KNOWLEDGE-AUGMENTED RL AND CP FOR FLEXIBLE JOB-SHOP SCHEDULING

Anonymous authors
Paper under double-blind review

ABSTRACT

Job-shop scheduling (JSSP) and flexible job-shop scheduling (FJSP) are challenging combinatorial problems in which fast constructive policies may be limited by early decisions during schedule construction. We propose a knowledge-augmented reinforcement learning (RL) and constraint programming (CP) method that separates schedule construction from a hierarchical Destroy-to-Improve stage. A heterogeneous scheduling graph combines structural information with heuristic rules, Pattern Memory, and Repair-Aware Memory. The Destroy Policy determines *where* to search, an Operation-Selection Transformer scores the operations within the selected candidate to determine *what* to release, and CP determines *how* to reconstruct the resulting region while preserving feasibility. The models are trained on a moderate-scale curriculum of synthetic JSSP/FJSP instances up to 30×20, yet remain effective on substantially larger benchmarks, obtaining competitive gaps with favorable quality-time trade-offs. They also transfer to flexible flow-shop scheduling without additional training. Finally, through a schedule-format adapter, the frozen Destroy-to-Improve model improves 445 of 675 schedules produced by five ReSched and DANIEL variants, without further fine-tuning.

1 INTRODUCTION

Job-shop scheduling (JSSP) and its flexible generalization (FJSP) must schedule operations on limited machines under precedence constraints. In JSSP, each operation has a fixed machine, whereas FJSP allows several eligible machines and therefore couples sequencing with machine assignment. Each machine can process one operation at a time in a non-preemptive fashion. The usual objective is to minimize the makespan, i.e., the completion time of the last operation. Although simple to state, these constraints interact strongly: moving or reassigning one operation can alter machine congestion and delay downstream operations (Dauzère-Pérès et al., 2024).

Learning-based scheduling increasingly replaces manually designed dispatching rules with policies that construct schedules directly from problem instances (Zhang et al., 2020; Park et al., 2021b; Song et al., 2023; Wang et al., 2024; Xiao et al., 2026). Such policies can produce solutions quickly, but early decisions may be difficult to revise. This has motivated methods that improve an incumbent through learned local search, deconstruction, or neighborhood selection (Zhang et al., 2024a;b; Wang et al., 2025; Zago et al., 2026). Construction and improvement therefore address complementary questions: how to obtain a feasible schedule and which parts of it are worth reconsidering.

This distinction is especially useful when learning is combined with standard model-based combinatorial optimization. A learned model can identify where additional search may help, while a constraint solver reconstructs the selected region under precedence, capacity, and eligibility constraints. Learning and CP have already been shown to complement each other in scheduling (Tassel et al., 2023). Our goal, therefore, is not to replace constraint reasoning, but to redirect its focus toward promising regions of the search space. To this end, we introduce a construct-and-improve method, which we call KARMS: **K**nowledge-**A**ugmented **R**einforcement learning for **M**achine **S**cheduling with adaptive constraint repair.

KARMS first constructs a complete feasible schedule by scoring only currently feasible operation-machine decisions. The environment exposes ready operations and their eligible machines, so invalid pairs never reach the policy. In JSSP, this reduces to choosing the next eligible operation; in

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.

FJSP, the policy jointly selects an operation and machine. A lightweight CP lookahead can rerank a small set of promising construction decisions before the schedule enters *Destroy-to-Improve*.

KARMS represents states with a knowledge-augmented heterogeneous graph rather than a flat feature vector. Operations, machines, jobs, and their relations encode structural information, while heuristic and memory signals provide relational evidence. During Construct, processing time, remaining work, earliest finish, slack, and bottleneck-related guidance support feasible decisions without acting as fixed rules, allowing the actor to determine their relevance from context. This use of domain and historical knowledge relates to knowledge-graph-enhanced RL (Qin & Lu, 2025).

Construct is further supported by an offline *Pattern Memory*. Each observed state is summarized by scheduling progress, machine-load imbalance, remaining work, readiness, bottleneck intensity, job-completion spread, machine flexibility, and remaining-work dispersion. Normalized signatures are grouped using K-Means with K-Means++ initialization (Arthur & Vassilvitskii, 2007; Celebi et al., 2013), and the resulting clusters retain statistics about historically useful scheduling rules and operation-machine decisions. At inference, nearby clusters provide graph evidence while the Construct actor remains responsible for the final action distribution.

Then, Destroy-to-Improve operates on the complete incumbent rather than rebuilding it. Complete-solution graph representations have proved useful for learned scheduling improvement (Zhang et al., 2024a; Wang et al., 2025); KARMS augments them with realized precedence and machine order, repair channels, rule information, memory evidence, and candidate context. The representation therefore captures both the current schedule and evidence about where it may be useful to modify it.

KARMS then generates structured candidate neighborhoods around critical paths and blocks, bottleneck-machine segments, low-slack operations, and late-job regions. A learned Candidate Head combines operation embeddings with graph-level and candidate-specific information to rank these neighborhoods, answering the first Destroy-to-Improve question: *where should the current schedule be reopened?* Candidates that fail to produce a useful change can be temporarily excluded.

A separate *Repair-Aware Memory*, built offline from previous repair attempts, supports this decision. Its signatures combine schedule diagnostics with repair context, while clusters retain statistics such as repair success, makespan improvement, runtime, repair cost, and repair budget. Nearby repair patterns provide historical evidence through graph relations; as with Pattern Memory, this evidence remains advisory and is considered jointly with the current schedule, candidate regions, and heuristic guidance. Selecting a neighborhood does not yet determine which operations should be released. KARMS therefore applies an Operation-Selection Transformer to the graph-derived embeddings of operations in the selected candidate. Self-attention allows each operation score to depend on the other operations in the same region (Lee et al., 2019), while a cardinality-constrained structured selector converts these scores into an explicit operation set (Hoppe et al., 2025). Job-tail closure then indicates the effective repair region by including required downstream operations.

CP answers the final question: *how should the selected region be reconstructed?* Operations outside the effective repair region retain their incumbent decisions, while released operations are reoptimized; in FJSP, eligible machine assignments may also change. Accepted reconstructions become the next incumbent and the process repeats for a fixed improvement horizon. Thus, the Candidate Head determines *where* to search, the Operation-Selection Transformer with budgeted Top-B selection determines *what* to release, and CP determines *how* to reconstruct the resulting region.

This decomposition also supports evaluation per component. Construct can be evaluated before and after CP lookahead, while candidate selection, reconstruction, and the two memory sources are examined through targeted component ablations. A paired 2×2 ablation further shows that learned selection and CP reconstruction are complementary: learned and heuristic selection are nearly indistinguishable with Most Work Remaining (MWKR) repair, whereas learned selection provides a large advantage when paired with CP, indicating that the learned policy identifies neighborhoods whose flexibility can be exploited effectively by the constraint solver. The same interfaces apply to both JSSP and FJSP: the learned representations do not require fixed numbers of jobs or machines, while repair handles fixed-machine JSSP operations and flexible-machine FJSP operations.

KARMS is trained once on a mixed curriculum of synthetic JSSP and FJSP instances up to 30×20 and evaluated with the same frozen models on established families containing substantially larger instances. Across all 500 JSSP/FJSP benchmarks, the complete post-construction pipeline reduces

the mean reference gap from 22.71% after raw Construct to 10.34% in the final schedule. On the common 162-instance JSSP subset, KARMS reaches an instance-weighted mean gap of 9%, placing it among the strongest learned methods with complete coverage. Across the 120 Hurink FJSP instances, it achieves a 4.29% mean gap, remaining competitive with strong dedicated improvement methods including NDS-S and MIStar-400 (Zago et al., 2026; Wang et al., 2025). Together, these results show that the same frozen pipeline maintains competitive learned performance across both JSSP and FJSP without size- or benchmark-specific retraining.

We additionally examine transfer beyond the training problem classes. Without FFSP-specific training or fine-tuning, the frozen KARMS pipeline transfers to 3,000 MatNet FFSP instances (Kwon et al., 2021), reducing mean makespan relative to MWKR by a 17.6% macro-average across FFSP20/50/100. All pipeline stages remain beneficial after transfer, and KARMS records hundreds of instance-level wins against MatNet, MatNet$\times 128$, and ReSched-samp (Xiao et al., 2026). Since these FFSP-specific methods remain stronger in mean makespan overall, we interpret this experiment as evidence of cross-problem transfer rather than FFSP-specific superiority.

We also test whether Destroy-to-Improve depends on schedules generated by KARMS's own Construct policy. Through a schedule-format adapter, the same frozen Destroy-to-Improve pipeline is applied to 675 schedules produced by three ReSched and two DANIEL variants (Wang et al., 2024), without retraining or access to the external policies' parameters or representations. KARMS strictly improves 445 of these schedules; the mean BKS-gap reduction is 2.19 percentage points for ReSched and 2.53-3.46 percentage points for the two DANIEL variants. This shows that the learned repair mechanism remains effective on independently generated feasible schedules rather than depending on the distribution produced by its own Construct policy.

Taken together, the experiments test three complementary forms of generalization: improvement of KARMS's own schedules, transfer to a scheduling problem class unseen during training, and repair of schedules produced by independently trained methods. KARMS can therefore operate both as an end-to-end construct-and-improve scheduler and as a pluggable learned repair component.

Contributions. **(i)** We present a common construct-and-improve pipeline for JSSP and FJSP, with separate learned components for initial schedule construction and hierarchical Destroy-to-Improve. **(ii)** We use a knowledge-augmented heterogeneous scheduling representation combining scheduling topology with heuristic rules, pattern and repair-aware memory as relational evidence rather than fixed decisions. **(iii)** We organize Destroy-to-Improve hierarchically: a learned Candidate Head determines *where* to search, the Operation-Selection Transformer with structured Top-B selection determines *what* to release, and CP determines *how* to reconstruct the resulting region while preserving feasibility. **(iv)** We evaluate cross-problem transfer to FFSP and schedule-source transfer on 675 schedules from five ReSched and DANIEL variants, 445 of which are strictly improved without retraining; a paired factorial ablation further isolates learned selection from constraint-based reconstruction.

2 Related Work

Learning-based approaches to JSSP have largely focused on constructing schedules through sequences of learned decisions. L2D introduced a graph-based RL policy for JSSP and demonstrated transfer across instance sizes (Zhang et al., 2020). ScheduleNet represents scheduling problems through an agent-task graph and learns decentralized assignment decisions using type-aware graph attention (Park et al., 2021a). For FJSP, HGNN models operation-machine interactions with a heterogeneous graph (Song et al., 2023), while DANIEL and DOAGNN develop richer attention and aggregation mechanisms for constructive scheduling (Wang et al., 2024; Zhao et al., 2025). Residual Scheduling instead constructs JSSP/FJSP schedules from a reduced residual state that retains only the remaining relevant jobs and machines (Ho et al., 2024). More recently, ReSched uses a reduced state representation and Transformer-based policy and reports transfer across FJSP, JSSP, and FFSP (Xiao et al., 2026). Together, these works show that constructive policies can produce schedules efficiently and transfer across sizes and, in some cases, related scheduling problems.

A complementary line of work starts from a complete schedule and learns how to improve it. L2S learns local improvement decisions for JSSP (Zhang et al., 2024a), while TBGAT incorporates the topology of the realized disjunctive graph into learned local search (Zhang et al., 2024b). MGRO combines a multi-graph schedule representation with RL to adapt search operators within Iterated

Greedy for hybrid flow-shop scheduling (Ni et al., 2021). For FJSP, MIStar combines a heterogeneous solution graph with memory-enhanced improvement search (Wang et al., 2025). Most closely related to our improvement stage, NDS-S learns a deconstruction policy that removes operations from complete JSSP/FJSP schedules and reconstructs the resulting partial solution with a simple repair strategy (Zago et al., 2026). Knowledge-graph-based scheduling has also used relational manufacturing information to guide RL policies (Qin & Lu, 2025). KARMS follows the general improve-from-a-solution view but decomposes Destroy-to-Improve into *where* to search, *what* to release, and *how* to reconstruct, while heuristic, structural, and memory information act as relational evidence rather than fixed repair rules.

Learning has also been combined directly with optimization solvers. Tassel et al. use CP to train a learned dispatching policy for JSSP (Tassel et al., 2023), while Echeverria et al. use CP-generated solutions for learning and combine the learned model with CP during FJSP construction (Echeverria et al., 2025). More recently, PRISMA uses CP solutions to pretrain decentralized FJSP policies and to guide RL exploration through expert-derived reward shaping (Jesus et al., 2026). Hybrid RL-CP has also been explored in distributed flexible flow-shop scheduling, where RL partitions the problem into smaller job subsets that are subsequently optimized by CP (Avgerinos et al., 2026). L-RHO instead learns which variables can remain fixed between rolling-horizon subproblems, reducing the amount of optimization required (Li et al., 2025). Beyond scheduling, learning-guided LNS has learned neighborhood or destroy policies whose selected variables are subsequently reoptimized by mathematical solvers (Sonnerat et al., 2021; Wu et al., 2021). KARMS uses CP in a different role: learned selection determines the region and operations to reopen, while CP performs localized reconstruction of the incumbent schedule at inference time. Learning therefore controls the scope of constraint search rather than using CP primarily as training supervision or repeatedly solving the complete instance.

Generalization remains an important issue in neural scheduling. Size transfer has been studied in graph-based constructive policies (Zhang et al., 2020), and curriculum learning has been used to train dispatching policies on progressively harder JSSP instances (Iklassov et al., 2023). ReSched studies transfer across JSSP, FJSP, and FFSP (Xiao et al., 2026), while multi-policy self-evolution approaches consider several scheduling variants within a common learning framework (Choi et al., 2025). KARMS is trained through a curriculum of JSSP and FJSP instances and evaluated on larger benchmark instances as well as FFSP without FFSP-specific training. We further apply the same frozen Destroy-to-Improve model and CP repair backend to schedules generated by ReSched and DANIEL without retraining or fine-tuning. This evaluates not only transfer across problem settings, but also whether the learned repair mechanism can serve as reusable post-processing for schedules produced by other learned methods.

3 Problem Setting

We consider jobs $\mathcal{J}$ processed on machines $\mathcal{M}$. Each job $j \in \mathcal{J}$ consists of an ordered sequence of operations $o_{j,0}, \ldots, o_{j,n_j-1}$. Operation o can be processed on an eligible-machine set $\mathcal{M}_o \subseteq \mathcal{M}$, with processing time $p_{o,m}$ on machine $m \in \mathcal{M}_o$. Operation order must be respected, machines process at most one operation at a time, and processing is non-preemptive. For a feasible schedule S, the objective is to minimize the makespan $C_{\max}(S) = \max_{o \in \mathcal{O}} e_o$, where $\mathcal{O}$ is the operation set and e_o is the completion time of o (Pinedo, 2008; Dauzère-Pérès et al., 2024). Full notation and constraints are given in Appendix A.

The same representation covers JSSP and FJSP. In JSSP, $|\mathcal{M}_o| = 1$, so machine assignment is fixed and only sequencing remains; in FJSP, $|\mathcal{M}_o| \geq 1$, so assignment and sequencing are decided jointly. For FFSP transfer, each processing stage is mapped to an operation whose eligible-machine set contains the parallel machines of that stage, preserving the same precedence, eligibility, capacity, and makespan interface; the exact conversion to the canonical representation is given in Appendix A.3.

During construction, let S_t be the partial schedule after t decisions. An unscheduled operation is *ready* when its immediate job predecessor has been scheduled, and the feasible action set is $\mathcal{A}_t = \{(o, m) \mid o \text{ is ready}, \ m \in \mathcal{M}_o\}$. The environment therefore enforces precedence readiness and machine eligibility, so learning ranks only feasible alternatives. In JSSP an action selects a ready operation on its predetermined machine; in FJSP it jointly selects a ready operation and an eligible machine.

Destroy-to-Improve instead starts from a complete feasible incumbent S_t. Besides job precedence, the completed schedule induces an order between consecutive operations on each machine, exposing critical paths and blocks, congestion, slack, and late-job regions used to form improvement neighborhoods. A destroy decision releases operations R_t; the repair contract expands them when necessary to R_t^{eff}, while decisions outside this region remain fixed. Released FJSP operations may also change machine assignment within their eligible sets. Detailed feasibility, closure, and local-repair semantics are given in Appendices A and H.

4 KARMS

Construct ranks feasible operation-machine actions and can use bounded CP lookahead to reconsider uncertain decisions; Destroy-to-Improve then operates on the resulting complete schedule, with the Candidate Head determining *where* to search, the Operation-Selection Transformer determining *what* to release, and CP determining *how* to reconstruct the selected region. The complete pipeline is illustrated in Figure 1 in Appendix B; graph dimensions, relations, model parameters, learning objectives, and training procedures are detailed in Appendices B-J.

4.1 CONSTRUCT

The partial schedule at step t is represented by a graph $\mathcal{G}_t$ containing operation, machine, job, and feasible operation-machine pair nodes. Typed relations encode precedence, job membership, eligibility, and the links between feasible actions and their operations and machines. Each feasible pair corresponds exactly to one action in $\mathcal{A}_t$, so invalid combinations never reach the policy. After every committed decision, the graph is rebuilt to reflect readiness, machine availability, remaining workload, and the updated action set. A heterogeneous GATv2 encoder produces the actor representations; full node features, relations, and dimensions are given in Appendix B.

