workflows, 4 processors), gap to HEFT (%)

Workflow (# Tasks)	FIFO-EFT	SPT-EFT	Qi2025	Jeon2023	SchedGPT-S	SchedGPT-U
SIPHT (<500)	2.90	40.80	8.51	0.95	**0.69**	4.91
SIPHT (1000)	**0.37**	35.71	3.53	0.89	0.69	2.48
LIGO (<500)	1.44	8.66	4.04	**0.13**	0.20	1.22
LIGO (1000)	0.22	3.90	1.22	**0.04**	0.41	0.93
GENOME (<500)	1.45	4.69	2.05	0.53	**0.25**	1.04
GENOME (1000)	**0.20**	1.28	0.48	0.21	0.55	0.62
CYBERSHAKE (<500)	0.16	13.78	0.77	−1.52	**−17.47**	1.89
CYBERSHAKE (1000)	**−0.99**	20.56	1.92	2.40	−0.40	1.99
MONTAGE (<500)	−0.05	5.56	0.61	−0.21	**−1.01**	0.65
MONTAGE (1000)	−0.01	4.31	0.24	**−0.05**	0.18	0.55
Average	0.93	14.39	2.85	0.12	**−2.72**	1.82

5 Experiments

5.1 Experimental Setup

Datasets. We evaluate SchedGPT on the three scheduling problems introduced in Section 3. For **JSP**, we follow L2D (Zhang et al., 2020) to evaluate on Taillard public JSP benchmarks. For **FJSP**, we use the *Brandimarte* and three *Hurink* benchmark sets (i.e., *edata*, *rdata*, and *vdata*), which cover different levels of task–processor flexibility (Song et al., 2023). For **WSP**, we evaluate on 5 Pegasus scientific workflow datasets (Deelman et al., 2015): *SIPHT*, *LIGO*, *Genome*, *CyberShake*, and *Montage*. Full benchmark statistics and generation settings are provided in Appendix E.

Baselines. We compare SchedGPT with representative learning-based schedulers and conventional heuristics. For JSP, we compare against L2D (Zhang et al., 2020), ScheduleNet (Park et al., 2021), and ReSched (Xiao et al., 2026). For FJSP, we compare against HGNN (Song et al., 2023), DANIEL (Wang et al., 2024), and ReSched (Xiao et al., 2026). For workflow scheduling on Pegasus, we compare against FIFO-EFT and SPT-EFT (Pinedo, 2016), as well as the learning-based schedulers

Qi2025 (Qi et al., 2025) and Jeon2023 (Jeon et al., 2023). Detailed implementations are provided in Appendix F. For all baselines and SchedGPT, we use greedy decoding for solution quality evaluation.

Metrics. We report the average *makespan* $\bar{C}_{\max} = \frac{1}{|\mathcal{D}|}\sum_{\mathcal{I}_i\in\mathcal{D}} C^i_{\max}$ as the primary metric. To compare instances with different cost scales, we additionally use the average relative Gap:

$$\mathrm{Gap}(\%) = \frac{100}{|\mathcal{D}|}\sum_{\mathcal{I}_i\in\mathcal{D}}\left(\frac{C^i_{\max}}{C^{i,\mathrm{ref}}_{\max}} - 1\right). \tag{24}$$

Lower values are better, where $\mathrm{Gap} < 0$ indicates an improvement over the reference.

Table 2: **Statistical significance on Pegasus.** Mean paired difference in per-instance gap to HEFT (percentage points) between SchedGPT-S and each baseline; negative values favour SchedGPT-S. Size columns pool the five workflows (100 paired instances each); "All" uses all 500. Two-sided Wilcoxon signed-rank tests, Holm-corrected over all 24 comparisons: $^{\dagger}p < 0.05$, $^{\dagger\dagger}p < 0.01$, $^{\dagger\dagger\dagger}p < 0.001$. W/T/L counts instances where SchedGPT-S is better/equal/worse.

SchedGPT-S vs. baseline (Pegasus, 4 processors)

	# Tasks						
Baseline	100	200	300	400	1000	All	W/T/L
FIFO-EFT	$-9.85^{\dagger\dagger\dagger}$	$-4.59^{\dagger\dagger\dagger}$	$-2.07^{\dagger\dagger\dagger}$	$-2.06^{\dagger\dagger\dagger}$	$+0.33^{\dagger\dagger\dagger}$	$-3.65^{\dagger\dagger\dagger}$	351/0/149
SPT-EFT	$-22.06^{\dagger\dagger\dagger}$	$-17.87^{\dagger\dagger\dagger}$	$-16.28^{\dagger\dagger\dagger}$	$-16.45^{\dagger\dagger\dagger}$	$-12.87^{\dagger\dagger\dagger}$	$-17.10^{\dagger\dagger\dagger}$	492/0/8
Qi2025	$-10.76^{\dagger\dagger\dagger}$	$-6.68^{\dagger\dagger\dagger}$	$-4.97^{\dagger\dagger\dagger}$	$-4.25^{\dagger\dagger\dagger}$	$-1.19^{\dagger\dagger\dagger}$	$-5.57^{\dagger\dagger\dagger}$	465/0/35
Jeon2023	$-6.01^{\dagger\dagger}$	$-3.31^{\dagger\dagger\dagger}$	$-2.54^{\dagger\dagger\dagger}$	$-1.92^{\dagger\dagger\dagger}$	-0.41	$-2.84^{\dagger\dagger\dagger}$	323/0/177

Training Configuration Details. Architecturally, SchedGPT couples a graph encoder of three GATv2 layers with a nanoGPT-based decoder with six causal Transformer layers, a hidden dimension of $d = 256$, and eight attention heads. For training, we adopt AdamW with a maximum learning rate of 1.5×10^{-4} and a weight decay of 10^{-4}, while annealing the entropy coefficient from 0.01 to 0.001. We investigate two training configurations. SchedGPT-S is trained separately for each problem family, mixing instances of varying sizes within that family. SchedGPT-U, by contrast, is trained jointly on FJSP, JSP, and WSP, with a single set of parameters shared across all three. Unless otherwise specified, we evaluate using greedy decoding. Further details are provided in Appendix D.

5.2 Performance across Variant List Scheduling Problems

Table 1 compares SchedGPT with problem-specific expert models across JSP, FJSP, and WSP. It is worth noting that each baseline is a specialized method designed and tuned for its own problem family, whereas SchedGPT uses one shared architecture across all three. The purpose of this experiment is not only to compare solution quality, but also to test whether the unified representation and autoregressive construction in Section 4 can be applied without changing the model architecture across structurally different scheduling problems.

Job Shop Scheduling. In JSP, each task has exactly one compatible processor, so processor selection in Eq. (15) becomes deterministic. Nevertheless, SchedGPT uses exactly the same task–processor representation and decoder as in the other problems. On the Taillard benchmark, SchedGPT-S attains an average gap of 14.51%, substantially outperforming the JSP expert models L2D (Zhang et al., 2020) (27.10%), ReSched (Xiao et al., 2026) (17.13%), and ScheduleNet (Park et al., 2021) (16.1%). This result is notable because it shows that even when processor selection is trivial, the unified interface does not hinder performance. The model matches or exceeds specialized JSP schedulers without any JSP-specific policy.

Flexible Job Shop Scheduling. FJSP additionally requires the model to select among multiple compatible processors for each task, so both terms of the two-stage policy contribute to solution construction. Under greedy decoding, SchedGPT-S achieves an average gap of 7.04%, clearly better than the FJSP expert models ReSched (Xiao et al., 2026) (9.32%), DANIEL (Wang et al., 2024) (10.52%), and HGNN (Song et al., 2023) (14.23%). Under sampling, it improves further to 3.71%, again the best among all methods. These results confirm that the same architecture handles both the deterministic processor choice of JSP and the flexible choice of FJSP without modification.

Workflow Scheduling. Workflow scheduling changes the precedence structure from job chains to general DAGs and introduces substantially different task and processor characteristics. On Pegasus, SchedGPT-S attains the best average gap to HEFT (-2.72%), ahead of the workflow expert models Jeon2023 (Jeon et al., 2023) (0.12%) and Qi2025 (Qi et al., 2025) (2.85%) as well as the heuristics FIFO-EFT (Pinedo, 2016) (0.93%) and SPT-EFT (Pinedo, 2016) (14.39%), corresponding to a mean makespan of 2137.15. This advantage is driven mainly by the larger-margin wins on CYBERSHAKE (< 500, -17.47%) and MONTAGE (< 500, -1.01%). On several individual workflows, however, specialized baselines remain stronger: Jeon2023 leads on LIGO at both scales, GENOME at 1000 tasks, and MONTAGE at 1000 tasks, and FIFO-EFT leads on SIPHT at 1000 tasks. The result is therefore not uniform dominance but a consistently strong average across structurally diverse workflows, which is notable because Pegasus instances differ fundamentally from job-shop instances in graph structure and resource model, yet the same architecture delivers state-of-the-art aggregate performance.

Joint Training and Generalization. Beyond domain-specific training, we examine whether one model can serve multiple scheduling families without per-domain specialization. SchedGPT-U is trained jointly on FJSP, JSP, and WSP with a single set of shared parameters. As shown in Table 1, it remains competitive with expert models across all three domains. On JSP, it reaches an average gap of 19.06%, outperforming the JSP expert L2D (Zhang et al., 2020) (27.10%) but trailing ScheduleNet (16.10%) and ReSched (Xiao et al., 2026) (17.13%). On FJSP, it attains 10.48% under greedy decoding, ahead of the FJSP expert HGNN (Song et al., 2023) (14.23%) and marginally ahead of DANIEL (10.52%), though behind ReSched (Xiao et al., 2026) (9.32%). On Pegasus, it achieves an average gap of 1.82% (mean makespan 2151.40), better than the workflow expert Qi2025 (Qi et al., 2025) (2.85%) and the heuristic SPT-EFT (14.39%), but behind FIFO-EFT (0.93%) and Jeon2023 (Jeon et al., 2023) (0.12%). These results reveal a clear trade-off: joint training incurs a modest performance drop relative to domain-specific training, and on some domains it falls short of the strongest specialized baselines, yet it still surpasses several expert models without any architecture change. This supports the central premise of our work, namely that heterogeneous scheduling problems share a common structure that a single autoregressive model can capture, and it highlights joint training as a promising direction for building general-purpose schedulers.

Overall, Table 1 shows that a single autoregressive architecture can match or surpass problem-specific expert models across all three problem families. The domain-specific SchedGPT-S is strongest throughout, and the jointly trained SchedGPT-U remains competitive even though it shares one set of parameters across structurally different problems. These findings justify the central claim of this work: the fragmentation of learning-based scheduling is not inherent to the problems themselves. A unified representation and a single autoregressive model can deliver strong performance across heterogeneous scheduling domains without any problem-specific architectural engineering.

Significant Test on Pegasus. We compare SchedGPT-S against each baseline using two-sided paired Wilcoxon signed-rank tests on per-instance gaps (relative to HEFT), with Holm correction across all 24 comparisons (significance at adjusted $p < 0.05$). On Pegasus, SchedGPT-S significantly outperforms all four baselines over the 500 test instances (Holm-corrected Wilcoxon, $p < 0.001$). This includes the learned schedulers Qi2025 (-5.57 pp; better on $465/500$ instances) and Jeon2023 (-2.84 pp; better on $323/500$ instances). The advantage is concentrated on instances with 100–400 tasks, where SchedGPT-S is significantly better than every baseline. At 1000 tasks, however, it is no longer significantly better than Jeon2023, and it is slightly but significantly worse than FIFO-EFT ($+0.33$ pp). Full results are reported in Table 2.

6 Conclusion

We introduced SchedGPT, the first autoregressive foundation model with a graph encoder for list scheduling. Its core contribution is a unified formulation in which a heterogeneous graph represents tasks and processors, and a priority list pairs task ordering with processor assignments, recasting list scheduling as sequence generation. SchedGPT couples a GATv2 encoder with a nanoGPT decoder and is trained from scratch via policy-gradient learning with a rollout baseline, without optimal labels. Across JSP, FJSP, and 20 workflow variants, it consistently outperforms specialized state-of-the-art algorithms within a single shared architecture. These results suggest that fragmentation in learning-based scheduling is not inherent, and that a common representation can yield a general-purpose scheduler without problem-specific engineering.