Construct augments this structural state with advisory scheduling knowledge. Shortest Processing Time (SPT), Longest Processing Time (LPT), (MWKR), Most Operations Remaining (MOR), Earliest Finish Time (EFT), slack, and machine-load criteria support currently feasible operation-machine pairs without directly executing a dispatching rule (Panwalkar & Iskander, 1977; Haupt, 1989; Pinedo, 2008). The learned policy can therefore combine several scheduling principles according to context rather than commit to a fixed heuristic; the same structured-evidence view is used throughout the knowledge-augmented representation.

An offline *Pattern Memory* provides complementary historical evidence. Scheduling states are summarized by progress, machine-load imbalance, remaining work, readiness, bottleneck intensity, job progress, machine flexibility, and remaining-work dispersion, standardized and grouped offline using K-Means with K-Means++ initialization. At inference, nearby clusters indicate scheduling rules and operation-machine decisions that were useful in similar states. Pattern Memory remains advisory and cannot introduce infeasible actions or override the actor. Heuristic definitions, signatures, clustering, retrieval, and graph support relations appear in Appendix C.

The actor assigns a scalar logit z_a to each $a \in \mathcal{A}_t$, with policy $\pi_\theta(a \mid \mathcal{G}_t) = \frac{\exp(z_a)}{\sum_{a' \in \mathcal{A}_t} \exp(z_{a'})}$, where θ denotes the actor parameters. Since normalization is restricted to $\mathcal{A}_t$, every action receiving probability mass is feasible. Construct is trained with Proximal Policy Optimization (PPO) in an actor-critic architecture (Schulman et al., 2017); the critic combines a pooled graph representation with compact topology context. Reward shaping, critic inputs, PPO parameters, rollout storage, warm starts, and training are detailed in Appendices D and J.

After selecting $a_t = (o, m)$, operation o is inserted into the earliest feasible gap on machine m that respects its job predecessor and avoids overlap with scheduled machine intervals. This preserves precedence and capacity while exploiting idle gaps rather than always appending operations to the machine sequence; Appendix D.4 gives the exact insertion and state-update procedure. At final inference, bounded CP lookahead may re-rank Construct decisions. The 3 highest-scoring actions are provisionally committed and a restricted CP model reasons over up to 8 precedence-compatible future operations, using at most 100 ms and one worker per candidate. Its objective combines the restricted CP value with a lower bound on the unmodeled tail and a small actor prior, selecting the usable candidate with the lowest score (if none is usable, the original decision is retained).

Lookahead is thus an inference-time re-ranker rather than a replacement for Construct or the later Repair; its model, scoring, configuration, and fallback rules appear in Appendix E.

4.2 DESTROY-TO-IMPROVE

Destroy-to-Improve receives a complete feasible incumbent S_t. Its heterogeneous graph preserves job precedence and adds the realized machine order, timing, criticality, machine load, and repair context. Complete-solution graph representations have proved useful for learned scheduling improvement (Zhang et al., 2024a; Wang et al., 2025); KARMS uses a separate GATv2 encoder with its own parameters to represent the incumbent for the improvement policy. The complete-schedule graph is detailed in Appendix B.2.

A separate *Repair-Aware Memory*, built from previous repair attempts rather than construction states, provides historical evidence on repair success, makespan improvement, runtime, repair cost, and budget in similar contexts. Repair signatures are normalized and clustered offline using the same K-Means++ principle as Pattern Memory. Retrieved statistics remain contextual evidence and neither execute a repair nor directly select a region; full representation and retrieval details are given in Appendix C.3.

Where to search. KARMS generates structured candidate neighborhoods from the realized incumbent, including critical paths and blocks, bottleneck-machine segments, low-slack regions, late jobs, and job tails. This follows learned local-search and deconstruction approaches that identify promising parts of an incumbent instead of rebuilding the complete schedule (Zhang et al., 2024a; Zago et al., 2026). Because these structures depend on the incumbent, candidates are regenerated after every accepted schedule change; critical-path analysis, candidate families, deduplication, and structural features are detailed in Appendix F. For each candidate C_i, the Candidate Head combines structural descriptors with pooled representations of its proposed and candidate-level effective regions, touched machines, and global schedule context to produce a scalar logit normalized over the candidate set. Candidate-family labels remain provenance metadata rather than categorical network inputs. This implements the learned *where* decision; full representations and policy equations are given in Appendix G.

What to release. A candidate contains several operations that may be reopened. KARMS applies an Operation-Selection Transformer to their graph-derived representations, using self-attention so each score depends on the other operations in the region (Lee et al., 2019). A cardinality-constrained selector maps these scores to the destroy set (Hoppe et al., 2025): $R_t = \mathrm{TopB}\left(\{\theta_o : o \in R_t^{\mathrm{prop}}\}, B_t\right)$, where R_t^{prop} is the proposed operation set, θ_o the contextual score of o, and B_t the repair budget. The module is trained stage-wise using repair-outcome subset supervision, hierarchical PPO, and later CP-guided Fenchel-Young (FY) (Blondel et al., 2020) training, while the final adaptation keeps it fixed. Thus, the Transformer ranks what to reconsider and the budget determines how many operations are initially released; architecture, training, and budget details are given in Appendices G and I.2. Before reconstruction, job-tail closure expands the selected set: releasing $o_{j,k}$ also releases its downstream job operations, giving $R_t^{\mathrm{eff}} = \bigcup_{o_{j,k} \in R_t} \{o_{j,k}, o_{j,k+1}, \ldots, o_{j,n_j-1}\}$. Proposed, selected, and effective regions are therefore distinct, and closure may enlarge the CP neighborhood beyond the operations selected directly by the policy.

How to repair. Reconstruction follows the destroy-and-repair principle of large-neighborhood search (Shaw, 1998), but the learned *where* and *what* decisions determine the neighborhood. CP optimizes only the released region rather than the complete instance (Perron et al., 2023). Let $\mathcal{F}(S_t, R_t^{\mathrm{eff}})$ be the feasible complete schedules preserving incumbent decisions outside R_t^{eff}; the repair problem is $\min_{S \in \mathcal{F}(S_t, R_t^{\mathrm{eff}})} C_{\max}(S)$. JSSP machine assignments remain fixed, whereas released FJSP operations may move to other eligible machines; operations outside the region retain their incumbent start times and machine assignments, with precedence and capacity constraints enforced across the fixed-released boundary. Detailed JSSP/FJSP CP repair formulations, including fixed and released decisions, optional machine intervals, precedence and machine-capacity constraints, makespan minimization, solver-status handling, and bounded search settings, are reported in Appendix H. A feasible reconstruction is then compared with the incumbent. Strict improvements are accepted; a different feasible schedule with equal makespan may also be retained to continue subsequent search from an alternative realization, while true no-ops and worsening repairs are rejected. Failed candidates can be temporarily filtered while the incumbent is unchanged; after an accepted change, the realized structure, candidate bank, and candidate representations are recom-

puted. Destroy-to-Improve is therefore sequential: one candidate, one structured operation set, and one CP repair are resolved before the next improvement decision. Finally, the repair stage is independent of the architecture that produced its starting schedule. It requires only a materialized feasible schedule that can be converted to the common representation and validated, without access to the external scheduler's parameters, logits, embeddings, or training procedure. Candidate Head learning, the structured operation-set role, parameter updates, and stage-wise training are detailed in Appendices I and J.

5 Experimental Setup and Results

KARMS is trained only on synthetic JSSP/FJSP instances up to 30×20. The trained models are then frozen and evaluated without benchmark-specific gradient updates on 242 established JSSP and 258 FJSP instances, including 120 Hurink flexibility variants, for a total of 500 benchmark instances. We further evaluate cross-problem transfer on 3,000 FFSP instances, schedule-source transfer on 675 schedules produced by ReSched and DANIEL, and a paired 2×2 component ablation on 380 matched JSSP/classical-FJSP incumbents. The ablation crosses learned versus heuristic selection with CP versus MWKR reconstruction. Benchmark composition, data splits, evaluation protocols, and the controlled ablation design are detailed in Appendices L and N.7. For JSSP and FJSP, solution quality is measured by $\mathrm{Gap}(\%) = 100\,\frac{C_{\mathrm{method}}-C_{\mathrm{ref}}}{C_{\mathrm{ref}}}$, where C_{ref} is the instance reference makespan. KARMS returns one deterministic solution per instance without test-time sampling or best-of-N selection, whereas published methods retain their reported inference protocols, including sampling or iterative improvement where applicable. Baseline configurations, reference and runtime conventions and evaluation metrics are provided in Appendices L.5 and L.6; published runtimes are used only as computational context because hardware, sampling budgets, and timing scopes differ across methods. Detailed JSSP results and additional FJSP families are reported in Appendix M.

JSSP: The JSSP comparison includes methods with different inference roles. L2D, RL-GNN, ScheduleNet, and ReSched construct schedules directly, whereas L2S, TBGAT, and NDS-S start from complete schedules and perform learned improvement. KARMS combines both roles in a single pipeline through learned construction, bounded CP lookahead, and complete-schedule improvement. The training and inference regimes also differ substantially. L2S trains separate policies for five problem sizes and TBGAT for eight, using 128,000 generated training instances per size; both use 500-step improvement trajectories during training and are evaluated with search budgets of up to 5000 improvement steps. In contrast, KARMS trains shared Construct and Improve models on a mixed JSSP/FJSP curriculum and reuses the same frozen models across all evaluation sizes, with deterministic inference and no best-of-N sampling. We therefore claim size-independent model reuse rather than lower total training compute. Table 1 highlights representative competitive JSSP settings against the strongest available high-budget improvement baseline for each setting. The complete comparison over Taillard, ABZ, FT, LA, SWV, ORB, YN, and 80 additional DMU instances is reported in Appendix M.1. Across the common 162-instance JSSP subset, KARMS achieves an instance-weighted mean gap of 9%, ranking second among the learned methods with complete coverage, behind L2S-5000 (6.90%) but ahead of NDS-S (9.41%), L2S-500 (11.05%), ScheduleNet (16.46%), RL-GNN (20.30%), and L2D (24.64%). Across all overlapping size-level comparisons with the construct-only baselines L2D, RL-GNN, ScheduleNet, and ReSched, KARMS obtains a lower mean gap in 91.9% of cases, showing consistently stronger final solution quality than these construct-only baselines under their reported inference settings.

Table 1: Representative competitive JSSP results. Gap is mean gap (%).The comparator is the strongest reported high-budget improvement baseline among L2S-5000, TBGAT-5000 and NDS-S for the corresponding setting.KARMS time is end-to-end, whereas baseline times report improvement search.

Benchmark	KARMS Gap	KARMS Time	Comparator	Gap	Time
Taillard 15×15	**3.5**	35.5s	NDS-S	4.0	60s
LA 15×15	3.5	35.4s	NDS-S	**3.1**	60s
SWV 20×10	**11.1**	31.6s	TBGAT-5000	**11.1**	2.5m
SWV 50×10	**12.5**	2.1m	TBGAT-5000	14.5	4.9m
Taillard 20×15	6.9	56.1s	TBGAT-5000	**6.3**	2.3m
ORB 10×10	4.6	9.6s	NDS-S	**2.2**	60s

KARMS achieves a lower gap than the strongest listed improvement comparator on Taillard 15×15 and SWV 50×10, and matches TBGAT-5000 on SWV 20×10. It remains close on LA 15×15 and Taillard 20×15, with differences of 0.4 and 0.6 percentage points, respectively, while NDS-S is stronger on ORB 10×10. Across these representative settings, the comparison illustrates that KARMS remains competitive with dedicated high-budget improvement methods while reporting the complete end-to-end Construct-Improve pipeline. Complete Gap/Time results and timing conventions are given in Appendices M.1 and L.6.

FJSP: Table 2 focuses on the four benchmark groups with broad shared published coverage; the complete evaluation additionally includes Barnes, Dauzère-Pérès, Behnke, Fattahi, Kacem, and the remaining Brandimarte instances (Appendix M.2). The comparison methods also differ in their inference role. HGNN, DANIEL, Residual Scheduling, ReSched, and PRISMA-S construct schedules directly, whereas NDS-S and MIStar improve complete schedules. PRISMA uses CP-derived supervision and guidance during training but remains a constructive policy at test time; NDS-S performs iterative learned deconstruction and heuristic reconstruction under a 60 s search budget, while MIStar-400 performs 400 improvement iterations. In contrast, KARMS combines learned construction, bounded CP lookahead, and complete-schedule improvement, reusing the same frozen Construct and Improve models across all FJSP families and producing one final solution through deterministic sequential inference without stochastic sampling or best-of-N selection (Appendix L.5).

Table 2: FJSP results on the shared benchmark families. Entries are mean Gap (%). Bold denotes the lowest gap among the listed learned methods.

Method	MK01-10	Hurink-e	Hurink-r	Hurink-v
HGNN-samp	13.6	16.3	11.4	3.3
DANIEL-samp	13.0	14.4	12.1	3.8
Residual RS+100	15.4	7.9	4.7	0.7
ReSched-samp	6.6	8.1	5.0	0.8
PRISMA-S	10.0	7.7	6.8	1.7
NDS-S	3.5	**2.3**	2.7	3.6
MIStar-400	**3.0**	6.8	**2.4**	**0.2**
KARMS	9.5	5.7	5.2	2.0

Across the 120 Hurink instances, KARMS achieves an instance-weighted mean gap of 4.29%. This is lower than Residual RS+100 (4.43%), ReSched-samp (4.63%), PRISMA-S (5.42%), DANIEL-samp (10.10%), and HGNN-samp (10.33%), while NDS-S (2.87%) and MIStar-400 (3.13%) obtain lower mean gaps. A paired comparison with NDS-S shows no significant difference on the common JSSP-162 subset after Holm correction, whereas NDS-S is significantly better on Hurink-120 (Appendix L.7).

Across the full 258-instance FJSP evaluation, the complete post-construction pipeline reduces the instance-weighted gap from 20.83% after raw Construct to 8.62%, a 58.6% relative reduction. Complete results are reported in Appendix M.2.

Component analysis: We next quantify the quality recovered after initial construction. Table 3 compares raw Construct output with final KARMS schedules over all 500 established JSSP/FJSP evaluations. Because the difference includes the complete post-construction inference pipeline, it is not an isolated Destroy-to-Improve effect. Across all 500 instances, the mean gap falls from 22.71% after raw Construct to 10.34% in the final schedule, with reductions of 12.53 percentage points on JSSP, 13.84 on classical FJSP, and 10.34 on Hurink FJSP. The gain is therefore not concentrated in one benchmark group. Appendix N complements this comparison with the historical Construct progression: the largest improvement followed the transition to structured curriculum and distillation, the mature knowledge representation provided a smaller refinement, and bounded CP lookahead improved most, but not all, evaluated sizes. Because these runs were produced sequentially during development, they describe design progression rather than a strict one-factor causal ablation.

Table 3: Effect of the complete post-construction pipeline. Values are mean reference gaps (%); ΔGap is the reduction from raw Construct to the final schedule.

Set	N	Construct	Final	ΔGap
JSSP	242	24.70	12.17	12.53
Classical FJSP	138	26.23	12.39	13.84
Hurink FJSP	120	14.63	4.29	10.34
All	500	22.71	10.34	12.37

Table 4: Controlled Destroy-to-Improve ablation. Common parent gap: 22.40%. ΔGap is the mean reduction from that parent; W/T/L denotes improved/unchanged/worsened schedules.

Selection	Repair	Final Gap	ΔGap	W/T/L
Learned	CP	**13.52**	**8.89**	349/31/0
Heuristic	CP	20.34	2.07	247/133/0
Learned	MWKR	20.96	1.44	178/202/0
Heuristic	MWKR	20.96	1.45	147/233/0

The diagnostic in Appendix N.6 distinguishes schedules where an improving candidate exists but is not ranked first from schedules whose current candidate bank contains no improving region. The former reflects a ranking limitation, the latter is unresolvable by ranker improvement alone. The paired 2×2 ablation in Table 4 shows strong complementarity between learned selection and CP reconstruction. With CP, learned selection reduces the final gap from 20.34% to 13.52%, whereas learned and heuristic selection are identical under MWKR (20.96% versus 20.96%). Conversely, replacing MWKR with CP improves the gap by 7.44% under learned selection but only 0.62% under heuristic selection. Thus, together with Appendix N.5, the factorial ablation supports the *where-what-how* decomposition of KARMS. The paired difference-of-differences is 6.83% with a

95% bootstrap confidence interval of $[6.08, 7.58]$. A Friedman test across the four matched variants gives $p < 0.001$, and all three paired comparisons against learned selection + CP remain significant after Holm correction ($p_{\text{Holm}} < 0.001$), with rank-biserial effects of 0.950-0.967 (see also Appendix N.7). Finally runtime profiling further shows that CP reasoning accounts for 78.1% of mean E2E time, versus 21.2% for learned policy evaluation (Appendix L.6).

Statistical Comparison with full-instance CP: Full-instance CP provides a strong quality reference. For paired analysis, we use a separate local $1{,}800$ s CP evaluation on the same instances as KARMS. On 242 JSSP instances, CP obtains a mean gap of 1.58% versus 12.17% for KARMS; on 138 FJSP instances, these gaps are 1.02% and 12.39%. Two-sided Wilcoxon signed-rank tests with Pratt handling of zero differences and Holm correction give $p_{\text{Holm}} < 0.001$ for both groups. Defining positive rank-biserial values in favor of KARMS, the effect sizes are $r_{\text{rb}} = -0.997$ for JSSP and $r_{\text{rb}} = -0.982$ for classical FJSP, confirming that long-budget full-instance CP remains stronger in quality. KARMS instead occupies a different quality-time operating point, combining fast learned decisions with bounded local reasoning rather than reproducing a long full-instance solve. Testing conventions, paired results, and external-initializer statistics are reported in Appendix L.7.

Transfer Beyond the Training Distribution: First, FFSP instances are mapped to the same FJSP-compatible representation used by KARMS and evaluated without FFSP-specific training or fine-tuning. We use the MatNet FFSP benchmark sets (Kwon et al., 2021), comprising 1,000 instances each of FFSP20, FFSP50, and FFSP100. Despite the domain shift, the frozen KARMS pipeline reduces mean makespan over MWKR by 22.9%, 16.5%, and 13.4% at the three scales, respectively (17.6% on average), with every pipeline stage contributing further improvement. It also achieves 415, 234, and 225 instance-level wins against MatNet, MatNet$\times 128$, and ReSched-samp, respectively. Complete baseline comparisons and stage-wise transfer results are reported in Appendices L.3 and N.9.

Table 5: Transfer of the frozen Destroy-to-Improve model to external schedules. W/T/L denotes improved/unchanged/worsened schedules relative to the initializer; positive ΔGap denotes mean reduction in BKS gap, with 95% CI.

Initializer	N	Initial gap (%)	Final gap (%)	W/T/L	ΔGap [95% CI]
ReSched, 3 checkpoints	405	9.99	7.80	249/156/0	2.19 [1.82, 2.58]
DANIEL SD1	135	10.82	8.30	94/41/0	2.53 [1.93, 3.18]
DANIEL SD2-mix	135	10.67	7.22	102/33/0	3.46 [2.84, 4.11]

Second, we remove KARMS Construct and apply the frozen Destroy-to-Improve model directly to 675 ReSched and DANIEL schedules. It improves 445 schedules and worsens none: 249/405 across ReSched and 196/270 across DANIEL. Mean BKS gap decreases from 9.99 to 7.80 for ReSched, from 10.82 to 8.30 for DANIEL SD1, and from 10.67 to 7.22 for DANIEL SD2-mix; all reported 95% confidence intervals for the gap reduction exclude zero.

This experiment removes both dependence on the schedule distribution produced by KARMS Construct and access to the external initializer's parameters, logits, or embeddings. The resulting gains there-fore show that the learned repair stage remains useful on independently generated feasible schedules. Complete checkpoint-level paired results are reported in Appendix N.10.

6 Conclusion

KARMS combines learned schedule construction with hierarchical Destroy-to-Improve, where learning determines *where* and *what* to reconsider and CP determines *how* to reconstruct the selected region while preserving feasibility. The intuition here resembles old arguments on the integration of CP with local search (Beck et al., 2011): learning exploits the quality of solution segments as observed during training to carefully allow CP to focus propagation and search on a restricted combinatorial scope. KARMS remains competitive across benchmark families and scales, while the full post-construction pipeline substantially improves the schedules of our learned Construct but also most of those produced by ReSched and DANIEL; the same frozen framework also transfers to FFSP without additional training. A controlled ablation further shows that learned selection and CP repair are complementary, with each providing its main benefit when paired with the other rather than heuristics. Although long-budget full-instance CP remains stronger in final solution quality and other learned methods remain better in several benchmark families, our results provide motivating evidence on the interplay between learned guidance and localized CP reconstruction as a reusable approach across problem sizes, scheduling variants, and starting schedules. Future work will broaden training diversity and the supported classes of scheduling problems.

REPRODUCIBILITY STATEMENT

Our Appendix is deliberately long and detailed to cover all reproducibility aspects and support the review process. To facilitate its use, there are detailed references to specific, well-targeted, parts of the Appendix throughout the text. As an outline, the problem formulation and graph representations are listed in Appendices A and B; the Construct policy, CP lookahead, Destroy-to-Improve policy, structured operation selection, and CP repair are described in Appendices D-H. Learning objectives and the stage-wise training procedure are documented in Appendices I and J, including the training of candidate and operation-selection components. Model dimensions, solver settings, hardware, software, and the final inference configuration are reported in Appendix K. Appendix L specifies the benchmark sets, reference solutions, baseline protocols, evaluation metrics, runtime measurement, and statistical procedures, while complete benchmark and ablation results are provided in Appendices M and N.

AI USE STATEMENT

During the writing of this article, the authors used generative AI tools as supporting aids for language translation, improving writing clarity and flow, organizing the presentation of the manuscript, and assisting with preliminary literature searches. Any AI-assisted material included into the paper was reviewed and revised by the authors. The authors take responsibility for the scientific content, results, claims, and final manuscript.

REFERENCES

Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988. doi: 10.1287/mnsc.34.3.391.

David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991. doi: 10.1287/ijoc.3.2.149.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035. SIAM, 2007. doi: 10.1145/1283383.1283494.

Ioannis Avgerinos, Christos Katrinakis, Andreas Ktenidis, Aggelos Ioannis Lagos, Ioannis Mourtos, and Georgios Zois. Enhancing scalability in distributed flexible flowshop scheduling: A hybrid rl-cp approach. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 16595 of *Lecture Notes in Computer Science*, pp. 36–53. Springer, 2026. doi: 10.1007/978-3-032-27242-3_3.

J. Wesley Barnes and John B. Chambers. Flexible job shop scheduling by tabu search. Technical Report ORP96-09, Graduate Program in Operations Research and Industrial Engineering, The University of Texas at Austin, Austin, TX, 1996.

J. Christopher Beck, T. K. Feng, and Jean-Paul Watson. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23(1):1–14, 2011. doi: 10.1287/ijoc.1100.0388.

Dennis Behnke and Martin Josef Geiger. Test instances for the flexible job shop scheduling problem with work centers. Technical Report RR 12-01-01, Helmut Schmidt University, 2012.

Mathieu Blondel, André F. T. Martins, and Vlad Niculae. Learning with fenchel-young losses. *Journal of Machine Learning Research*, 21(35):1–69, 2020.

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993. doi: 10.1007/BF02023073.

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

M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1): 200–210, 2013. doi: 10.1016/j.eswa.2012.07.021.

Inguk Choi, Woo-Jin Shin, Sang-Hyun Cho, and Hyun-Jung Kim. Towards generalizable multi-policy optimization with self-evolution for job scheduling. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0828.

Stéphane Dauzère-Pérès and Jan Paulli. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70:281–306, 1997. doi: 10.1023/A:1018930406487.

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. doi: 10.1016/S0377-2217(97)00019-2.

Imanol Echeverria, Maialen Murua, and Roberto Santana. Leveraging constraint programming in a deep learning approach for dynamically solving the flexible job-shop scheduling problem. *Expert Systems with Applications*, 265:125895, 2025. doi: 10.1016/j.eswa.2024.125895.

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

Herbert Fisher and Gerald L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In John F. Muth and Gerald L. Thompson (eds.), *Industrial Scheduling*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

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

Kuo-Hao Ho, Jui-Yu Cheng, Ji-Han Wu, Fan Chiang, Yen-Chi Chen, Yuan-Yu Wu, and I-Chen Wu. Residual scheduling: A new reinforcement learning approach to solving job shop scheduling problem. *IEEE Access*, 12:14703–14718, 2024. doi: 10.1109/ACCESS.2024.3357969.

Heiko Hoppe, Léo Baty, Louis Bouvier, Axel Parmentier, and Maximilian Schiffer. Structured reinforcement learning for combinatorial decision-making. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-5780.

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

Zangir Iklassov, Dmitrii Medvedev, Ruben Solozabal Ochoa de Retana, and Martin Takac. On the study of curriculum learning for inferring dispatching policies on the job shop scheduling. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, pp. 5350–5358. International Joint Conferences on Artificial Intelligence Organization, 2023. doi: 10.24963/ijcai.2023/594.

Alexandre Jesus, Arthur Corrêa, Miguel Vieira, Catarina Marques, Cristóvão Silva, and Samuel Moniz. Enhancing multi-agent deep reinforcement learning for flexible job-shop scheduling through constraint programming. *Computers & Operations Research*, 190:107428, 2026. doi: 10.1016/j.cor.2026.107428.

Imed Kacem, Slim Hammadi, and Pierre Borne. Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic. *Mathematics and Computers in Simulation*, 60(3–5):245–276, 2002. doi: 10.1016/S0378-4754(02)00019-8.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

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.

Stephen Lawrence. Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1984.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3744–3753. PMLR, 2019.

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

Fei Ni, Jianye Hao, Jiawen Lu, Xialiang Tong, Mingxuan Yuan, Jiahui Duan, Yi Ma, and Kun He. A multi-graph attributed reinforcement learning based optimization algorithm for large-scale hybrid flow shop scheduling problem. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 3441–3451, 2021. doi: 10.1145/3447548.3467135.

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

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

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, 2021b. doi: 10.1080/00207543.2020.1870013.

Laurent Perron, Frédéric Didier, and Steven Gay. The cp-sat-lp solver. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics*, pp. 3:1–3:2. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.CP.2023.3.

Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, New York, 3 edition, 2008.

Zhaojun Qin and Yuqian Lu. Knowledge graph-enhanced multi-agent reinforcement learning for adaptive scheduling in smart manufacturing. *Journal of Intelligent Manufacturing*, 36:5943–5966, 2025. doi: 10.1007/s10845-024-02494-0.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming—CP98*, volume 1520 of *Lecture Notes in Computer Science*, pp. 417–431. Springer, 1998. doi: 10.1007/3-540-49481-2_30.

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.

Nicolas Sonnerat, Pengming Wang, Ira Ktena, Sergey Bartunov, and Vinod Nair. Learning a large neighborhood search algorithm for mixed integer programs. *CoRR*, abs/2107.10201, 2021.

Robert H. Storer, S. David Wu, and Renzo Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992. doi: 10.1287/mnsc.38.10.1495.

Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. doi: 10.1016/0377-2217(93)90182-M.

Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. An end-to-end reinforcement learning approach for job-shop scheduling problems based on constraint programming. In *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling*, pp. 614–622, 2023. doi: 10.1609/icaps.v33i1.27243.

Jiaqi Wang, Zhiguang Cao, Peng Zhao, Rui Cao, Yubin Xiao, Yuan Jiang, and You Zhou. Learning memory-enhanced improvement heuristics for flexible job shop scheduling. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2776.

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. doi: 10.1109/TNNLS.2023.3306421.

Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Learning large neighborhood search policy for integer programming. In *Advances in Neural Information Processing Systems*, volume 34, pp. 30075–30087. Curran Associates, Inc., 2021.

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.

Takeshi Yamada and Ryohei Nakano. A genetic algorithm applicable to large-scale job-shop problems. In Reinhard Männer and Bernard Manderick (eds.), *Parallel Problem Solving from Nature 2*, pp. 283–292. North-Holland, 1992.

Davide Zago, André Hottung, Fynn Martin Gilbert, Rossella Cancelliere, and Kevin Tierney. Reinforcement learning guided neural deconstruction search for flexible job scheduling. *Machine Learning*, 115(8), 2026. doi: 10.1007/s10994-026-07116-9.

Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. 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*, 2024a.

Cong Zhang, Zhiguang Cao, Yaoxin Wu, Wen Song, and Jing Sun. Learning topological representations with bidirectional graph attention network for solving job shop scheduling problem. In *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*, volume 244 of *Proceedings of Machine Learning Research*, pp. 4192–4208. PMLR, 2024b.

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 on Web Conference 2025*, pp. 4089–4100, 2025. doi: 10.1145/3696410.3714616.

APPENDIX

A Detailed Problem Formulation and Notation

A.1 Unified Scheduling Notation

KARMS uses a common FJSP-compatible representation for all scheduling instances. Let $\mathcal{J} = \{0, \ldots, J-1\}$ denote the set of J jobs, with job $j \in \mathcal{J}$ containing the ordered operations $O_j = (o_{j,0}, \ldots, o_{j,n_j-1})$. We identify each operation by $o = (j, k)$, where j is the job index and k its position within the job; jobs, operations, and machines are indexed from zero in the implementation.

Let $\mathcal{M} = \{0, \ldots, M-1\}$ be the set of M machines. Each operation o has an eligible-machine set $\mathcal{M}_o \subseteq \mathcal{M}$, with processing time $p_{o,m} \in \mathbb{Z}_{>0}$ for each $m \in \mathcal{M}_o$; ineligible operation-machine pairs are not represented. Thus, $|\mathcal{M}_o| = 1$ in JSSP, whereas FJSP operations may have multiple eligible machines.

For a completed schedule, $m(o)$ denotes the machine assigned to operation o, and s_o and e_o its start and completion times, with $e_o = s_o + p_{o,m(o)}$ under non-preemptive processing. A materialized schedule stores the job and operation identifiers, assigned machine, start time, and completion time of every operation, together with the realized operation order on each machine used by Destroy-to-Improve. For tensor processing, (j, k) is additionally mapped to a padded flat operation index used only for storage and graph tensors; throughout the paper we retain the notation $o = (j, k)$.

Table 6: Unified notation used by KARMS.

Symbol	Meaning
$\mathcal{J}$	set of jobs
J	number of jobs
O_j	ordered operations of job j
$o = (j, k)$	operation k of job j
$\mathcal{M}$	set of machines
$\mathcal{M}_o$	machines eligible for operation o
$p_{o,m}$	processing time of o on machine m
$m(o)$	machine assigned to operation o
s_o	start time of operation o
e_o	completion time of operation o
$C_{\max}$	makespan of the schedule

A.2 JSSP AND FJSP CONSTRAINTS

JSSP and FJSP share the same precedence, machine-capacity, and non-preemption constraints. For consecutive operations $o_{j,k}$ and $o_{j,k+1}$ of job j, $s_{j,k+1} \geq e_{j,k}, \quad e_{j,k} = s_{j,k} + p_{o_{j,k},m(o_{j,k})}$, so an operation cannot start before its predecessor completes. Operations assigned to the same machine must have disjoint processing intervals: $[s_o, e_o) \cap [s_{o'}, e_{o'}) = \varnothing$. Construction enforces this condition through earliest-feasible insertion into the current machine schedule, whereas the CP formulation uses interval variables and one `NoOverlap` constraint per machine. Processing is non-preemptive and $s_o, e_o \in \mathbb{Z}_{\geq 0}$.

In JSSP, $|\mathcal{M}_o| = 1$, so machine assignment is fixed and only sequencing remains. In FJSP, let $x_{o,m} \in \{0, 1\}$ indicate assignment of operation o to eligible machine m. Exactly one alternative is selected: $\sum_{m \in \mathcal{M}_o} x_{o,m} = 1$, with the selected machine determining $p_{o,m}$. In the CP model, eligible alternatives are represented by optional intervals, exactly one of which is activated.

The benchmark formulation does not include release dates, due dates, setup or transportation times, machine calendars, blocking, no-wait constraints, or precedence relations beyond each job's technological order. Local CP repair may additionally fix operations outside the released region to their incumbent decisions; this restricts the repair search space but does not alter the underlying scheduling problem.

A.3 FFSP TRANSFER REPRESENTATION

For the FFSP transfer experiment, the MatNet benchmark representation is converted to the same FJSP-compatible interface used by KARMS. The supported instances contain three ordered processing stages with four parallel machines per stage, for twelve physical machines in total, and every job visits all three stages in the same order.

For job j and stage $r \in \{0, 1, 2\}$, the adapter creates one operation $o_{j,r}$ whose eligible machines are the four machines of that stage, $\mathcal{M}_r = \{4r, 4r+1, 4r+2, 4r+3\}$. Let $p_{j,r,m}$ denote the processing time of job j at stage r on machine m. The operation is represented as $o_{j,r} \to \{(m, p_{j,r,m}) \mid m \in \mathcal{M}_r\}$, and stage order becomes ordinary job precedence, $o_{j,0} \prec o_{j,1} \prec o_{j,2}$.

After conversion, the instance is handled as an FJSP instance rather than by a separate FFSP solver or policy. The mapping preserves stage order, stage-specific machine eligibility, machine-dependent processing times, machine non-overlap, non-preemption, and makespan, and introduces no FFSP-specific constraint after canonicalization.

This representation covers the FFSP benchmark setting considered here: jobs share the same ordered stages, do not skip or revisit stages, and there are no setup or transportation times. The experiment therefore evaluates transfer to this supported FFSP representation rather than to every flexible flow-shop variant.

A.4 Feasibility and Makespan Objective

Under the constraints above, a schedule is feasible when every operation is assigned to exactly one eligible machine, job precedence and machine non-overlap are respected, and processing is non-preemptive. Compactly, $m(o) \in \mathcal{M}_o, \quad s_{j,k+1} \geq e_{j,k}, \quad [s_o, e_o) \cap [s_{o'}, e_{o'}) = \varnothing$ for distinct operations o, o' assigned to the same machine.

The objective throughout is makespan minimization: $\min C_{\max}, \quad C_{\max} = \max_{o \in \mathcal{O}} e_o, \quad \mathcal{O} = \bigcup_{j \in \mathcal{J}} O_j$. Reinforcement-learning rewards, candidate and CP-lookahead scores, memory statistics, and training losses guide learning or search but do not modify this objective. Likewise, fixing incumbent decisions outside a local repair region restricts the CP search space without changing the definitions of feasibility or makespan.

B Knowledge-Augmented Graph Representation

KARMS represents scheduling states using heterogeneous graphs implemented with PyTorch Geometric `HeteroData`. Construct and Destroy-to-Improve operate on partial and complete schedules, respectively, and therefore use separate graphs with the same GATv2-based encoding design. Figure 1 summarizes how these representations support feasible construction, CP lookahead, learned destroy selection, and local CP reconstruction.