AI USE STATEMENT

In this work, we did not use generative AI tools for tasks with required disclosure, i.e., generation of research ideas, coding, experimental design, statistical analysis, or data analysis, and the rest of the required disclosure tasks are not applicable to this work. We used generative AI tools for improving readability, literature interpretation, and figure generation. We have reviewed all these AI-assisted works. Specifically, all language edits were reviewed and validated by human authors. Furthermore, all literature interpretations were human-verified against original publications, and all AI-generated figures were derived from data produced through human-designed experiments and were thoroughly checked for correctness. All experimental work and data organization were conducted entirely by human authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

All experiments in this work use fixed random seeds and a fixed evaluation program. The detailed configurations of the model architecture, training procedure, hyperparameters, and evaluation metrics are fully specified in the main text and the appendix. The appendix also reports all experimental runs and computing costs. The complete training code, pretrained model weights, and datasets will be released upon acceptance to facilitate reproduction of all reported results.

ETHICS STATEMENT

This work studies learning-based methods for combinatorial scheduling problems (JSP, FJSP, and DAG workflow scheduling). All experiments use publicly available benchmark instances (Taillard, Brandimarte, and Hurink) or synthetically generated workflows (Pegasus workflows from a public generator). No human subjects, personal data, or proprietary user data are involved. Despite its name, SchedGPT is a graph-based policy trained from scratch with reinforcement learning, not a language model: it uses no pretrained language models, text prompts, or web-scale data. External baseline results are either reproduced under the same protocol or taken from their original publications, with sources cited.

Improved schedulers can reduce makespan and resource usage in manufacturing and computing systems, potentially lowering energy consumption. We are not aware of direct harmful applications beyond those of general-purpose optimization tools. However, learned schedulers provide no optimality or worst-case guarantees, and our results show that their advantage can diminish on instances larger than those seen during training. Deployment in safety- or cost-critical settings should therefore include validation against established heuristics and appropriate fallback mechanisms. Training used a small number of GPUs, and the associated computational cost is modest relative to large-scale model training.

REFERENCES

Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. Routefinder: Towards foundation models for vehicle routing problems. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2022.

Inguk Choi, Woo-Jin Shin, Sanghyun Cho, and Hyun-Jung Kim. Towards generalizable multi-policy optimization with self-evolution for job scheduling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 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. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2023.05.017.

Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H Jordan, Carl Kesselman, Philip Maechling, et al. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pp. 14–14. IEEE, 2006.

Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

Edgar Fajardo, Frank Wuerthwein, Brian Bockelman, Miron Livny, Greg Thain, James Alexander Clark, Peter Couvares, and Josh Willis. Adapting ligo workflows to run in the open science grid. *SoftwareX*, 14:100679, 2021.

Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45 (9):1563–1581, 1966.

Wonseok Jeon, Mukul Gagrani, Burak Bartan, Weiliang Will Zeng, Harris Teague, Piero Zappi, and Christopher Lott. Neural DAG scheduling via one-shot priority sampling. In *International Conference on Learning Representations*, 2023.

Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P Berman, and Phil Maechling. Scientific workflow applications on amazon ec2. In *2009 5th IEEE International Conference on e-Science Workshops*, pp. 59–66. IEEE, 2009.

Andrej Karpathy. nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts. GitHub repository, 2022. URL https://github.com/karpathy/nanoGPT. Accessed: 2026-09-26.

Daniel S Katz, Joseph C Jacob, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, GB Berriman, J Good, AC Laity, and Thomas A Prince. A comparison of two methods for building astronomical image mosaics on a grid. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pp. 85–94. IEEE, 2005.

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, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

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

Jonathan Livny, Hidayat Teonadi, Miron Livny, and Matthew K Waldor. High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas. *PloS one*, 3(9):e3197, 2008.

Junyoung Park, Sanjar Bakhtiyar, and Jinkyoo Park. Schedulenet: Learn to solve multi-agent scheduling problems with reinforcement learning. *arXiv preprint arXiv:2106.03051*, 2021.

Michael L. Pinedo. *General Purpose Procedures for Deterministic Scheduling*, pp. 375–398. Springer International Publishing, 2016. ISBN 978-3-319-26580-3. doi: 10.1007/978-3-319-26580-3_14.

Xumai Qi, Dongdong Zhang, Taotao Liu, and Hongcheng Wang. Reinforcement learning for one-shot dag scheduling with comparability identification and dense reward. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 149539–149563, 2025. doi: 10.52202/085713-5002.

Ya Shen, Gang Chen, Hui Ma, and Mengjie Zhang. Deft scheduling of dynamic cloud workflows with varying deadlines via mixture-of-experts. In *International Conference on Learning Representations*, 2026.

Paul ME Shutler. A priority list based heuristic for the job shop problem. *Journal of the Operational Research Society*, 54(6):571–584, 2003.

Igor G. Smit, Jianan Zhou, Robbert Reijnen, Yaoxin Wu, Jian Chen, Cong Zhang, Zaharah Bukhsh, Yingqian Zhang, and Wim Nuijten. Graph neural networks for job shop scheduling problems: A survey. *Computers & Operations Research*, 176:106914, 2025. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2024.106914.

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. doi: 10.1109/TII.2022.3189725.

H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3): 260–274, 2002. doi: 10.1109/71.993206.

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. *Advances in Neural Information Processing Systems*, 38:82849–82877, 2026.

Run Wang et al. Daniel: A deep attention network for flexible job shop scheduling. *IEEE Transactions on Industrial Informatics*, 2024.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

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.

Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research*, 142:105731, 2022.

Yifan Yang, Gang Chen, Hui Ma, Cong Zhang, Zhiguang Cao, and Mengjie Zhang. Graph assisted offline-online deep reinforcement learning for dynamic workflow scheduling. In *International Conference on Learning Representations*, 2025.

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.

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*, volume 2024, pp. 45696–45716, 2024.