B.1 Construct Graph

At construction step t, let $\mathcal{F}_t$ denote the frontier of unscheduled operations whose job predecessors have already been scheduled. For each $o \in \mathcal{F}_t$ with eligible-machine set $\mathcal{M}_o$, the feasible action set is $\mathcal{A}_t = \{(o, m) \mid o \in \mathcal{F}_t,\ m \in \mathcal{M}_o\}$, where (o, m) schedules o on eligible machine m, and $|\mathcal{A}_t| = \sum_{o \in \mathcal{F}_t} |\mathcal{M}_o|$.

The Construct graph contains operation, machine, job, feasible operation-machine pair, heuristic-rule, and Pattern-Memory nodes. Let J denote the number of jobs, M the number of machines, and $K_{\max}$ the largest number of operations in any job of the current instance. Operation tensors are padded to $JK_{\max}$ positions for batched graph processing, while only positions corresponding to real operations participate in scheduling decisions.

Table 7: Node types in the Construct graph. The raw dimension denotes the number of scalar values used before neural projection.

Node type	Number of nodes	Raw dimension	Meaning
`op`	$JK_{\max}$	8	operation state
`machine`	M	8	machine state
`job`	J	8	job state
`pair`	$\lvert\mathcal{A}_t\rvert$	8	feasible (o, m) action
`rule`	R	8	heuristic evidence
`memory`	1	8	Pattern-Memory context

Here, R is the number of heuristic rules used by Construct. Each `pair` node corresponds exactly to one action in $\mathcal{A}_t$; invalid operation-machine combinations are never inserted, so the pair-node set itself defines the action space. Pair nodes are linked to their operation and machine, with reverse

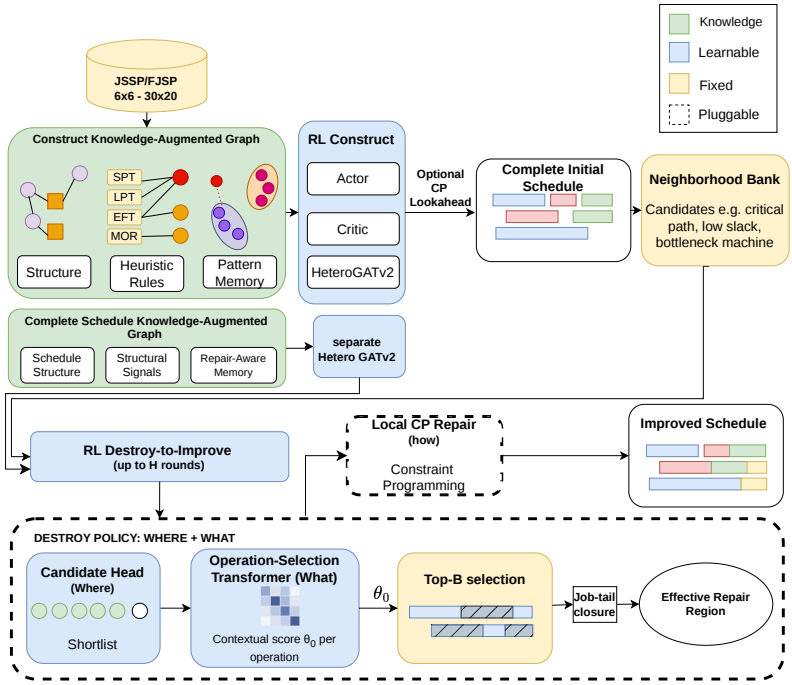


Figure 1: Overview of the KARMS construct-destroy-repair pipeline. Construct builds a feasible schedule from operation-machine decisions, while Destroy-to-Improve selects where and what to reopen before CP reconstructs the released region. Pattern Memory supports construction and Repair-Aware Memory provides historical repair context.

relations allowing information to flow back toward the pair representation. Heuristic rules and Pattern Memory add support relations to feasible pair nodes; their scoring and retrieval are described in Appendix C.

B.2 Complete-Schedule Destroy-to-Improve Graph

Destroy-to-Improve operates on a complete incumbent S_t. Its graph represents the original job structure together with current machine assignments, realized machine order, timing information, and repair context. Operation, machine, and job nodes describe the incumbent, while rule and Repair-Aware Memory information provide contextual evidence. Candidate neighborhoods are stored in `HeteroData` when the candidate-based Destroy Policy is active, but candidate nodes are excluded from the shared schedule GATv2 encoder and are processed later by the Candidate Head using embeddings produced from the encoded schedule.

Current machine assignment is represented by operation-to-machine relations. Realized machine order uses separate directed operation-to-operation relations between consecutive operations on the same machine, whereas technological job precedence retains its own relation type. Keeping these relations distinct allows the encoder to distinguish job-imposed precedence from incumbent machine ordering. Schedule-dependent timing, criticality, machine load, and remaining job-tail work are encoded as node features. Higher-level structures such as critical blocks, bottleneck regions, and candidate neighborhoods are derived from the incumbent rather than introduced as additional schedule node types; their construction is described in Appendix F.

B.3 Node Types and Features

The principal schedule node types use eight raw scalar features before projection into the common hidden representation.

Construct operation features. For an operation o, the Construct feature vector is $x_o^{\mathrm{con}} = [1,\ q_o^{\mathrm{sched}},\ q_o^{\mathrm{ready}},\ r_o^{\mathrm{pos}},\ p_o^{\mathrm{min}},\ n_o^{\mathrm{elig}},\ w_o^{\mathrm{rem}},\ c_o^{\mathrm{pred}}]$.

Here, $x_o^{\mathrm{con}} \in \mathbb{R}^8$; $q_o^{\mathrm{sched}} \in \{0,1\}$ indicates whether o has been scheduled, $q_o^{\mathrm{ready}} \in \{0,1\}$ whether it is currently ready, r_o^{pos} its normalized position in its job, $p_o^{\mathrm{min}} = \min_{m \in \mathcal{M}_o} p_{o,m}$ its minimum processing time over eligible machines, $n_o^{\mathrm{elig}} = |\mathcal{M}_o|$ its number of eligible machines, w_o^{rem} the minimum remaining processing workload of its job, and c_o^{pred} the completion time of its immediate job predecessor. Continuous scheduling quantities are normalized over the real operations of the current state before neural encoding.

Construct machine features. A machine node contains its normalized current availability time, expected remaining workload, and expected number of remaining operations. The remaining five positions are zero-filled; availability is the earliest time at which the machine can process another operation under the current partial schedule.

Construct job features. A job node contains a constant value equal to one, the fraction of its operations already scheduled, the fraction still unscheduled, its normalized number of operations, normalized remaining minimum workload, normalized latest completion time, and normalized job identifier. The final entry is zero-filled.

Construct pair features. For feasible action $a = (o, m)$, the pair-node representation is $x_a^{\mathrm{pair}} = [1,\ r_o^{\mathrm{id}},\ r_m^{\mathrm{id}},\ p_{o,m}^{\mathrm{norm}},\ \hat{s}_{o,m},\ \hat{e}_{o,m},\ c_o^{\mathrm{job}},\ c_m^{\mathrm{ready}}]$.

Here, $x_a^{\mathrm{pair}} \in \mathbb{R}^8$; r_o^{id} and r_m^{id} are normalized operation and machine identifiers, $p_{o,m}^{\mathrm{norm}}$ is the normalized processing time, $\hat{s}_{o,m}$ and $\hat{e}_{o,m}$ are the estimated start and completion times if the action is selected, c_o^{job} is the earliest time permitted by job precedence, and c_m^{ready} is the current ready time of machine m.

Rule and Pattern-Memory features. Rule nodes use eight-dimensional sparse representations encoding rule identity. Pattern Memory is represented by an eight-dimensional retrieved context associated with the current construction state. Rule definitions, memory signatures, clustering, and retrieval are described in Appendix C.

Complete-schedule operation features. In Destroy-to-Improve, each operation uses an eight-dimensional representation containing a constant value, start time normalized by the current makespan, completion time normalized by the current makespan, normalized position within its job, normalized processing duration, a criticality value derived from slack, the load of its assigned machine relative to the most heavily loaded machine, and normalized remaining job-tail workload.

A complete-schedule machine node contains a constant value, busy time divided by the current makespan, idle time divided by the current makespan, the fraction of real operations assigned to that machine, and a normalized machine identifier. Its final three entries are zero-filled. A complete-schedule job node contains a constant value, job completion time divided by the current makespan, the fraction of all real operations belonging to the job, total job workload divided by the current makespan, and a normalized job identifier. Its final three entries are zero-filled.

Candidate descriptors. Candidate neighborhoods additionally carry an 18-dimensional structural feature vector describing properties of their proposed and candidate-level effective regions. Candidate nodes are not encoded by the shared GATv2 network, so these descriptors are consumed later by the Candidate Head. Their definitions and the distinction between proposed and effective candidate regions are given in Appendix F.6.

B.4 Relation Types and Edge Features

Table 8 summarizes the principal typed relations in the Construct graph.

Eligibility relations describe which machines can process an operation independently of readiness, whereas pair relations represent the concrete feasible actions available at the current construction step. Rule-to-pair and memory-to-pair support relations carry four scalar attributes: normalized

Table 8: Principal typed relations in the Construct graph.

Source	Relation	Destination
op	precedes	op
op	eligible	machine
machine	processes	op
op	belongs_to	job
job	has_operation	op
pair	pair_for_operation	op
op	has_pair	pair
pair	pair_uses_machine	machine
machine	machine_used_by_pair	pair
rule	recommends	pair
memory	supports	pair

recommendation rank, normalized support score, confidence, and a validity indicator for the current scheduling state. These attributes are transformed by separate learned multilayer perceptrons before action scoring and are not supplied directly as edge features to the GATv2 convolutions.

For Destroy-to-Improve, the encoded schedule contains separate operation-to-operation relations for job precedence and realized machine order, together with job-membership, current machine-assignment, rule-support, and memory-support relations. Candidate membership is additionally recorded through candidate-to-operation and candidate-to-machine relations describing proposed operations, candidate-level effective operations, and touched machines. These candidate relations remain part of the stored heterogeneous representation but are excluded from the shared schedule GATv2 message-passing path and are used later to form candidate representations, as described in Appendix G.

B.5 Heterogeneous GATv2 Encoder

Both KARMS stages use a heterogeneous GATv2 encoder (Brody et al., 2022). For every active typed relation, the encoder creates a separate `GATv2Conv` transformation, and relation-specific messages arriving at the same destination node type are combined by summation.

Each encoded schedule node begins with an eight-dimensional raw feature vector. A learned input projection maps this vector from dimension 8 to 32. Two heterogeneous message-passing layers then operate at width 32, with each relation-specific GATv2 convolution receiving and producing a 32-dimensional representation.

Each relation-specific convolution uses one attention head with `concat=False`. The aggregated message is added to the representation entering the layer through a residual connection, followed by a ReLU activation, where $\mathrm{ReLU}(x) = \max(0, x)$ for scalar x. No dropout or graph normalization is used in the final encoder configuration.

A final learned projection maps the 32-dimensional hidden representation to a 16-dimensional output embedding. For graph node v, the final representation is $h_v \in \mathbb{R}^{16}$, where $\mathbb{R}^{16}$ denotes the 16-dimensional real vector space.

The heuristic- and memory-support attributes described above are not passed directly into `GATv2Conv`; they are processed separately and contribute to the downstream decision modules together with the structural node embeddings.

For Destroy-to-Improve, candidate nodes are removed before the shared schedule graph is encoded. Let $\mathcal{O}$ denote the set of real operations in the incumbent schedule. A global schedule representation is obtained by mean pooling their final embeddings, $h_G = \frac{1}{|\mathcal{O}|} \sum_{o \in \mathcal{O}} h_o$, where $h_o \in \mathbb{R}^{16}$ is the encoded representation of operation o, $|\mathcal{O}|$ is the number of real operations, and $h_G \in \mathbb{R}^{16}$ is the resulting global schedule embedding. The Candidate Head subsequently combines this schedule representation with candidate-specific structural and pooled representations, as detailed in Appendix G.

Table 9: GATv2 configuration used by the Construct and Destroy-to-Improve encoders.

Parameter	Final value
Raw node-feature dimension	8
Input-projection dimension	32
Message-passing layers	2
Hidden dimension	32
Attention heads per relation	1
Per-head output dimension	32
Head concatenation	disabled
Relation aggregation	sum
Final node-embedding dimension	16
Residual connections	enabled
Activation	ReLU
Dropout	0
Normalization	none

B.6 Separate Construct and Destroy-to-Improve Encoders

Construct and Destroy-to-Improve use the same GATv2 architecture but separate model instances and separate checkpoints; their learned parameters are not shared. Both follow the dimensional sequence $8 \rightarrow 32 \rightarrow 32 \rightarrow 16$, where 8 is the raw schedule-node feature width, 32 is the hidden width used by the two heterogeneous message-passing layers, and 16 is the final node-embedding dimension. Construct encodes a partial schedule and uses the resulting representations to score currently feasible operation-machine actions, whereas Destroy-to-Improve encodes a complete incumbent and supplies its operation, machine, job, and global representations to the candidate-ranking and operation-set modules.

B.7 Graph Updates

The Construct graph is rebuilt after every committed scheduling action because operation readiness, machine availability, remaining workload, and the feasible action set change. The next graph therefore recomputes the frontier $\mathcal{F}_{t+1}$, action set $\mathcal{A}_{t+1}$, pair nodes, dynamic node features, heuristic support, and Pattern-Memory evidence. Consequently, the graph contains exactly $|\mathcal{A}_t|$ pair nodes at step t and $|\mathcal{A}_{t+1}|$ pair nodes after the decision.

The Destroy-to-Improve graph is tied to the current complete incumbent schedule. After an accepted repair, the next iteration reconstructs the schedule-dependent graph from the updated solution, including machine assignments, realized machine order, timing information, loads, and structural diagnostics. Candidate neighborhoods are then regenerated from the new state without inserting candidate nodes into the shared GATv2 message-passing computation.

The detailed procedures for candidate generation, failure-aware filtering, learned candidate and operation-set selection, and CP reconstruction are given in Appendices F, G, and H, respectively.

C Statistical Memory and Domain Knowledge

C.1 Scheduling Heuristics and Structural Rules

KARMS uses heuristic and structural information as advisory evidence rather than hard decision rules. During Construct, each heuristic scores the currently feasible operation-machine pairs. For a feasible pair (o, m), let $p_{o,m}$ denote the processing time of o on eligible machine m, $\hat{s}_{o,m}$ its estimated start time, and $\hat{e}_{o,m} = \hat{s}_{o,m} + p_{o,m}$ its estimated completion time. The final Construct configuration uses Shortest Processing Time (SPT), Longest Processing Time (LPT), Most Work Remaining (MWKR), Most Operations Remaining (MOR), Earliest Finish Time (EFT), Slack, and

Least Loaded Machine (LLM), with scores

$$\begin{aligned} \mathrm{SPT}(o,m) &= -p_{o,m}, & \mathrm{LPT}(o,m) &= p_{o,m}, \\ \mathrm{MWKR}(o,m) &= W_o^{\mathrm{rem}}, & \mathrm{MOR}(o,m) &= N_o^{\mathrm{rem}}, \\ \mathrm{EFT}(o,m) &= -\hat{e}_{o,m}, & \mathrm{SLACK}(o,m) &= -\max(0, \hat{s}_{o,m} - c_o^{\mathrm{pred}}), \\ \mathrm{LLM}(o,m) &= -B_m. \end{aligned}$$

Here, W_o^{rem} is the minimum remaining processing workload of the job containing o, N_o^{rem} the number of operations still remaining in that job, c_o^{pred} the completion time of the job predecessor of o, and B_m the accumulated busy time of machine m. The negative signs used by SPT, EFT, Slack, and LLM convert their natural lower-is-better interpretation into the common higher-is-better scoring convention.

Each heuristic evaluates all currently feasible operation-machine pairs and retains its three highest-ranked recommendations, summarized in Table 10. These recommendations are attached to the graph as support relations; they neither remove feasible actions nor force a particular decision, so the Construct actor may select an action not preferred by any individual heuristic.

Table 10: Scheduling heuristics used as advisory knowledge in Construct. Each rule ranks currently feasible operation-machine pairs; the learned actor remains responsible for the final action selection.

Rule	Priority quantity	Preference
SPT	Processing time	Shorter operation
LPT	Processing time	Longer operation
MWKR	Remaining job work	Larger remaining work
MOR	Remaining operations	More remaining operations
EFT	Earliest finish time	Earlier completion
SLACK	Insertion slack	Lower slack
Least-loaded machine	Current machine load	Lower machine load

Destroy-to-Improve uses complete-schedule structural evidence instead: critical-path participation, machine bottleneck load, low slack, long processing duration, and late-job behavior. These quantities identify potentially useful repair locations rather than prescribing reconstruction; candidate generation and the use of these regions are described in Appendix F.

For both stages, retained recommendations use four support values. Let r be the zero-based rank among n retained recommendations, with $r = 0$ denoting the best item. Its normalized rank is $r_{\mathrm{norm}} = \frac{r}{\max(1,n-1)}$. Let s be the raw heuristic score, and $s_{\min}$ and $s_{\max}$ the smallest and largest retained scores for the same rule. The normalized score is $s_{\mathrm{norm}} = \frac{s-s_{\min}}{\max(s_{\max}-s_{\min},10^{-6})}$. The confidence is $c = 1 - \frac{1}{2} r_{\mathrm{norm}}$, and the validity indicator v is one when the recommendation is valid in the current scheduling state. The resulting support vector $[r_{\mathrm{norm}}, s_{\mathrm{norm}}, c, v]$ is processed by a small learned multilayer perceptron before being combined with the representation of the supported action or repair choice.