CONTENTS

A Unified List Scheduling Formulation **13**

A.1 Unified Scheduling Formulation 13

A.2 Specialization to JSP, FJSP, and WSP 14

A.3 List Scheduling Construction 15

B Static and Dynamic Raw Features **15**

B.1 Static Graph Features 16

B.2 Dynamic Decoder Features 16

C Deterministic Schedule Builder **18**

D Training Algorithm and Configurations **18**

E Details of Benchmarks **20**

F Details of Baselines **20**

A UNIFIED LIST SCHEDULING FORMULATION

We provide the mathematical formulation underlying the common list scheduling process introduced in Section 3. The formulation covers JSP, FJSP, and WSP through the same task–processor abstraction, while preserving their problem-specific precedence and compatibility constraints.

A.1 UNIFIED SCHEDULING FORMULATION

Consider an instance $\mathcal{I} = (\mathcal{O}, \mathcal{M}, \mathcal{E}_{\text{pre}})$, where $\mathcal{O}$ is the set of tasks, $\mathcal{M}$ is the set of processors, and $\mathcal{E}_{\text{pre}} \subseteq \mathcal{O} \times \mathcal{O}$ contains the precedence constraints. For each task $O_{ij} \in \mathcal{O}$, let

$$\text{Pred}(O_{ij}) = \{O_{pq} \mid (O_{pq}, O_{ij}) \in \mathcal{E}_{\text{pre}}\} \tag{25}$$

denote its immediate predecessors. Each task can be processed by a nonempty compatible processor set $\mathcal{M}_{ij} \subseteq \mathcal{M}$. Let

$$\mathcal{K}_{ij} = \{k \mid M_k \in \mathcal{M}_{ij}\} \tag{26}$$

be the corresponding processor-index set, and let $\mu_{ij} \in \mathcal{K}_{ij}$ denote the processor selected for O_{ij}. The processing time of O_{ij} on M_k is denoted by pt_{ij}^k.

Once μ_{ij} is determined, we denote the realized start and finish times of task O_{ij} by st_{ij} and ft_{ij}, respectively:

$$ft_{ij} = st_{ij} + pt_{ij}^{\mu_{ij}}. \tag{27}$$

Since processing is nonpreemptive, each task occupies one contiguous interval $[st_{ij}, ft_{ij})$ on its selected processor.

A feasible schedule must first satisfy all precedence constraints:

$$ft_{pq} \le st_{ij}, \qquad \forall\, (O_{pq}, O_{ij}) \in \mathcal{E}_{\text{pre}}. \tag{28}$$

It must also satisfy processor compatibility:

$$\mu_{ij} \in \mathcal{K}_{ij}, \qquad \forall\, O_{ij} \in \mathcal{O}, \tag{29}$$

and processor capacity. Specifically, any two distinct tasks assigned to the same processor cannot overlap:

$$\mu_{ij} = \mu_{pq} \implies (ft_{ij} \le st_{pq}) \vee (ft_{pq} \le st_{ij})\,. \tag{30}$$

Table 3: Specialization of the unified scheduling formulation to JSP, FJSP, and WSP.

Problem	Precedence structure	Compatible processor set $\mathcal{M}_{ij}$	Processing time pt_{ij}^k
JSP	Job-specific chains	$\{M_{\kappa_{ij}}\}$	pt_{ij}
FJSP	Job-specific chains	$\mathcal{M}_{ij} \subseteq \mathcal{M}$	Machine-dependent pt_{ij}^k
WSP	General DAG	$\mathcal{M}$	$\frac{tw_{ij}}{ms_k}$

The makespan of a complete schedule is

$$C_{\max} = \max_{O_{ij} \in \mathcal{O}} ft_{ij}, \tag{31}$$

and the unified scheduling objective is

$$\begin{aligned} \min_{\{\mu_{ij}, st_{ij}\}} \quad & C_{\max} \\ \text{s.t.} \quad & ft_{ij} = st_{ij} + pt_{ij}^{\mu_{ij}}, \\ & ft_{pq} \le st_{ij}, \qquad \forall\, (O_{pq}, O_{ij}) \in \mathcal{E}_{\text{pre}}, \\ & \mu_{ij} \in \mathcal{K}_{ij}, \qquad \forall\, O_{ij} \in \mathcal{O}, \\ & (ft_{ij} \le st_{pq}) \vee (ft_{pq} \le st_{ij}), \\ & st_{ij} \ge 0, \qquad \forall\, O_{ij} \in \mathcal{O}. \end{aligned} \tag{32}$$

A.2 Specialization to JSP, FJSP, and WSP

The three scheduling problems differ only in their precedence graph, compatible processor sets, and processing-time definitions under the above formulation, as summarized in Table 3.

JSP. For each operation O_{ij}, let $M_{\kappa_{ij}}$ denote its designated machine. Its compatible set is the singleton

$$\mathcal{M}_{ij} = \{M_{\kappa_{ij}}\}, \qquad pt_{ij}^{\kappa_{ij}} = pt_{ij}. \tag{33}$$

The precedence graph consists of job-specific chains:

$$\mathcal{E}_{\text{pre}} = \bigcup_{J_i \in \mathcal{J}} \{(O_{ij}, O_{i,j+1}) \mid j = 1, \ldots, \ell_i - 1\}. \tag{34}$$

Therefore, processor selection is fixed for JSP, and solution construction only determines the relative scheduling priority of operations sharing machines.

FJSP. FJSP has the same job-specific precedence structure as JSP, but each operation can choose one processor from a compatible subset:

$$\mu_{ij} \in \mathcal{K}_{ij}, \qquad \mathcal{M}_{ij} \subseteq \mathcal{M}, \tag{35}$$

where the processing time pt_{ij}^k depends on the selected processor M_k. Hence, both task prioritizing and processor selection affect the resulting schedule.