C.2 Pattern Memory

Pattern Memory provides Construct with statistical information from previously observed construction states. Each state is summarized by the eight-dimensional signature $x_t = [r_t^{\mathrm{sched}}, d_t^{\mathrm{load}}, r_t^{\mathrm{work}}, r_t^{\mathrm{ready}}, b_t^{\mathrm{mach}}, d_t^{\mathrm{job}}, f_t^{\mathrm{elig}}, d_t^{\mathrm{work}}]$, where t is the current construction step and $x_t \in \mathbb{R}^8$. The scheduled-operation ratio r_t^{sched} is the number of real operations already scheduled divided by the total number of real operations; the remaining-work ratio r_t^{work} is the sum of minimum eligible processing times of all unscheduled operations divided by the corresponding full-instance sum; and the ready-operation ratio r_t^{ready} is the number of currently available operations divided by the number of jobs.

For each machine, pressure combines its already scheduled busy time with estimated future workload from unscheduled operations, whose processing work is distributed across eligible machines. The load-imbalance value d_t^{load} is the population standard deviation of machine pressure divided by the absolute mean pressure plus one, and b_t^{mach} is the largest machine pressure divided by the same

denominator. The job-completion spread d_t^{job} is the population standard deviation of job completion ratios, where each ratio is the number of scheduled operations of a job divided by its total number of operations. The flexibility value f_t^{elig} is the mean fraction of machines eligible for each unscheduled operation. Finally, d_t^{work} is the population standard deviation of per-job remaining minimum workload divided by its absolute mean plus one. A spread of zero is used when only one value is available.

Before clustering, each signature component is standardized: $z_t = \frac{x_t - \mu}{\sigma}$, where $z_t \in \mathbb{R}^8$, $\mu \in \mathbb{R}^8$ contains component-wise means estimated from the offline memory data, and $\sigma \in \mathbb{R}^8$ the corresponding population standard deviations. Each component of σ is lower-bounded by 10^{-6} to avoid division by zero, and the same stored μ and σ are used during inference.

The standardized signatures are grouped offline using K-Means++ with Euclidean distance. K-Means++ initializes centroids so that states far from already selected centroids are more likely to become new initial centroids; clustering then alternates between assigning each state to its nearest centroid and recomputing centroids. Each cluster stores the number of observations, average rollout return, average makespan change, heuristic-rule support, and rule-quality information, associating regions of construction-state space with heuristic behavior useful in similar previously observed states.

At inference, the standardized current signature is compared with all stored centroids and the two nearest clusters are retrieved. If d_i is the distance to retrieved cluster i, its weight is $w_i = \frac{\exp(-d_i/\tau)}{\sum_{j=1}^{2}\exp(-d_j/\tau)}$, where $\tau = 1$ and w_i is the normalized contribution of cluster i, so smaller distances receive larger weights. The two clusters' statistics are combined using these weights and converted into support for currently feasible operation-machine pairs. Pattern Memory therefore changes the evidence available to Construct but does not directly select an action. Their individual scheduling contribution is evaluated through a controlled two-factor ablation in Appendix N.8.

C.3 Repair-Aware Memory

Repair-Aware Memory provides corresponding historical context during Destroy-to-Improve. It is built offline from repair attempts and remains read-only during inference. Each repair attempt is represented by a twelve-dimensional signature produced by `repair_signature_from_diagnostics` in `ImproveRepairClusterMemory`. Its first eight entries are criticality, machine-load ratio, late-job score, slack pressure $1/(1 + \text{slack})$, normalized duration, normalized machine index, normalized operation index, and hint-size fraction. The final four entries are a one-hot encoding of the repair channel: `seed`, `machine_block`, `job_successor`, or `critical_region`. These channels represent, respectively, a generic seed-centered repair, a region associated with machine congestion, a region extending through job successors, and a region associated with critical schedule structure.

Stored cluster statistics include the number of observations, historical repair score, strict-improvement frequency, average makespan improvement, runtime, repair cost, and repair-budget information. A repair is successful only when it produces a strictly smaller makespan than its starting schedule; finding a feasible equal-makespan schedule therefore does not count as a successful repair.

Repair signatures are standardized component by component using statistics estimated during offline memory construction and clustered with K-Means++ under Euclidean distance. During retrieval, the current repair signature is compared with stored centroids, the two nearest clusters are selected, and their statistics are blended with the same soft distance weighting used by Pattern Memory. The retrieved statistics are conditioned on the relevant repair channel and provide evidence about, for example, historical improvement frequency, repair cost, and typical gain in similar regions. Repair-Aware Memory informs Destroy-to-Improve but does not execute repair or replace candidate selection or CP reconstruction.

C.4 Memory Clustering and Offline Selection

The number of clusters used by each memory is selected offline rather than fixed manually or tuned on the final benchmark evaluation. Pattern Memory uses the eight-dimensional state signature above,

while Repair-Aware Memory uses the twelve-dimensional repair signature defined in Appendix C.3. For each memory, a discrete set of clustering granularities is evaluated using standard internal validation criteria.

Table 11: Offline selection of memory clustering granularity. The automatic rule selects the value with the highest Silhouette score; the elbow criterion is used only when Silhouette is unavailable. Davies-Bouldin values are reported as an additional diagnostic but do not determine the automatic choice.

Memory	Tested k	Silhouette choice	DB choice	Selected k
Pattern Memory	$\{32, 48, 64, 96, 128, 192, 256\}$	32	32	32
Repair-Aware Memory	$\{16, 24, 32, 48, 64, 96\}$	16	96	16

For Pattern Memory, both Silhouette and Davies-Bouldin favor $k = 32$, with Silhouette 0.359 and Davies-Bouldin 0.907 for the full quality-weighted memory. For Repair-Aware Memory, the criteria disagree: Silhouette favors $k = 16$, whereas Davies-Bouldin decreases toward $k = 96$. The implemented automatic rule prioritizes the maximum Silhouette score, using smaller k to break ties, and therefore selects $k = 16$ for Repair-Aware Memory. These values are selected from offline memory states and are not tuned against the final scheduling benchmarks; the sweep is therefore a reproducibility and configuration-selection procedure rather than a scheduling-performance ablation.

C.5 Memory Retrieval and Graph Support Relations

Pattern Memory queries the current Construct state and supports feasible operation-machine pair nodes, whereas Repair-Aware Memory queries complete-schedule repair context and supports repair-related Destroy-to-Improve decisions. Both retrieve evidence from similar offline observations and are not updated from the current evaluation instance. For Construct, Pattern Memory is introduced through `memory→supports→pair` relations, where each `pair` represents one feasible operation-machine action. Destroy-to-Improve uses corresponding historical support for repair representations. Support relations use the same normalized rank, normalized score, confidence, and validity information defined above.

Memory and heuristic evidence are advisory: they cannot make an infeasible operation-machine pair feasible, remove a feasible pair, or force a policy action or repair region. Feasibility is determined by the scheduling environment and CP repair model, while these signals add information to the learned decision process.

C.6 Offline Memory Construction

Pattern Memory is built from states collected during Construct rollouts. For each sampled state, the offline builder computes the eight-dimensional signature and records associated information including the selected heuristic rule, rollout return, makespan change, and rule quality. The signatures are standardized, clustered, and summarized into the statistics later retrieved by Construct.

Repair-Aware Memory is built separately from recorded repair attempts. For each observation, the builder reconstructs the schedule context, computes the twelve-dimensional repair signature and associated diagnostics, records the repair channel, and stores outcome information including makespan improvement, runtime, repair cost, and repair budget. These observations are standardized and clustered so that similar repair contexts can share statistical evidence.

Both memories are fixed after offline construction. During training or evaluation, retrieval reads stored centroids and statistics without online updates. This keeps historical evidence stable across an inference episode and prevents outcomes observed later in the same evaluation run from changing the memory used for subsequent decisions.

D CONSTRUCT POLICY

D.1 STATE AND FEASIBLE ACTIONS

At construction step t, the Construct policy receives the heterogeneous graph $\mathcal{G}_t$ described in Appendix B. An operation is ready if it is real and unscheduled and either it is the first operation of its job or its immediate job predecessor has already been scheduled.

Let $\mathcal{M}_o$ denote the machines eligible to process operation o. The feasible action set is $\mathcal{A}_t = (o, m) \mid o$ is ready, , $m \in \mathcal{M}_o$, where each feasible action $a = (o, m) \in \mathcal{A}_t$ is represented by one pair node. Invalid operation-machine combinations are absent from $\mathcal{A}_t$, so no additional invalid-action mask is required before computing policy probabilities.

In JSSP, each ready operation contributes one action because it has exactly one eligible machine; in FJSP, it may contribute several, one per eligible machine. Internally, an action is represented by the flattened operation identifier together with the machine identifier, and the flattened operation identifier maps back to its job and position within that job.

D.2 ACTOR NETWORK

The graph encoder produces learned representations for feasible pair nodes and their corresponding operations and machines. The actor combines this structural representation with the heuristic and Pattern-Memory evidence described in Appendix C; these sources influence the learned pair representation but are never added as fixed overrides to the final action logit.

For each feasible action $a \in \mathcal{A} * t$, the actor receives a 16-dimensional representation and applies a two-layer multilayer perceptron with hidden width 16 and a Gaussian Error Linear Unit (GELU) between the two linear transformations. The output is a scalar logit z_a, representing the actor's unnormalized preference for action a. The policy distribution is $\pi * \theta(a \mid \mathcal{G} * t) = \frac{\exp(z_a)}{\sum *a' \in \mathcal{A} * t \exp(z * a')}$, where $\pi_\theta(a \mid \mathcal{G}_t)$ is the probability assigned to action a, θ denotes the trainable actor parameters, $\mathcal{G}_t$ is the current graph, and z_a is the action logit. Since normalization is restricted to $\mathcal{A}_t$, every action receiving probability mass is feasible.

During evaluation, the actor deterministically selects the feasible action with the largest logit. No sampling temperature is used, and an exact tie is resolved by the first maximum in the action ordering. When CP lookahead is enabled, the actor decision and action scores are passed to the lookahead module, which may replace the first choice with another evaluated feasible action; if no usable decision is returned, the original actor choice is retained.

D.3 CRITIC AND GLOBAL TOPOLOGY CONTEXT

The critic estimates the expected future return of the current construction state. Let $h_{\mathcal{G}_t} \in \mathbb{R}^{16}$ denote the global graph representation obtained by mean pooling the embeddings of the real operation nodes. It is augmented with a 20-dimensional topology context c_t summarizing the current partial schedule and the policy distribution.

Fourteen entries of c_t describe the scheduling state: scheduled-operation ratio, ready-operation ratio, relative number of feasible actions, mean, spread, maximum, and concentration of remaining work, mean, spread, and concentration of machine availability, concentration of remaining machine load, average machine flexibility, spread of remaining job workload, and Pattern-Memory confidence. Workload and timing quantities with strongly varying magnitudes are log-scaled before entering this context.

The remaining six entries summarize policy behavior: normalized action entropy, the probability gap between the two most likely actions, the probability mass assigned to the five most likely actions, and the mean source attention assigned to structural, heuristic, and memory information. Entropy measures how dispersed the action distribution is, while the top-action gap and top-five concentration indicate how strongly the policy favors a small subset of actions.

The critic computes $V_\psi(\mathcal{G} * t) = f * \psi(h_{\mathcal{G} * t} \| c_t)$, where $V * \psi(\mathcal{G} * t)$ is the scalar state-value estimate, ψ denotes the critic parameters, $\|$ denotes concatenation, and $f * \psi$ is the critic multilayer

perceptron. The 16-dimensional graph representation and 20-dimensional topology context form a 36-dimensional input, processed through a hidden layer of width 16 with GELU activation and mapped to one scalar output.

The actor and critic share the graph encoder but use separate output heads. The dynamic policy-related part of the topology context is detached before being passed to the critic, preventing critic gradients from flowing back through action probabilities or source-attention statistics.

D.4 Earliest-Feasible Insertion

After selecting action $a_t = (o, m)$, operation o is inserted into the earliest feasible position on machine m. Let $p_{o,m}$ denote its processing time and $c_{\mathrm{pred}(o)}$ the completion time of its immediate job predecessor, with $c_{\mathrm{pred}(o)} = 0$ when o is the first operation of its job.

Starting at $c_{\mathrm{pred}(o)}$, the procedure scans the operations already assigned to machine m in chronological order. If an interval of length $p_{o,m}$ fits completely before the next scheduled operation, o is inserted into that gap; otherwise, the search moves to the completion time of the conflicting operation and continues. If no earlier gap is available, o is placed after the last conflicting interval.

The resulting start time satisfies $s_o \geq c_{\mathrm{pred}(o)}$, and $[s_o, s_o + p_{o,m})$ does not overlap any operation already assigned to machine m. Its completion time is $e_o = s_o + p_{o,m}$, so the transition preserves job precedence, machine eligibility, machine capacity, and non-preemption. The environment then updates operation status, machine assignment, start and completion times, job progress, realized machine sequence, machine availability, and the ready-operation frontier before rebuilding the graph for construction step $t + 1$.

D.5 Reward and PPO Objective

Construct uses a dense shaped reward so that learning receives feedback before the full schedule is completed. For a valid decision at step t, the reward is

$$\begin{aligned} r_t = &-(C_t - C_{t-1}) + w_1(\phi_{1,t} - \phi_{1,t-1}) + w_2(\phi_{2,t-1} - \phi_{2,t}) \\ &+ w_3(\phi_{3,t-1} - \phi_{3,t}) + w_4(\phi_{4,t-1} - \phi_{4,t}) \\ &+ \alpha_{\mathrm{idle}} P_{\mathrm{idle}} + \delta_{\mathrm{job}} B_{\mathrm{job}} + \beta_{\mathrm{heavy}} P_{\mathrm{heavy}} + \gamma_{\mathrm{early}} B_{\mathrm{early}}. \end{aligned}$$

Here, C_t is the partial makespan after decision t, defined as the largest completion time among operations scheduled so far, so the first term penalizes increases in current makespan. The quantity $\phi_{1,t}$ is the fraction of real operations already scheduled, while $\phi_{2,t}$ is the remaining minimum workload divided by the total minimum workload of the instance, using the minimum processing time over eligible machines for each operation.

The quantity $\phi_{3,t}$ is a normalized lower-bound proxy for the remaining completion time: it takes the larger of a job-based and a machine-based lower bound and divides it by the total minimum workload. The job-based bound combines each job's latest scheduled completion with its remaining minimum work, while the machine-based bound combines already assigned busy time with estimated remaining machine pressure. The quantity $\phi_{4,t}$ is the largest number of unscheduled operations remaining in any job divided by the largest number of operations in a job.

The idle term is $P_{\mathrm{idle}} = -\frac{I_t}{T_t M + 10^{-6}}$, where I_t is the total unused machine time up to the current horizon, T_t is the largest current job completion time, and M is the number of machines. It therefore penalizes excessive idle capacity. The job-completion bonus B_{job} is the number of jobs completed by the current transition.

The machine-load term is $P_{\mathrm{heavy}} = -\frac{L_m - \bar{L}}{L_{\max} + 10^{-8}}$, where L_m is the accumulated busy time of the selected machine, $\bar{L}$ is the mean busy time across machines, and $L_{\max}$ is the largest machine busy time. Selecting a machine more heavily loaded than average therefore produces a negative contribution, while selecting a lighter machine may produce a positive one.

The earliest-finish term is $B_{\mathrm{early}} = -(e_{o,m} - e_o^{\mathrm{best}})$, where $e_{o,m}$ is the feasible completion time produced by the selected machine and e_o^{best} is the earliest feasible completion time available to the same operation over all its eligible machines. The term is zero for an earliest-finishing choice and negative for a slower alternative.

The coefficients are $w_1 = 0.2$, $w_2 = 0.2$, $w_3 = 0.4$, $w_4 = 0.2$, $\alpha_{\text{idle}} = 0.1$, $\delta_{\text{job}} = 1.0$, $\beta_{\text{heavy}} = 0.1$, and $\gamma_{\text{early}} = 0.3$. They determine the relative contributions of progress, remaining-work reduction, lower-bound reduction, job-tail reduction, idle time, job completion, machine balance, and earliest-finish behavior, respectively. The resulting reward is used directly without global clipping, standardization, or division by instance size.

Construct is optimized with Proximal Policy Optimization (PPO). Let r_t be the immediate reward and $V(\mathcal{G}_t)$ the critic estimate for graph $\mathcal{G} * t$. The temporal-difference residual is $\delta_t = r_t + \gamma V(\mathcal{G} * t + 1) - V(\mathcal{G}_t)$, where $\gamma = 0.99$ is the discount factor controlling the importance of future rewards.

Advantages are estimated recursively using Generalized Advantage Estimation (GAE), $A_t = \delta_t + \gamma\lambda(1 - d_t)A_{t+1}$, where A_t is the advantage of the selected action, $\lambda = 0.95$ controls temporal smoothing, and $d_t \in 0, 1$ indicates whether the transition terminates the episode. Advantages are normalized before optimization.

Let π_θ denote the current policy and π_{old} the policy used to collect the rollout. Their probability ratio for action a_t is $\rho_t(\theta) = \frac{\pi_\theta(a_t|\mathcal{G}*t)}{\pi*\text{old}(a_t|\mathcal{G}*t)}$. The actor loss is $L * \text{actor} = -\mathbb{E}_t\left[\min\left(\rho_t(\theta)A_t, \text{clip!}\left(\rho_t(\theta), 1-\epsilon, 1+\epsilon\right)A_t\right)\right]$, where $\mathbb{E}_t$ denotes the mean over the optimization transitions and $\epsilon = 0.2$ limits the permitted change in the policy probability ratio.