WSP. For WSP, $\mathcal{E}_{\text{pre}}$ forms a general DAG rather than job-specific chains. Under the formulation considered in this work, each task is compatible with every processor:

$$\mathcal{M}_{ij} = \mathcal{M}. \tag{36}$$

Given task workload tw_{ij} and processor speed ms_k, the processor-dependent processing time is

$$pt_{ij}^k = \frac{tw_{ij}}{ms_k}. \tag{37}$$

Thus, JSP, FJSP, and WSP share the same scheduling objective and feasibility conditions in Eq. (32), while differing in $\mathcal{E}_{\text{pre}}$, $\mathcal{M}_{ij}$, and pt_{ij}^k.

A.3 List Scheduling Construction

SchedGPT constructs a solution through the two-stage list scheduling process introduced in Section 3. Before construction step t, let

$$\mathcal{V}_{t-1} = \{O^{(1)}, \ldots, O^{(t-1)}\} \tag{38}$$

denote the tasks already placed in the partial schedule. A task is *construction-ready* when all of its predecessors have already been placed:

$$\mathcal{R}_t = \{O_{ij} \in \mathcal{O} \setminus \mathcal{V}_{t-1} \mid \mathrm{Pred}(O_{ij}) \subseteq \mathcal{V}_{t-1}\}. \tag{39}$$

This definition concerns the order in which a static schedule is constructed. The actual execution of O_{ij} remains constrained by Eq. (28), i.e., it cannot start before all of its predecessors have finished.

At step t, *task prioritizing* selects

$$O^{(t)} \in \mathcal{R}_t, \tag{40}$$

after which *processor selection* chooses

$$M^{(t)} \in \mathcal{M}_{i_t j_t}, \qquad O^{(t)} = O_{i_t j_t}. \tag{41}$$

The resulting decisions extend the two aligned lists:

$$\mathbf{L}_t^O = \mathbf{L}_{t-1}^O \oplus O^{(t)}, \qquad \mathbf{L}_t^M = \mathbf{L}_{t-1}^M \oplus M^{(t)}, \tag{42}$$

where $\oplus$ denotes the append operation.

After selecting the pair $(O^{(t)}, M^{(t)})$, the schedule builder determines its start and finish times. Let $\mathcal{O}_k^{t-1}$ denote the tasks already assigned to processor M_k in the partial schedule. The assigned time interval must satisfy

$$st_{i_t j_t} \geq \max_{O_{pq} \in \mathrm{Pred}(O^{(t)})} ft_{pq}, \tag{43}$$

with an empty maximum defined as zero, and

$$[st_{i_t j_t}, ft_{i_t j_t}) \cap [st_{pq}, ft_{pq}) = \emptyset, \qquad \forall\, O_{pq} \in \mathcal{O}_{k_t}^{t-1}, \tag{44}$$

where $M^{(t)} = M_{k_t}$. The exact deterministic rule used by the schedule builder to choose a feasible start time is described in Appendix C.

The construction repeats until $t = n = |\mathcal{O}|$. Since $\mathcal{E}_{\mathrm{pre}}$ is acyclic, every nonempty unscheduled subgraph contains at least one task whose predecessors have already been scheduled; therefore, $\mathcal{R}_t$ is nonempty until all tasks are selected. Moreover, Eq. (41) enforces processor compatibility, while Eqs. (43) and (44) enforce precedence and processor capacity. Consequently, the complete aligned lists $(\mathbf{L}_n^O, \mathbf{L}_n^M)$ can be deterministically mapped to a feasible schedule satisfying Eq. (32).

B Static and Dynamic Raw Features

We detail the features used by the graph encoder and the two pointer heads, as shown in Table 4. Static features are computed once per instance, whereas dynamic features are updated from the partial schedule before each decision. For compactness, let $v = O_{ij}$, $\mathcal{M}_v = \mathcal{M}_{ij}$, and $p_{vk} = pt_{ij}^k$. Processing times are expressed in the input time units.

Table 4: Summary of the static and dynamic features used in SchedGPT.

Feature type	Symbol	Dimension
Static task features	$\mathbf{f}_{ij}^O$	6
Static processor features	$\mathbf{f}_k^M$	3
Static compatibility-edge features	$\mathbf{f}_{ij,k}^E$	3
Dynamic task features	$\mathbf{f}_{O_{ij}}^{(t)}$	7
Dynamic processor features	$\mathbf{f}_{M_k}^{(t)}(O^{(t)})$	4

B.1 Static Graph Features

For each task v, define its average and minimum processing times as

$$\bar{p}_v = \frac{1}{|\mathcal{M}_v|} \sum_{M_k \in \mathcal{M}_v} p_{vk}, \qquad p_v^{\min} = \min_{M_k \in \mathcal{M}_v} p_{vk}. \tag{45}$$

The instance-level processing-time scale is

$$\mu_{\mathcal{I}} = \frac{\sum_{v \in \mathcal{O}} \sum_{M_k \in \mathcal{M}_v} p_{vk}}{\sum_{v \in \mathcal{O}} |\mathcal{M}_v|}. \tag{46}$$

The following definitions concern tasks with positive processing times and nonempty compatible sets. We use $\epsilon_f = 10^{-5}$ in feature-ratio denominators.

Task features. Let $\mathrm{Pred}(v)$ and $\mathrm{Succ}(v)$ denote the immediate predecessors and successors of v. Forward and backward processing aggregates are

$$U_v = \bar{p}_v + \max_{u \in \mathrm{Pred}(v)} U_u, \qquad V_v = \bar{p}_v + \max_{w \in \mathrm{Succ}(v)} V_w, \tag{47}$$

where an empty maximum is zero. On a job chain, these become the prefix and suffix sums of average processing times, including v. They describe processing requirements along precedence paths, rather than guaranteed makespan lower bounds.

The remaining-position feature is

$$\rho_v = \begin{cases} (\ell_i - j + 1)/\ell_i, & v = O_{ij} \text{ in JSP or FJSP}, \\ d_v / \max_{u \in \mathcal{O}} d_u, & v \text{ in WSP}, \end{cases} \qquad d_v = 1 + \max_{w \in \mathrm{Succ}(v)} d_w. \tag{48}$$

Thus, ρ_v measures remaining job position for job chains and normalized remaining path depth for workflows.

The six-dimensional task feature vector is

$$\mathbf{f}_{ij}^{O} = \left[\frac{U_v}{5\mu_{\mathcal{I}}}, \frac{V_v}{5\mu_{\mathcal{I}}}, \rho_v, \frac{\bar{p}_v}{\mu_{\mathcal{I}}}, \frac{p_v^{\min}}{\mu_{\mathcal{I}}}, \frac{|\mathcal{M}_v|}{m} \right]. \tag{49}$$

Processor features. Let $\mathcal{C}_k = \{v \in \mathcal{O} : M_k \in \mathcal{M}_v\}$ be the tasks compatible with processor M_k. Its feature vector is

$$\mathbf{f}_k^{M} = \left[\frac{|\mathcal{C}_k|}{n_{\mathrm{slot}}}, \frac{\sum_{v \in \mathcal{C}_k} p_{vk}}{|\mathcal{C}_k| \mu_{\mathcal{I}}}, \frac{1}{|\mathcal{C}_k|} \sum_{v \in \mathcal{C}_k} \frac{p_{vk}}{\bar{p}_v + \epsilon_f} \right]. \tag{50}$$

The entries describe compatibility coverage, average processing time, and average relative processing time, respectively. Here, n_{slot} is the number of task slots stored for an instance before batch-level padding; it equals n for an unpadded instance. The shop converter retains within-instance padding in this denominator. If $\mathcal{C}_k = \emptyset$, we set $\mathbf{f}_k^{M} = \mathbf{0}$. These features are static instance statistics, not current processor loads.

Edge features. Each compatible task–processor pair has

$$\mathbf{f}_{ij,k}^{E} = \left[\frac{p_{vk}}{1000}, \frac{p_{vk}}{p_v^{\min} + \epsilon_f}, \frac{p_{vk}}{\bar{p}_v + \epsilon_f} \right]. \tag{51}$$

The two directed compatibility edges share this vector. Precedence edges and task self-loops have zero-valued edge features under the zero-communication formulation in Appendix A.

B.2 Dynamic Decoder Features

Before step t, let $\mathcal{V}_{t-1}$ contain the operations already scheduled, and let $\mathcal{U}_t = \mathcal{O} \setminus \mathcal{V}_{t-1}$. Let $\mathcal{J}^+$ denote the set of nonempty jobs. For each $j \in \mathcal{J}^+$, let n_j be its total number of operations and $q_j^{(t)}$ the number already scheduled before step t. The frontier-operation set is

$$\mathcal{F}_t = \left\{ O_{j, q_j^{(t)}+1} : j \in \mathcal{J}^+,\ q_j^{(t)} < n_j \right\} \subseteq \mathcal{U}_t. \tag{52}$$

Thus, $\mathcal{F}_t$ contains exactly one candidate operation from each unfinished job.

Denote machine M_k's current tail completion time by $a_k^{(t)}$, defined as the maximum completion time of the operations already assigned to that machine, or zero if none has been assigned. All processing,

start, and completion times below are expressed in the original instance units. For a frontier operation v, define

$$\begin{aligned} r_v^{(t)} &= \max_{u \in \mathrm{Pred}(v)} ft_u, \qquad v \in \mathcal{F}_t, \\ \tau_{\mathcal{I}} &= \max_{\substack{v \in \mathcal{O} \\ M_k \in \mathcal{M}_v}} pt_v^k, \\ \nu_t &= \max\left\{ \tau_{\mathcal{I}}, \max_{M_k \in \mathcal{M}} a_k^{(t)} \right\}. \end{aligned} \tag{53}$$

Here, $\mathrm{Pred}(v)$ denotes the job-precedence predecessors of v, with an empty maximum defined as zero. Since $v \in \mathcal{F}_t$, all its predecessors have already been scheduled, and $r_v^{(t)}$ is its exact precedence release time. The scale $\tau_{\mathcal{I}}$ excludes padded operations and incompatible operation–machine pairs.

For each $v \in \mathcal{F}_t$ and $M_k \in \mathcal{M}_v$, let $S_{v,k}^{(t)}$ denote the earliest feasible insertion start time on M_k: the operation starts no earlier than $r_v^{(t)}$ and its processing interval does not overlap any interval already scheduled on that machine. Let $B_{v,k}^{(t)}$ denote the completion time of the occupied interval immediately preceding the corresponding insertion gap, with $B_{v,k}^{(t)} = 0$ if no such interval exists. These queries leave previously scheduled intervals unchanged.

We define

$$\begin{aligned} k_v^\star(t) &\in \operatorname*{arg\,min}_{k:\, M_k \in \mathcal{M}_v} \left(S_{v,k}^{(t)} + pt_v^k \right), \\ E_v^{(t)} &= S_{v,k_v^\star(t)}^{(t)}, \\ \underline{a}_v^{(t)} &= B_{v,k_v^\star(t)}^{(t)}, \\ \widehat{p}_v^{(t)} &= pt_v^{k_v^\star(t)}, \\ \bar{p}_{\mathcal{F}_t}^{(t)} &= \frac{\sum_{u \in \mathcal{F}_t} \widehat{p}_u^{(t)}}{\max\{1, |\mathcal{F}_t|\}}, \\ \rho_j^{(t)} &= \frac{n_j - q_j^{(t)}}{n_j}, \qquad j \in \mathcal{J}^+, \\ \bar{g}_t &= \frac{1}{|\mathcal{J}^+|} \sum_{j \in \mathcal{J}^+} \left(1 - \rho_j^{(t)}\right). \end{aligned} \tag{54}$$

Here, $E_v^{(t)}$ is the earliest feasible insertion start time on the compatible machine yielding the earliest completion of operation v; it need not be the minimum start time across all compatible machines. The quantity $\underline{a}_v^{(t)}$ is the left boundary of the corresponding insertion gap, rather than the minimum machine-tail completion time. The reference machine $M_{k_v^\star(t)}$ is used only to construct the dynamic features; the actual machine assignment is selected by the policy.

The mean processing time $\bar{p}_{\mathcal{F}_t}^{(t)}$ averages the reference-machine processing times over frontier operations only. The mean progress $\bar{g}_t$ averages the scheduled-operation fraction across all nonempty jobs, including fully scheduled jobs. Fully scheduled jobs contribute 1, while unfinished jobs contribute $1 - \rho_j^{(t)}$.