The complete loss is $L = L_{\text{actor}} + c_V L_{\text{value}} - c_H H$, where L_{value} is the critic loss, H is the entropy of the action distribution, $c_V = 0.5$ is the value-loss coefficient, and c_H is the entropy coefficient. The entropy coefficient starts at 0.005 and is reduced over the three PPO optimization epochs, gradually decreasing the explicit incentive for exploration.

Optimization uses Adam (Kingma & Ba, 2015) with an initial learning rate of 3×10^{-4}. A cosine schedule gradually reduces the learning rate during training, and the global gradient norm is clipped at 0.5 to reduce unstable updates.

E Constraint-Programming Lookahead

E.1 Candidate Actions and Restricted Subproblem

Constraint-programming lookahead is used only during Construct inference as an external reranking step after the actor scores the feasible actions. Let $\mathcal{A}_t$ denote the feasible operation-machine actions at construction step t. KARMS forms $\mathcal{A}_t^{(3)} \subseteq \mathcal{A}_t$ from the three actions with the largest actor logits, or all available actions when fewer than three exist. If only one feasible candidate remains, the actor decision is used directly and no lookahead solve is required.

For each candidate $a = (o, m) \in \mathcal{A}_t^{(3)}$, KARMS clones the current environment, commits operation o to machine m, and inserts it using the same earliest-feasible rule as ordinary Construct. The restricted CP model then reasons only about operations following this committed decision and cannot change the candidate's selected machine.

The future CP window contains at most eight unscheduled operations and is not simply the next eight Construct decisions or the next eight operations in a global order. Starting from the frontier created after the candidate action, the window builder expands through job successors, initially considers a larger operation pool, ranks it using readiness and knowledge-graph evidence, and retains at most eight precedence-compatible operations. An operation is included only when its predecessor is already scheduled or also present in the selected window.

The restricted CP model preserves the constructed schedule by representing scheduled operations as fixed machine intervals. Each selected future operation receives start and completion variables and a corresponding machine interval. In JSSP, machine assignment is fixed; in FJSP, future window operations may choose among eligible machines using optional intervals and an exactly-one assignment constraint. Machine capacity is enforced through `NoOverlap`, while job precedence is maintained within the window and through completion-time lower bounds when a predecessor lies outside it.

Let $J_{\text{window}}(a)$ denote the minimum makespan of the modeled CP window after candidate action a has been committed. This is the largest completion time among future operations represented in

the restricted model, not the makespan of the complete instance. Operations outside the selected window are not CP decision variables and are instead accounted for through the tail estimate below.

E.2 LOOKAHEAD SCORE

To avoid favoring candidates that perform well only inside the short window, KARMS augments the restricted CP objective with a lower-bound estimate of unmodeled work and a small Construct-actor prior. For candidate a, the score is $J_{\mathrm{LH}}(a) = J_{\mathrm{window}}(a) + \lambda_{\mathrm{tail}} LB_{\mathrm{tail}}(a) - \lambda_{\mathrm{actor}} z_a$, where $J_{\mathrm{LH}}(a)$ is the final lookahead score, $J_{\mathrm{window}}(a)$ the restricted CP makespan, $LB_{\mathrm{tail}}(a)$ a lower-bound estimate for operations outside the window, z_a the actor logit, $\lambda_{\mathrm{tail}} = 1.0$, and $\lambda_{\mathrm{actor}} = 0.05$. Because the actor term is subtracted, an actor-preferred candidate receives a small advantage when CP-based scores are otherwise similar.

The tail lower bound is the maximum of two quantities: the largest minimum processing workload still required by any job outside the modeled window and the largest expected remaining machine workload outside the window. A candidate is therefore penalized when either a job or a machine retains substantial unmodeled work.

A candidate is usable when CP returns `OPTIMAL` or `FEASIBLE`. `OPTIMAL` proves optimality for the restricted window, whereas `FEASIBLE` provides a valid incumbent without such a proof; both are retained for reranking. Results with `UNKNOWN`, `INFEASIBLE`, or `MODEL_INVALID` are discarded.

The selected action is therefore

$$\mathcal{A}_t^{\mathrm{use}} = \{a \in \mathcal{A}_t^{(3)} : \mathrm{status}(a) \in \{\mathrm{OPTIMAL}, \mathrm{FEASIBLE}\}\}, \qquad a_t^* = \arg\min_{a \in \mathcal{A}_t^{\mathrm{use}}} J_{\mathrm{LH}}(a).$$

where a_t^* is the action returned by the lookahead module and $\mathrm{status}(a)$ is the CP status obtained for candidate a. If two usable candidates have exactly the same lookahead score, the first candidate encountered is retained; candidates are evaluated in descending actor-logit order, so this favors the actor's higher-ranked candidate.

E.3 FALLBACK, CACHING, AND CONFIGURATION

Each candidate CP model is solved with a time limit of 100 ms and one CP search worker. Since at most three candidates are evaluated, the nominal solver budget is approximately $3 \times 100 = 300$ ms per Construct decision, excluding model-building and Python overhead. Lookahead is skipped when fewer than two feasible actor candidates are available or fewer than four unscheduled operations remain.

Only candidates returning `OPTIMAL` or `FEASIBLE` are usable. If none yields a usable CP result, or if lookahead cannot be applied, KARMS retains the original actor top-1 action. Thus, lookahead changes the actor decision only when it obtains a valid alternative and cannot prevent Construct from continuing when the restricted CP solve fails.

The active implementation does not cache CP-lookahead results. Each eligible candidate is solved again for the current construction state, and no score, solver status, schedule fragment, or runtime information is reused across subsequent decisions. The decision therefore depends only on the current actor scores and restricted CP solves.

The final configuration uses three actor candidates, a maximum CP window of eight future operations, a 100 ms CP time limit per candidate, one search worker, tail-bound weight $\lambda_{\mathrm{tail}} = 1.0$, and actor-prior weight $\lambda_{\mathrm{actor}} = 0.05$. CP lookahead is used only at inference and does not modify the trained Construct policy. It is separate from Destroy-to-Improve CP repair: lookahead evaluates short future consequences of a construction decision, whereas repair reconstructs a selected region of an already complete schedule.

F Complete-Schedule Analysis and Candidate Generation

F.1 Realized Machine Order and Critical-Path Analysis

Destroy-to-Improve starts from a complete incumbent schedule S_t, where t is the current improvement round. KARMS constructs a directed acyclic graph over operations using the original job-precedence edges and the realized machine order in S_t. For each job j, consecutive operations are connected by $o_{j,k} \to o_{j,k+1}$; for each machine, an edge $u \to v$ connects consecutive operations in its realized sequence. Only consecutive machine operations are linked.

Let p_o denote the processing duration of operation o, $\mathrm{Pred}(o)$ its predecessors, and $\mathrm{Succ}(o)$ its successors in the realized graph. The induced earliest start and completion times are $ES(o) = \max_{u \in \mathrm{Pred}(o)} EC(u), \qquad EC(o) = ES(o) + p_o$, where the maximum is zero when o has no predecessor. The realized-graph makespan is $C_{\mathrm{graph}} = \max_o EC(o)$.

A backward pass computes the remaining tail length $T(o) = \max_{s \in \mathrm{Succ}(o)} (p_s + T(s))$, with $T(o) = 0$ when o has no successor. The latest start compatible with the same makespan is $LS(o) = C_{\mathrm{graph}} - p_o - T(o)$, and the operation slack is $\mathrm{slack}(o) = LS(o) - ES(o)$, where $LS(o)$ is the latest feasible start without increasing C_{graph}.

An operation is critical when $|\mathrm{slack}(o)| \le 10^{-6}$. An edge $u \to v$ is critical when both operations are critical and the edge is temporally tight: $|ES(v) - (ES(u) + p_u)| \le 10^{-6}$. KARMS retains all critical operations and tight critical edges and also reconstructs one deterministic canonical critical path. The complete critical set supports structural analysis, while the canonical path provides a consistent reference for candidate-level coverage features.

F.2 Critical Blocks and Bottleneck Regions

A critical block is a maximal sequence of consecutive critical operations on the same machine whose adjacent pairs are connected by tight critical machine-order edges. Only blocks with at least two operations are retained, identifying machine-local critical structure where changing relative timing or assignment may affect makespan.

Critical-block candidates are complemented by bottleneck-machine regions. Operations with high machine-load diagnostics are used as seeds and expanded through nearby operations in the realized machine sequence. The standard bottleneck neighborhood uses two machine neighbors around the seed, while the segment variant uses radius three. These machine-centered candidates can expose congested regions not captured by a single canonical critical path.

F.3 Low-Slack, Late-Job, and Job-Tail Regions

Low-slack regions extend search beyond exactly critical operations. Operations are ranked by increasing slack; in the exact schedule analysis, the lowest-slack quarter is retained and connected components are formed using realized job and machine relations, producing coherent near-critical regions rather than isolated operations.

For operation o in job $j(o)$, the late-job score is $\ell(o) = \frac{C_{j(o)}}{C_{\max}}$, where $C_{j(o)}$ is the completion time of its job and $C_{\max}$ the current makespan. Larger values indicate jobs finishing closer to the end of the schedule; "late" is relative to the current makespan and does not assume due dates.

If $o_{j,k}$ is a seed and job j contains K_j operations, its complete job suffix is $\mathcal{T}(o_{j,k}) = o_{j,k}, o_{j,k+1}, \ldots, o_{j,K_j-1}$, where $\mathcal{T}(o_{j,k})$ is the job tail beginning at $o_{j,k}$. Candidate generation may initially use a shorter successor window; for example, the late-job-tail family considers up to four successive job operations. This local proposal remains distinct from the closure later applied to determine the region actually released for repair.

F.4 Candidate Families and Region Semantics

The candidate bank combines complementary structural views of the incumbent. Active generic families include critical-path blocks, bottleneck-machine segments, late-job tails, critical-path seeds,

bottleneck-machine seeds, low-slack regions, long-operation seeds, late-job seeds, and random job-tail seeds. When exact schedule analysis is applicable, additional candidates come from exact critical blocks, windows around the canonical critical path, and connected low-slack regions. These families provide structurally meaningful alternatives for the learned Candidate Head rather than prescribing repair actions.

For candidate i, let R_i^{prop} denote the proposed operation set. Before repair, KARMS applies job-tail closure: $R_i^{\text{eff}} = \bigcup_{o_{j,k} \in R_i^{\text{prop}}} \mathcal{T}(o_{j,k})$, where $\mathcal{T}(o_{j,k})$ is the complete job suffix beginning at proposed operation $o_{j,k}$. Thus, every proposed operation and its downstream job operations required by the repair contract belong to the effective region.

KARMS records both R_i^{prop} and R_i^{eff} because a compact proposal can expand after closure. The effective region may also exceed the nominal proposal budget when closure introduces additional job successors.

F.5 Deduplication and Provenance

Different structural families may produce the same effective repair region. Before neural scoring, KARMS therefore performs semantic deduplication using the effective operation set together with closure and backend semantics. Proposals releasing the same effective set under the same repair contract are treated as one candidate even if generated by different structural rules.

Merged candidates retain combined provenance, including source families, seed operations, and original proposed regions. Candidate-family identity is used for provenance, family balancing, and analysis but is not encoded as a categorical neural feature; the Candidate Head receives structural candidate properties rather than an explicit family label. Consequently, one effective region may accumulate evidence from several interpretations, such as both a critical block and a bottleneck-machine segment, while appearing only once in the candidate set.

F.6 Candidate Features

Each candidate is summarized by $x_i^{\text{cand}} \in \mathbb{R}^{18}$, describing both R_i^{prop} and R_i^{eff} so the policy can distinguish structural intent from the region actually released.

Five descriptors characterize size and closure behavior: relative proposed and effective region sizes, expansion introduced by job-tail closure, whether the effective region exceeds its nominal budget, and a retention descriptor relating proposed and effective representations. The closure-expansion ratio is $\rho_i^{\text{cl}} = \frac{|R_i^{\text{eff}}|}{\max(1, |R_i^{\text{prop}}|)}$, where $|\cdot|$ denotes set cardinality and ρ_i^{cl} measures how strongly the repair contract enlarges candidate i.

Six descriptors summarize critical structure. For both proposed and effective regions, KARMS records the fraction of contained operations that are critical, the fraction of the canonical critical path covered, and the fraction of critical-block operations covered. These distinguish a small candidate concentrated on critical operations from a larger candidate covering a broader schedule region.

Four descriptors represent exact slack information: separate availability indicators for the proposed and effective sets and the corresponding mean slack when available. Explicit availability indicators prevent missing slack measurements from being interpreted as genuine zero slack.

The remaining descriptors capture repair compatibility and structural breadth: whether the candidate is compatible with the repair backend and the fractions of machines and jobs touched by the effective region. Together, these features distinguish narrowly focused neighborhoods from candidates spanning many machines or jobs.

The feature vector uses single-precision floating point and is passed to the downstream candidate-scoring model without additional candidate-level standardization. Candidate-family names and provenance remain outside this vector and therefore cannot provide a direct categorical shortcut to the learned ranking function.

G Destroy Policy: Candidate and Operation-Set Selection

G.1 Candidate Representation

After the candidate bank in Appendix F is generated, the Destroy Policy first selects which candidate region to reconsider. Let $\mathcal{C}_t = C_1, \ldots, C_K$ denote the candidate set for incumbent schedule S_t, where K is the number of available candidates at improvement round t. Candidate nodes are excluded from shared GATv2 message passing: the schedule is encoded first, and each candidate is represented afterward using encoded operation and machine embeddings together with its structural features.

Let $D = 16$ be the encoded operation and machine dimension. Each candidate C_i has an 18-dimensional raw feature vector $x_i^{\text{raw}} \in \mathbb{R}^{18}$, defined in Appendix F.6, projected to $q_i \in \mathbb{R}^D$ by a two-layer $18 \to 16 \to 16$ network with GELU between the linear layers.

The candidate representation combines seven D-dimensional sources. Let $h_i^{\text{prop},\mu}$ and $h_i^{\text{prop,max}}$ be the mean- and max-pooled operation embeddings of the proposed region, $h_i^{\text{eff},\mu}$ and $h_i^{\text{eff,max}}$ the corresponding embeddings of the candidate-level effective region, h_i^{mach} the mean-pooled embedding of touched machines, and h_t^{glob} the mean-pooled operation embedding of the complete incumbent schedule. Empty sets are represented by zero vectors. These components are concatenated as $u_i = q_i \| h_i^{\text{prop},\mu} \| h_i^{\text{prop,max}} \| h_i^{\text{eff},\mu} \| h_i^{\text{eff,max}} \| h_i^{\text{mach}} \| h_t^{\text{glob}}$, where $\|$ denotes concatenation. Since all seven components have dimension $D = 16$, $u_i \in \mathbb{R}^{7D} = \mathbb{R}^{112}$. A candidate-context network maps this 112-dimensional vector to a 16-dimensional learned representation h_i^C. Candidate-family names remain provenance metadata and are not categorical neural inputs.

G.2 Candidate Head and Candidate Policy

The Candidate Head maps u_i through two linear layers of dimensions $112 \to 16 \to 16$ with GELU activations, followed by a final linear layer producing scalar logit z_i, the learned preference for candidate C_i under the current incumbent. The candidate policy is $\pi_\theta^C(C_i \mid S_t) = \frac{\exp(z_i)}{\sum_{C_j \in \mathcal{C}*t} \exp(z_j)}$, where $\pi * \theta^C(C_i \mid S_t)$ is the probability of candidate C_i, θ the trainable Destroy Policy parameters, S_t the current complete schedule, and z_i the Candidate Head logit. Candidate scoring therefore depends on structural descriptors, proposed and effective regions, affected machines, and global schedule context rather than heuristic-family identity. This implements the *where* decision; after candidate selection, a second learned module determines which operations in its proposed region are released.

G.3 Failure-Aware Candidate Filtering

KARMS avoids repeatedly attempting the same unsuccessful neighborhood while the incumbent remains unchanged. Each candidate has a canonical identifier derived from its effective repair semantics, and candidates that already failed for the current incumbent can be placed in a temporary ban set and removed before Candidate Head selection. The ban set is local to the incumbent: once an accepted repair changes the schedule, it is cleared and candidates are regenerated, allowing previously unproductive regions to be reconsidered under a changed schedule.

This differs from Repair-Aware Memory: the memory provides statistical evidence from previous repair experience, whereas failure-aware filtering is a temporary hard restriction preventing repeated evaluation of the same candidate under an unchanged incumbent.

G.4 Proposed Operation Set and Operation Tokens

Let C_t be the candidate selected by the Candidate Head and R_t^{prop} its proposed operation set. Only operations in R_t^{prop} are presented to the operation-selection module. Operations introduced only by candidate-level closure contribute context to the Candidate Head but are not directly selectable at this stage.

For each operation $o \in R_t^{\text{prop}}$, KARMS constructs a 52-dimensional token $x_o^{\text{tok}} = h_o \| h_{C_t} \| h_t^{\text{glob}} \| d_o^{\text{set}}$, where $h_o \in \mathbb{R}^{16}$ is the encoded operation representation, $h_{C_t} \in \mathbb{R}^{16}$ the learned

selected-candidate representation, $h_t^{\text{glob}} \in \mathbb{R}^{16}$ the global schedule embedding, and $d_o^{\text{set}} \in \mathbb{R}^4$ four scalar structural descriptors indicating effective-region membership, seed-hint status, proposal-related information, and a constant bias feature. Thus, $16 + 16 + 16 + 4 = 52$.