The equations describe the exact-arithmetic quantities. The implementation uses tolerance-aware comparisons for gap feasibility and machine selection, choosing the smallest machine index among numerically tied minima.

Task features. The task pointer receives

$$\begin{aligned} \mathbf{f}_{O_{ij}}^{(t)} = \Big[& \frac{E_v^{(t)}}{\nu_t}, \frac{E_v^{(t)} + \bar{p}_v}{\nu_t}, \frac{E_v^{(t)} - r_v^{(t)}}{\nu_t}, \frac{E_v^{(t)} - \underline{a}_v^{(t)}}{\nu_t}, \\ & \rho_v, \frac{\bar{p}_v}{\bar{p}_{\mathcal{U}_t} + \epsilon_f}, (1 - \rho_v) - \bar{g}_t \Big]. \end{aligned} \tag{55}$$

In order, these entries describe estimated start and completion times, waiting and availability gaps, remaining position, relative processing time, and relative progress. The completion estimate $E_v^{(t)} + \bar{p}_v$ uses an average processing time; it is not the minimum processor-specific finish time. The feature ρ_v is reused from the static representation rather than recomputed after each decision.

Processor features. Given the selected task $v = O^{(t)}$, the processor pointer receives

$$\mathbf{f}_{M_k}^{(t)}(v) = \left[\frac{a_k^{(t)}}{\nu_t}, \frac{p_{vk}}{\nu_t}, \frac{\max\{r_v^{(t)}, a_k^{(t)}\} + p_{vk}}{\nu_t}, \frac{\nu_t - a_k^{(t)}}{\nu_t} \right]. \tag{56}$$

The entries represent availability, processing time, projected finish time, and relative tail-availability gap. The last entry is not total processor idle time. For incompatible processors, the processing-time and projected-finish entries are zeroed, and the final action mask assigns zero selection probability.

C Deterministic Schedule Builder

The builder maps the ordered assignments in $\mathbf{Y}_t$ to the partial schedule $\mathcal{X}_t$, as defined in Eq. (4). We use an earliest-feasible insertion rule: each selected operation is scheduled at the earliest feasible start time on its assigned machine, subject to job-precedence and machine-capacity constraints. Both sampled and greedy-baseline solutions are evaluated with this rule, whose pseudo-code is shown in Algorithm 1.

Algorithm 1 Deterministic List-to-Schedule Builder

Require: Instance $\mathcal{I}$ and valid ordered assignments $\mathbf{Y} = [(O^{(t)}, M^{(t)})]_{t=1}^{n}$ containing every operation exactly once
Ensure: Complete schedule $\mathcal{X}$ and makespan $C_{\max}$

```
Initialize 𝒳 ← ∅, 𝒱 ← ∅, and C_max ← 0
Initialize an empty sorted interval list ℒ_k ← [ ] for each M_k ∈ ℳ
for t = 1, ..., n do
    (v, M_k) ← (O^(t), M^(t))
    Require v ∉ 𝒱, Pred(v) ⊆ 𝒱, and M_k ∈ ℳ_v
    r_v ← max_{u∈Pred(v)} ft_u, with max ∅ = 0
    st_v ← r_v
    for each (s, e) in ℒ_k in increasing order of s do
        if st_v + pt_v^k ≤ s then
            break
        end if
        st_v ← max{st_v, e}
    end for
    ft_v ← st_v + pt_v^k
    Insert (st_v, ft_v) into ℒ_k, maintaining increasing start-time order
    C_max ← max{C_max, ft_v}
    𝒱 ← 𝒱 ∪ {v}
    𝒳 ← 𝒳 ∪ {(v, M_k, st_v, ft_v)}
end for
return 𝒳, C_max
```

Feasibility and replay. Compatibility follows from the processor mask. The release-time bound enforces precedence, and the processor-tail bound prevents overlap. The same instance and ordered assignments produce the same times, so replaying $\mathbf{Y}_t$ recovers $\mathcal{X}_t$. This establishes feasibility and determinism, not optimality.

D Training Algorithm and Configurations

Algorithm 2 summarizes the training procedure of SchedGPT. The same model parameters are shared across JSP, FJSP, and WSP. At each update, the current policy samples complete task–processor lists, while a frozen rollout policy greedily generates reference solutions for computing the relative advantage. The shared encoder and decoder are then optimized using Eq. (23). The rollout baseline

is periodically updated when the current policy improves on the validation instances. The training hyperparameters are presented in Table 5.

Algorithm 2 Training SchedGPT

Input: training distributions $\{\mathcal{D}_{\mathrm{JSP}}, \mathcal{D}_{\mathrm{FJSP}}, \mathcal{D}_{\mathrm{WSP}}\}$, model parameters θ
Output: trained parameters θ
Initialize frozen rollout baseline $\bar{\theta} \leftarrow \theta$
for each training epoch **do**
  Construct training batches alternately from JSP, FJSP, and WSP
  for each minibatch $\mathcal{B}$ **do**
    for each instance $\mathcal{I} \in \mathcal{B}$ **do**
      Sample $\mathbf{Y} \sim \pi_\theta(\cdot \mid \mathcal{I})$ using task prioritizing and processor selection
      $\mathcal{X} \leftarrow \mathrm{Builder}(\mathcal{I}, \mathbf{Y})$, $C \leftarrow C_{\max}(\mathcal{X})$
      Greedily decode $\bar{\mathbf{Y}} \leftarrow \mathrm{Greedy}(\pi_{\bar{\theta}}, \mathcal{I})$
      $\bar{\mathcal{X}} \leftarrow \mathrm{Builder}(\mathcal{I}, \bar{\mathbf{Y}})$, $\bar{C} \leftarrow C_{\max}(\bar{\mathcal{X}})$
      Compute relative advantage $A \leftarrow (C - \bar{C})/\bar{C}$
    end for
    Normalize advantages in $\mathcal{B}$ to obtain $\hat{A}$
    Compute $\mathcal{L}(\theta)$ using Eq. (23)
    Update shared parameters θ
  end for
  Evaluate π_θ and $\pi_{\bar{\theta}}$ on validation instances
  if π_θ improves over $\pi_{\bar{\theta}}$ **then**
    $\bar{\theta} \leftarrow \theta$
  end if
end for
return θ