Each token is projected from 52 to 16 dimensions by a two-layer network with GELU activations. Because candidates contain different numbers of operations, shorter token sequences are padded within a batch and a Boolean padding mask prevents padding tokens from participating in attention. No positional encoding is used, so the module treats the selected operations primarily as a set rather than a fixed sequence.

G.5 Operation-Selection Transformer

Projected operation tokens are processed by a two-layer Transformer with model dimension $d_{\text{model}} = 16$. Each layer uses four self-attention heads, so each head operates on four dimensions. The feed-forward sublayer has width 32 with GELU activation; the Transformer uses pre-layer normalization, residual connections, and zero dropout. These four heads are distinct from the GATv2 attention of the schedule encoder: self-attention here operates only among operations of the selected candidate, allowing each operation representation to depend on the other operations that may be released with it.

The Transformer returns $h_o^{\text{set}} \in \mathbb{R}^{16}$ for each $o \in R_t^{\text{prop}}$. A linear scoring layer produces one scalar operation logit θ_o. These logits are not independent Bernoulli probabilities; they are scores for the subsequent cardinality-constrained selector.

G.6 Structured Top-B Selection and Repair Budget

The repair budget controls how many operations from the selected candidate are initially released. Let $n = |R_t^{\text{prop}}|$, let ρ be the target fraction of the proposed set to release, and let $B_{\max}$ be the upper bound on selected operations. The budget is $B_t = \min\left(n, B_{\max}, \max\left(1, \lceil \rho n \rceil\right)\right)$, so B_t is at least one for a non-empty candidate and never exceeds the number of proposed operations.

Given Transformer scores θ_o, the selector solves $\max_{x \in 0,1^n} \sum_{o=1}^{n} \theta_o x_o \quad \text{s.t.} \quad \sum_{o=1}^{n} x_o = B_t$, where $x_o = 1$ indicates that operation o is selected for release and $x_o = 0$ otherwise. Operationally, this selects exactly the B_t highest-scoring operations, using operation identifiers to break equal-score ties deterministically. The proposed destroy set is $R_t = o \in R_t^{\text{prop}} \mid x_o = 1$. The Transformer learns contextual operation scores while the cardinality constraint is enforced deterministically. There is no separate stochastic operation-set distribution in this structured path: learned candidate selection is followed by deterministic budget-constrained selection from the operation logits. Destroy-to-Improve remains sequential: one candidate is selected, one operation subset is formed, and one corresponding CP repair is attempted before the next improvement decision.

The effect of learned operation ranking when the cardinality constraint is active is examined separately in Appendix N.5.

G.7 Job-Tail Closure and Effective Repair Region

The budget B_t applies before repair closure. After selecting proposed destroy set R_t, KARMS recomputes the job-tail closure required by the repair contract. For selected operation $o_{j,k}$ at position k of job j, all later operations of that job are also released. The final effective repair region is $R_t^{\text{eff}} = \bigcup_{o_{j,k} \in R_t} o_{j,k}, o_{j,k+1}, \ldots, o_{j,K_j - 1}$, where K_j is the number of operations in job j. Closure adds only job successors, not job predecessors or machine-order neighbors.

Because closure follows structured Top-B_t selection, the final number of released operations may exceed B_t. The CP backend receives R_t^{eff}, not only the raw subset R_t.

This yields three distinct region representations. Appendix F defines the candidate-level effective region obtained by closing the full candidate proposal. Structured Top-B selection then selects the smaller subset R_t from the proposed region, and closure is recomputed from this subset to obtain final repair region R_t^{eff}. The first effective region provides candidate-ranking context; the second is the region actually released to the repair backend.

H CONSTRAINT-PROGRAMMING REPAIR

H.1 FIXED AND RELEASED DECISIONS

The CP repair stage receives the current complete incumbent schedule S_t and the effective repair region R_t^{eff} produced by the Destroy Policy. Rather than reconstructing the full schedule from scratch, KARMS fixes all decisions outside this region and lets CP reconsider only the released operations.

Let $\mathcal{O}$ denote the set of all operations and $F_t = \mathcal{O} \setminus R_t^{\text{eff}}$ the operations outside the repair region. For every $o \in F_t$, the incumbent start time is hard-fixed. In JSSP, its machine is predetermined by the instance; in FJSP, the incumbent machine assignment is also hard-fixed. Completion times then follow from the fixed start time and the processing duration on the assigned machine.

Operations in R_t^{eff} remain temporally free. In JSSP, their start times and realized machine order may change while machine assignments remain fixed. In FJSP, released operations may also move to another eligible machine, with processing duration determined by the selected machine. Incumbent values of released operations may be supplied as CP search hints but are not hard constraints.

Job precedence remains active across the repair boundary: a fixed predecessor constrains a released successor and a released predecessor constrains a fixed successor. Fixed machine intervals also remain in the machine-capacity constraints, preventing released operations from overlapping preserved incumbent operations.

H.2 JSSP REPAIR MODEL

For JSSP operation $o_{j,k}$, where j is the job and k its position, the repair model uses integer start variable $s_{j,k}$, integer completion variable $e_{j,k}$, and mandatory interval variable $I_{j,k}$. If $p_{j,k}$ is its fixed processing time, then $e_{j,k} = s_{j,k} + p_{j,k}$.

Each JSSP operation has one predetermined machine, so no machine-selection variable is introduced. Operations outside R_t^{eff} have start times fixed to incumbent values, whereas released operations may move in time subject to precedence and machine-capacity constraints.

A safe scheduling horizon is obtained from the total processing workload, $T_{hor} = \sum_{j,k} p_{j,k}$, and the start and completion variables are bounded by $0 \le s_{j,k}, e_{j,k} \le T_{\text{hor}}$.

H.3 FJSP REPAIR MODEL AND MACHINE REASSIGNMENT

For an FJSP operation o, let $\mathcal{M} * o$ denote its eligible machines. For each $m \in \mathcal{M} * o$, the model introduces Boolean presence variable $x * o, m$ and optional interval $I * o, m$ of duration $p_{o,m}$. The alternatives satisfy $\sum_{m \in \mathcal{M} * o} x * o, m = 1$, so exactly one eligible machine is selected.

The alternatives share the operation start and completion variables. When machine m is active, $e_o = s_o + p_{o,m}$, where s_o and e_o are the operation start and completion times. For a released operation, the incumbent machine is only a search hint and any $m \in \mathcal{M}_o$ may be selected. Outside the repair region, incumbent machine presence and start time are hard-fixed. Machine reassignment therefore occurs only inside the released region.

H.4 PRECEDENCE AND NO-OVERLAP CONSTRAINTS

The repair model preserves the original job order. For consecutive operations of job j, $e_{j,k-1} \le s_{j,k}$, regardless of whether the two operations are fixed, released, or lie on opposite sides of the repair boundary.

For every machine m, let $\mathcal{I}_m$ denote the intervals that may occupy that machine. In JSSP, $\mathcal{I}_m$ contains the mandatory intervals assigned to m; in FJSP, it contains the optional intervals corresponding to machine alternative m. Machine capacity is enforced through $\text{NoOverlap}(\mathcal{I}_m)$.

Each operation is represented by one uninterrupted interval after its machine is determined, so repair remains non-preemptive. No additional no-wait, blocking, or sequence-dependent setup constraints are introduced.

H.5 Makespan Objective under Local Repair

Although only a subset of decisions is released, CP minimizes the makespan of the complete repaired schedule. Let $\mathcal{F}(S_t, R_t^{\text{eff}})$ denote the feasible complete schedules preserving the fixed incumbent decisions outside R_t^{eff}. The repair model is $\min_{S \in \mathcal{F}(S_t, R_t^{\text{eff}})} C_{\max}(S)$, where $C_{\max}(S)$ is the makespan of complete schedule S.

The model represents makespan using the final operation of every job, $C_{\max} = \max_j e_{j,K_j-1}$, where K_j is the number of operations in job j. Fixed operations outside the repair region may therefore still determine the objective value.

The repaired schedule is additionally constrained to be non-worsening relative to the incumbent, $C_{\max} \leq C_t$, where $C_t = C_{\max}(S_t)$ is the incumbent makespan. No secondary penalty for machine changes, repair size, or deviation from the incumbent is added to the CP objective.

H.6 Solver Status, Time Limit, and Fallback

KARMS solves each local repair problem with OR-Tools CP using a 15,s time limit, one search worker, and a fixed solver seed. Search terminates earlier if optimality is proved. An `OPTIMAL` result proves optimality for the restricted repair model, while a `FEASIBLE` result provides a valid schedule found within the time limit and is also usable.

Let S'_t denote a feasible complete schedule returned by CP. If the time limit expires without a usable feasible schedule, the solver reports an infeasible model, or the repair backend fails, the repair attempt leaves the incumbent unchanged.

H.7 Feasibility Validation and Acceptance

The extracted CP solution specifies the machine, start time, and completion time of every operation. In JSSP, machine assignments follow the problem definition; in FJSP, the selected machine of a released operation is recovered from the active machine-alternative variable.

Let $C' * t = C * \max(S'_t)$ be the repaired makespan. A repair with $C'_t < C_t$ is accepted as an improvement. The sequential repair path may also retain a different feasible schedule with the same makespan when the schedule itself changes, allowing later improvement rounds to continue from an alternative realization of the same objective value. A true no-op, where both makespan and schedule remain unchanged, is rejected, as is a worsening repair.

Thus, feasibility and acceptance have distinct roles: CP determines whether a valid repaired schedule exists, while the improvement loop decides whether that schedule becomes the next incumbent.

H.8 Iterative State Update

If a repair is accepted, the next incumbent is $S_{t+1} = S'_t$. The next Destroy-to-Improve iteration then rebuilds its schedule-dependent state from this updated schedule, including a new candidate bank and corresponding candidate representations.

If the repair is rejected or no usable CP result is obtained, the incumbent remains unchanged, $S_{t+1} = S_t$. Failure-aware selection can then prevent the same candidate from being immediately reconsidered under the unchanged incumbent.

When an accepted repair changes the incumbent, the state-local failure history is cleared because candidate semantics may change under the new schedule. Destroy-to-Improve therefore proceeds sequentially: one candidate is selected, one repair is attempted, and the resulting incumbent is resolved before the next repair decision.

I Destroy-to-Improve Learning Objective

I.1 Improvement Reward and Candidate PPO

Destroy-to-Improve learns from repair outcomes on complete incumbent schedules. Let S_t denote the incumbent at improvement round t, let $S' * t$ be the schedule returned by repair, and define $C_t = C * \max(S_t)$ and $C' * t = C * \max(S'_t)$. The primary improvement signal is $\Delta_t = C_t - C'_t$. A positive value indicates strict improvement and zero indicates no immediate makespan reduction. As described in Appendix H.7, a feasible equal-makespan repair may still become the next incumbent when it changes schedule structure, whereas a true no-op or a repair without a usable schedule leaves the incumbent unchanged.

The stochastic reinforcement-learning decision is the candidate selected by the Candidate Head. Let $\mathcal{C} * t^{\text{avail}}$ denote the candidates remaining after state-dependent filtering. For $C_i \in \mathcal{C} * t^{\text{avail}}$ with logit z_i, the candidate policy is $\pi * \theta^C(C_i \mid S_t) = \frac{\exp(z_i)}{\sum * C_j \in \mathcal{C}_t^{\text{avail}} \exp(z_j)}$, where θ denotes the trainable candidate-policy parameters. During training, a candidate is sampled from this categorical distribution and its log probability is stored with the resulting repair transition.

Let π^C_{old} denote the rollout policy and A_t the advantage of the selected candidate. The PPO probability ratio is $r_t(\theta) = \frac{\pi^C_\theta(C_t|S_t)}{\pi^C_{\text{old}}(C_t|S_t)}$, and the clipped candidate-policy objective is $L_{\text{policy}} = -\mathbb{E}_t \left[\min\left(r_t(\theta)A_t, , \text{clip!}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) A_t\right)\right]$, where ϵ is the PPO clipping parameter and $\mathbb{E}_t$ denotes the mean over collected training transitions.

A critic predicts the return of the current complete schedule and is optimized with value-regression loss L_{value}. Candidate-policy entropy H_C discourages premature concentration on a small number of candidate regions. The actor-critic objective is $L_{\text{PPO}} = L_{\text{policy}} + c_V L_{\text{value}} - c_H H_C$, where c_V and c_H control the value and entropy contributions. The policy π^C_{old} is the PPO rollout policy, not a separate persistent reference model.

I.2 Training of Structured Operation-Set Selection

Operation selection is trained progressively in three stages: supervised repair-outcome subset pretraining, hierarchical PPO over candidate and operation-set decisions, and CP-guided structured training. In the final Improve configuration, PPO updates candidate selection only, while operation-level parameters receive CP-guided Fenchel-Young supervision. Reported inference uses deterministic Top-B operation selection.

Supervised subset pretraining. For a candidate with proposed operation set P, an evaluated subset $A \subseteq P$ receives the normalized OperationSet score

$$s(A) = \frac{1}{|P|}\left[\sum_{o\in A} \log\sigma(z_o) + \sum_{o\in P\setminus A} \log\sigma(-z_o)\right],$$

where z_o is the contextual operation logit of operation o. Each evaluated subset A_j receives normalized CP repair utility

$$u_j = \frac{C_{\text{before}} - C_{\text{after},j}}{\max(1, C_{\text{before}})}.$$

The evaluated utilities define the soft listwise target

$$q_j = \text{softmax}\left(\frac{u_j - \max_k u_k}{T}\right), \qquad T = 0.01,$$

while the predicted subset probabilities are

$$p_j = \text{softmax}\big(s(A_j)\big).$$

The listwise objective is

$$L_{\text{list}} = -\sum_j q_j \log p_j.$$

We additionally compare each informative evaluated subset against the complete proposed set P, which serves as a fixed reference for the pairwise term. Let

$$\Delta_j = u_j - u_{\text{all}},$$

where u_{all} is the normalized CP utility obtained when all operations in P are passed to closure and repair. Subsets satisfying

$$|\Delta_j| \le \epsilon_{\text{tie}}$$

are treated as uninformative for the pairwise comparison. Let

$$\mathcal{I} = \{j : |\Delta_j| > \epsilon_{\text{tie}}\},$$

and, for each $j \in \mathcal{I}$, define

$$y_j = \begin{cases} +1, & \Delta_j > 0, \\ -1, & \Delta_j < 0. \end{cases}$$

Let $s_j = s(A_j)$ and let $s_{\text{all}} = s(P)$ denote the predicted score of the complete proposed set. The pairwise ranking loss is

$$L_{\text{pair}} = \frac{1}{|\mathcal{I}|} \sum_{j \in \mathcal{I}} \text{softplus}\left(-y_j \left(s_j - s_{\text{all}}\right)\right).$$

Thus, when a subset produces higher CP repair utility than the complete proposed set, the loss encourages $s_j > s_{\text{all}}$; when the complete proposed set produces higher utility, the preference is reversed. The complete supervised OperationSet objective is

$$L_{4A} = L_{\text{list}} + 0.5 L_{\text{pair}}.$$

This stage provides the initial operation-level supervision from CP-evaluated subsets.

Hierarchical PPO. The pretrained operation-selection module is subsequently refined within the hierarchical improvement policy. During this stage, a candidate region is first sampled from the Candidate Head and a stochastic operation-set decision is then made within the selected candidate. When operation-level subset sampling is active, the stored joint log probability contains the candidate-selection term together with the corresponding subset-selection and operation-level terms. These terms contribute jointly to the PPO probability ratio, allowing both candidate and operation-level decisions to receive policy-gradient updates from repair outcomes. This stochastic hierarchy is used as an intermediate training mechanism and is not the operation-set decoder used by the reported final configuration.

CP-guided structured training. The final operation-level training stage replaces stochastic subset decoding with structured Top-B selections evaluated directly through CP repair. Given operation logits θ, the deterministic structured decoder produces the base core

$$a_0 = f_B(\theta)$$

and five perturbed alternatives

$$a_k = f_B(\theta + \sigma_b \xi_k).$$

Each core is expanded through job-tail closure and evaluated using the same local CP repair mechanism used at inference. Repair utilities are clipped to ± 0.10 and converted into a soft target over the evaluated cores. The Fenchel-Young objective updates the operation-selection module together with the final GNN block, encouraging operation scores whose structured Top-B selections lead to better CP repairs.

In the final Improve configuration, PPO updates candidate selection only; operation logits are not included in the PPO probability ratio. Operation-level learning is instead provided by the CP-guided Fenchel-Young objective. The final candidate-ranking adaptation subsequently keeps the operation-selection module fixed.

Reported evaluation. At evaluation, the Operation-Selection Transformer produces contextual operation scores θ_o for the selected candidate, and deterministic Top-B_t selection chooses the operations to release,

$$R_t = f_B(\theta).$$

The reported configuration uses $\rho = 0.85$ and $B_{\max} = 250$. Job-tail closure then forms the effective repair region, which is passed to CP with the reported 15 s local repair budget. No stochastic subset gate or Bernoulli operation sampling is used at reported inference; the final operation set is produced deterministically by Top-B_t selection.