Table 5: **SchedGPT model and training hyperparameters.**

Hyperparameter	Value
Hidden dimension d	256
GATv2 layers ℓ_{enc}	3
GATv2 attention heads	1
Transformer layers ℓ_{dec}	6
Transformer attention heads	8
Pointer attention heads	8
Transformer FFN dimension	1024
Dropout	0
Optimizer	AdamW
Peak learning rate	1.5×10^{-4}
Weight decay	1×10^{-4}
Learning-rate scheduler	OneCycleLR
Maximum batch size	256
Maximum gradient norm	1.0
Training epochs	50
Updates per epoch	2000
Entropy coefficient β	$0.01 \rightarrow 0.001$
Validation instances per configuration	32
Random seed	42

Hardware and Implementation. We implement SchedGPT in PyTorch, with its three-layer heterogeneous GATv2 encoder implemented using PyTorch Geometric. Experiments are conducted on a server equipped with an AMD Ryzen Threadripper 3960X CPU with 24 cores and 48 threads, and four NVIDIA RTX A6000 GPUs, each with 48 GB of GPU memory.

E Details of Benchmarks

Table 6 summarizes the benchmark sets used for the three scheduling problems.

Table 6: **Summary of the evaluation benchmarks.**

Problem	Benchmark	Setting	# Instances
JSP	Taillard	8 size groups, 15×15–100×20	80
FJSP	Brandimarte	55–284 tasks	15
FJSP	Hurink-edata	50–300 tasks	40
FJSP	Hurink-rdata	50–300 tasks	40
FJSP	Hurink-vdata	50–300 tasks	40
WSP	Pegasus	5 workflows × 5 requested sizes	500

JSP. We use the Taillard benchmark instances following L2D[2] (Zhang et al., 2020). The eight evaluated job–machine sizes are 15×15, 20×15, 20×20, 30×15, 30×20, 50×15, 50×20, and 100×20, with 10 instances per size. These correspond to 225–2000 tasks and 15 or 20 processors. Following the standard benchmark protocol, solution quality is measured by the makespan gap to the reference upper bound.

FJSP. We evaluate on the 15 Brandimarte instances and three Hurink benchmark sets (*edata*, *rdata*, and *vdata*), each containing 40 instances[3] (Song et al., 2023). These benchmarks cover different task–processor compatibility structures and contain 50–300 tasks with 4–15 processors across the evaluated sets. We report the makespan gap to the corresponding reference upper bound.

WSP. As shown in Figure 3, this paper uses five Pegasus[4] scientific workflow types: *SIPHT*, *LIGO*, *Genome*, *CyberShake*, and *Montage* (Deelman et al., 2015). For each workflow type, we evaluate requested sizes

$$\{100, 200, 300, 400, 1000\},$$

with 20 independently generated instances for each workflow–size pair, giving 500 test instances in total. The requested size specifies the workflow-generator setting during training. The test dataset can be downloaded directly via this link[5].

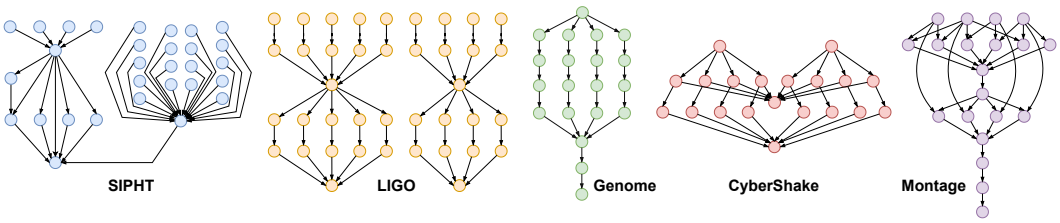


Figure 3: Representative DAG Topologies of Pegasus Scientific Workflows.

F Details of Baselines

We briefly describe the baselines used in Table 1. For learning-based methods, we use greedy inference for solution-quality evaluation, consistent with the setting of SchedGPT.

JSP Baselines. **L2D** (Zhang et al., 2020) formulates JSP as sequential dispatching and uses a graph neural network with reinforcement learning to select the next eligible operation. **ScheduleNet** (Park

[2]https://github.com/zcaicaros/L2D
[3]https://github.com/songwenas12/fjsp-drl
[4]https://pegasus.isi.edu/documentation/user-guide/introduction.html
[5]https://download.pegasus.isi.edu/misc/SyntheticWorkflows.tar.gz

et al., 2021) represents the JSP state as a graph and jointly learns graph representations and a reinforcement-learning scheduling policy. **ReSched** (Xiao et al., 2026) is a Transformer-based reinforcement-learning scheduler that uses a simplified state representation and can be directly applied to JSP.

FJSP Baselines. **HGNN** (Song et al., 2023) represents operations and machines using a heterogeneous graph and learns to select feasible operation–machine pairs. **DANIEL** (Wang et al., 2024) uses a dual-attention architecture to model interactions between operations and machines for reinforcement-learning-based FJSP scheduling. **ReSched** (Xiao et al., 2026) provides the Transformer-based baseline described above, instantiated for FJSP.

WSP Baselines. **HEFT** (Topcuoglu et al., 2002) is the reference list-scheduling heuristic used to compute the reported Pegasus gaps. It prioritizes tasks using upward ranks and assigns each selected task according to earliest finish time. Since HEFT has zero gap to itself, its column is omitted from the Pegasus block of Table 1. **FIFO-EFT** prioritizes ready tasks in first-in-first-out order, while **SPT-EFT** prioritizes the ready task with the smallest processing demand; both assign the selected task to the processor yielding the earliest feasible finish time (Pinedo, 2016). **Jeon2023** (Jeon et al., 2023) predicts task priorities with a one-shot neural model and converts the resulting priorities into a schedule through list scheduling. **Qi2025** (Qi et al., 2025) is a one-shot reinforcement-learning scheduler that introduces comparability-aware priority learning and denser training feedback before applying list scheduling.