I.3 Parameter Updates

Each Destroy-to-Improve transition links a candidate decision to the outcome of its repair. Candidate-selection and operation parameters are updated at different stages. The dedicated operation stage optimizes only the operation-token projection, Operation-Selection Transformer, and operation-logit head using repair-outcome supervision. During the intermediate hierarchical-PPO stage, candidate and operation-level decisions are jointly optimized through the stored joint policy probability; the final Improve stage subsequently removes operation-level terms from the PPO ratio. Later CP-guided Fenchel-Young updates provide direct operation-level supervision from CP-evaluated subsets, whereas CP-guided pairwise candidate-ranking updates affect candidate ranking rather than operation logits.

The constraint solver is non-differentiable and receives no gradient. Repair-Aware Memory is also not optimized through the reinforcement-learning loss; it remains an offline source of contextual evidence retrieved during policy evaluation. Learning therefore occurs through the neural decision components, while feasibility and schedule reconstruction remain the responsibility of CP.

This separation mirrors inference in Appendices G and H: the learned policy decides where to search, structured operation selection determines what to release, and CP determines how the released region is reconstructed feasibly.

J Training Procedure

KARMS is trained stage-wise rather than end-to-end. Construct is trained first to produce complete feasible schedules and is then frozen. Destroy-to-Improve is trained separately on complete schedules with its own parameters, so information passes between the two stages through schedules rather than shared weights or end-to-end gradients.

J.1 Construct Training

Construct is trained with PPO on synthetic JSSP and FJSP instances using the feasible operation-machine action space described in Appendix D. Training spans different problem sizes and machine-flexibility levels so the same policy learns both fixed-machine JSSP decisions and flexible FJSP assignments.

Training follows a size-based curriculum and continues from previously learned Construct parameters as harder instances are introduced. After selecting the final Construct model, its parameters are frozen and the model is used to generate complete schedules for Destroy-to-Improve training.

J.2 Destroy-to-Improve Training

Destroy-to-Improve is trained on complete feasible schedules rather than empty instances. Training states include schedules generated by the frozen Construct policy, schedules obtained from dispatching heuristics, and previously generated training trajectories, exposing the improvement policy to incumbents with different quality and structural characteristics.

Operation-level selection is trained separately from candidate ranking. It is initialized from CP-evaluated operation subsets using the supervised listwise and pairwise objectives of AppendixI.2, subsequently refined through hierarchical PPO, and finally updated through CP-guided structured/FY training using deterministic and perturbed Top-B subsets evaluated by the local repair backend.

A final adaptation stage further refines candidate ranking from additional CP repair outcomes. It continues from the already trained Improve model rather than random initialization, retains the operation-selection module learned in the preceding stages, and focuses on improving which regions are selected for CP search.

J.3 Pattern- and Repair-Memory Data Generation

Pattern Memory and Repair-Aware Memory are constructed offline from training-side observations. Pattern Memory uses Construct-related rollout states, whereas Repair-Aware Memory uses recorded repair contexts and outcomes; their representations and retrieval procedures are described in Appendix C.

Memory construction is separate from gradient-based optimization. Once loaded for training or evaluation, stored clusters and statistics act as fixed contextual evidence and are not updated through PPO or CP repair outcomes.

J.4 Training Curriculum

The reported curriculum uses synthetic JSSP and FJSP instances up to 30×20. Larger experimental instances are therefore out-of-training-scale tests of size generalization.

The Construct curriculum is organized into 12 phases per cycle: six JSSP phases and six FJSP phases. The corresponding problem sizes are 6×6, 10×10, 15×15, 20×15, 20×20, and 30×20 for each problem class. Thus, every curriculum cycle visits the same six size levels for JSSP and FJSP before proceeding to the next cycle. Destroy-to-Improve additionally varies the source and quality of the incumbent schedule.

J.5 Frozen and Trainable Components

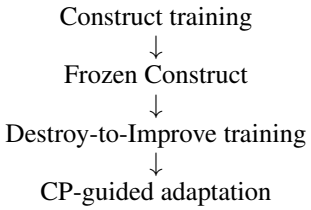
Construct and Destroy-to-Improve use separate graph encoders, policy components, and checkpoints. Once Construct training is complete, its parameters remain fixed throughout Destroy-to-Improve training, and no improvement loss propagates into the Construct model.

Within Destroy-to-Improve, different neural components are updated at different stages. Candidate-aware training updates the learned schedule representation and candidate-selection components. The operation-selection module used at inference is trained in preceding Improve stages through subset-level supervision, hierarchical PPO, and CP-guided structured/FY updates. Final CP-guided adaptation updates only the Candidate Head and the final GNN block; operation parameters are inherited unchanged.

Heuristic rules, candidate generation, Pattern Memory, Repair-Aware Memory, structured Top-B selection, and CP repair are non-gradient components. CP supplies repair outcomes to learning but is not part of the differentiable computation graph.

J.6 Stage-Wise Training, Seeds, and Model Selection

The overall training procedure is stage-wise rather than end-to-end. Construct is trained first and then frozen. Destroy-to-Improve is trained separately on complete schedules, and the final CP-guided adaptation continues from the previously selected Improve parameters rather than restarting from random initialization. Complete schedules therefore form the interface between Construct and Destroy-to-Improve, while CP supplies repair outcomes without receiving or propagating gradients.



For reproducibility, the reported Construct checkpoint is obtained from a training run with random seed 1, the Destroy-to-Improve training stages use seed 20260824, and the final CP-guided adaptation uses seed 20260905. Checkpoints are selected exclusively from held-out training-side validation data rather than from the public benchmark instances. Construct is selected according to normalized validation performance, Improve according to held-out repair performance, and the

final adaptation is retained only when it improves its held-out selection criterion without degrading previously learned scheduling behavior. The resulting Construct and Destroy-to-Improve models are then frozen for the reported benchmark evaluation. Sensitivity to Construct training randomness is examined separately through three independent Construct training runs in Appendix N.3.

K Implementation and Reproducibility Details

KARMS is implemented in Python using PyTorch and PyTorch Geometric for learned components and OR-Tools CP for constraint-based lookahead and repair. This section records the final execution settings required to reproduce evaluation; architectural and optimization details are given in the preceding appendices.

K.1 Key Model and Search Settings

Construct and Destroy-to-Improve use the heterogeneous GATv2 encoder of Appendix B.5. Raw schedule-node features have dimension 8, are projected to width 32, processed by two residual heterogeneous GATv2 layers of width 32, and mapped to 16-dimensional node embeddings. Each typed relation uses one GATv2 attention head; relation-specific messages are summed, ReLU is used between message-passing stages, and the final encoder uses neither dropout nor normalization.

Construct uses hidden width 16 for both actor and critic heads. The critic concatenates the 16-dimensional pooled graph representation with the 20-dimensional topology context of Appendix D.3, giving a 36-dimensional input.

Each repair candidate uses the 18-dimensional structural representation of Appendix F.6. For operation-set selection, each proposed operation is represented by a 52-dimensional token consisting of its 16-dimensional operation embedding, 16-dimensional candidate representation, 16-dimensional global schedule representation, and four scalar structural descriptors. Tokens are projected to dimension 16 and processed by a two-layer Operation-Selection Transformer with four self-attention heads, feed-forward width 32, pre-layer normalization, zero dropout, and no positional encoding.

Structured operation selection uses $B_t = \min\left(n, B_{\max}, \max\left(1, \lceil \rho n \rceil\right)\right)$, where n is the number of proposed operations in the selected candidate. The final repair configuration uses $\rho = 0.85$ and $B_{\max} = 250$. Because job-tail closure follows Top-B_t selection, the effective repair region may contain more than B_t operations.

Destroy-to-Improve performs at most 20 sequential improvement rounds. Each round processes one candidate, one operation subset, and one corresponding CP repair before the next decision.

K.2 Constraint-Programming Configuration

KARMS uses CP in two inference roles: short-horizon reranking during Construct and local reconstruction during Destroy-to-Improve.

Construct lookahead considers the three feasible actions with the largest actor logits. After temporarily committing each candidate, the restricted CP subproblem contains at most eight precedence-compatible future operations. Each candidate solve uses a 100,ms time limit and one CP search worker. Lookahead is skipped when fewer than two candidate actions are available or fewer than four operations remain unscheduled.

The lookahead score combines the restricted CP objective with a lower bound on the unmodeled remainder and a small actor prior, using $\lambda_{\text{tail}} = 1.0$ and $\lambda_{\text{actor}} = 0.05$. Both `OPTIMAL` and `FEASIBLE` results are usable; if none is obtained, Construct retains the actor's original top-ranked action.

Destroy-to-Improve uses OR-Tools CP to optimize the selected effective repair region while preserving unaffected incumbent decisions, as described in Appendix H. Each repair uses a 15,s time limit, one CP search worker, and a fixed solver seed. A feasible or optimal repaired schedule may be returned; if no usable solution is obtained, the incumbent remains unchanged.

The standalone CP baseline is separate from both mechanisms above: it solves the complete scheduling instance and is reported independently in the experimental protocol.

K.3 Hardware and Software Environment

Experiments were executed on a workstation with an Intel Core Ultra 9 285K CPU, 62,GiB of system memory, and an NVIDIA GeForce RTX 4060 GPU with 8,GB of memory. The software environment used Python 3.12.3, PyTorch 2.11.0, PyTorch Geometric 2.7.0, and OR-Tools 9.15. The PyTorch installation used CUDA 13.0.

Neural decisions are deterministic during evaluation. Construct selects the highest-scoring feasible action before optional CP reranking, while Destroy-to-Improve uses deterministic candidate selection and deterministic structured Top-B_t operation selection. CP uses one worker and a fixed random seed to improve run-to-run reproducibility.

K.4 Final Inference Configuration

For JSSP and FJSP, inference starts from an empty Construct state. At each step, KARMS rebuilds the heterogeneous graph, enumerates feasible operation-machine actions, evaluates them with Construct, and optionally reranks the highest-scoring actions with CP lookahead. The selected operation is committed by earliest-feasible insertion until a complete feasible schedule S_0 is obtained.

Destroy-to-Improve then operates on the complete incumbent. At each round, KARMS recomputes complete-schedule analysis and the candidate bank, removes candidates excluded by failure-aware filtering, and deterministically selects one candidate. The Operation-Selection Transformer jointly scores operations in its proposed region, structured Top-B_t selection forms the initial destroy set, and job-tail closure produces the effective region passed to CP.

Repair remains sequential: the current CP result is accepted or rejected before another candidate is considered. After an accepted schedule-changing repair, the schedule-dependent state, graph, and candidate bank are rebuilt from the new incumbent. After failure or rejection, the incumbent is retained and failure-aware state can prevent immediate repetition of the same candidate.

FFSP evaluation uses the same trained neural models through the canonical FJSP-compatible adapter; no FFSP-specific neural training is performed. For external schedules produced by other methods, Construct is bypassed and the supplied feasible schedule is passed directly to the frozen Destroy-to-Improve pipeline.

L Experimental Protocol

L.1 Training and Validation Instances

The reported training protocol uses synthetically generated JSSP and FJSP instances with problem sizes up to 30×20. Training and model-selection instances are kept separate from the public benchmark suites used for the final evaluation. Construct and Destroy-to-Improve use training-side validation data, while later model-selection stages additionally use held-out selection and guard instances to prevent degradation before an update is retained. The public JSSP and FJSP families described below are reserved for evaluation and are not used for policy training or benchmark-specific fine-tuning. The resulting experiments therefore measure transfer from the synthetic training distribution to established scheduling benchmarks and larger problem sizes.

L.2 JSSP and FJSP Benchmark Suites

We evaluate KARMS on 242 JSSP benchmark instances from the FT, Taillard, DMU, ABZ, LA, ORB, SWV, and YN families (Fisher & Thompson, 1963; Taillard, 1993; Demirkol et al., 1998; Adams et al., 1988; Lawrence, 1984; Applegate & Cook, 1991; Storer et al., 1992; Yamada & Nakano, 1992), and on 258 FJSP benchmark instances from the Brandimarte, Hurink, Dauzère-Pérès-Paulli, Barnes-Chambers, Kacem, Fattahi, and Behnke families (Brandimarte, 1993; Hurink et al., 1994; Dauzère-Pérès & Paulli, 1997; Barnes & Chambers, 1996; Kacem et al., 2002)

(Fattahi et al., 2007; Behnke & Geiger, 2012), for a total of 500 established shop-scheduling instances. The FJSP collection includes the three Hurink flexibility variants (*edata*, *rdata*, and *vdata*).

Table 12: JSSP and FJSP benchmark suites used in the evaluation.

JSSP		**FJSP**	
Family	Instances	Family	Instances
ABZ	5	Barnes	21
DMU	80	Brandimarte	15
FT	3	Dauzère-Pérès	18
LA	40	Fattahi	20
ORB	10	Kacem	4
SWV	20	Behnke	60
Taillard	80	Hurink edata	40
YN	4	Hurink rdata	40
		Hurink vdata	40
Total	**242**	**Total**	**258**

The Hurink *edata*, *rdata*, and *vdata* subsets are reported separately because they represent different flexibility settings over the same underlying LA problem structure. For comparisons with published JSSP methods that do not cover the complete 242-instance collection, results are restricted to the common 162-instance subset so that all compared methods use the same instances.

L.3 FFSP Transfer Protocol

We additionally evaluate transfer to flexible flow-shop scheduling (FFSP) without FFSP-specific gradient training or fine-tuning. FFSP instances are mapped to the canonical FJSP-compatible representation described in Appendix A.3, while the learned Construct and Destroy-to-Improve parameters remain unchanged.

The evaluation contains 3,000 fixed test instances, divided equally among problems with 20, 50, and 100 jobs. Each instance contains three stages with four parallel machines per stage.

Table 13: FFSP transfer evaluation set.

Jobs	Instances	Stages	Machines/stage
20	1,000	3	4
50	1,000	3	4
100	1,000	3	4
Total	**3,000**		

Each KARMS instance produces one solution using the frozen models. All compared methods are evaluated on the same fixed FFSP test set. Performance is reported by mean makespan within each problem size because this transfer set does not provide a common BKS reference suitable for the gap measure used in the JSSP and FJSP experiments.

L.4 External-Schedule Repair Protocol

To evaluate Destroy-to-Improve independently of the KARMS Construct policy, we additionally apply it to complete schedules produced by ReSched and DANIEL. Construct is bypassed: each source schedule is converted to the common schedule representation, validated, and then supplied directly to the frozen Destroy-to-Improve model.

The experiment uses 135 common FJSP instances: all 15 Brandimarte instances and 40 instances from each of Hurink *edata*, *rdata*, and *vdata*. ReSched contributes three schedules per instance, while two DANIEL model variants each contribute one schedule per instance. This gives 675 paired before-after evaluations.

Table 14: External schedules used to evaluate the frozen Destroy-to-Improve model.

Source	Variants	Instances/variant	Schedules
ReSched	3	135	405
DANIEL	2	135	270
Total	5		**675**

Table 15: Provenance and inference protocol of the principal comparison methods. Published runtimes are used as computational context when hardware, sampling budgets, or timing boundaries differ from the local KARMS evaluation.

Method	Domain	Result source	Inference mode	Runtime use
CP	JSSP/FJSP	Local/reference evaluation	Full-instance bounded solve	Separate CP comparison
RL-GNN	JSSP	Published	Source-defined	Context only
ScheduleNet	JSSP	Published	Source-defined	Context only
L2S-500/5000	JSSP	Published	Iterative search	Context only
TBGAT-1000/5000	JSSP	Published	Iterative search	Context only
Residual Scheduling	JSSP/FJSP	Published	Source-defined	Context only
HGNN	FJSP	Published	Sampling-based	Context only
DANIEL	FJSP	Published/supplied	Sampling-based	Context only
NDS-S	JSSP/FJSP	Official released outputs, validation-selected model	Iterative deconstruction search, ~ 60 s	Context only
PRISMA-S	FJSP	Published per-instance results	Sampling-based construction	Context only
MIStar-400	FJSP	Published	Improvement search	Context only
ReSched	JSSP/FJSP/FFSP	Published/supplied	Sampling-based	Context only
MatNet	FFSP	Published	Sampling/augmentation	Context only
MatNet$\times$128	FFSP	Published	128-sample setting	Context only
KARMS	JSSP/FJSP/FFSP	Local	One deterministic run	Local timing

Before improvement, each imported schedule is checked for complete operation coverage, processing-time consistency, precedence feasibility, machine eligibility, machine non-overlap, and consistency of the reconstructed makespan. All 675 schedules used in the experiment pass these checks. No parameters, embeddings, logits, or training data are transferred from the external initializer to KARMS.

PRISMA uses CP-derived solutions for imitation learning and training-time exploration guidance; its reported inference remains a learned constructive policy. NDS-S instead performs iterative learned deconstruction followed by heuristic reconstruction. These roles differ from KARMS, where CP is invoked directly at inference time for bounded lookahead and localized schedule reconstruction.

For every input schedule, the repaired schedule is paired with its original initializer. ReSched therefore provides three repeated schedules for each underlying instance, whereas the two DANIEL models correspond to distinct model/training variants rather than independent random seeds. Worsening repairs are rejected by the acceptance rule; consequently, zero worsened final schedules should not be interpreted as an independent learned safety property.

L.5 Baselines, Reference Values, and Comparison Coverage

The experiments compare KARMS with a full-instance CP baseline and with published learning-based scheduling methods. KARMS produces one deterministic solution per instance and does not use test-time sampling or best-of-N selection. Published methods retain the inference protocol reported in their original sources, including sampling, augmentation, or iterative improvement budgets where applicable. We therefore compare published solution-quality values under their reported protocols rather than reinterpret them as deterministic one-shot inference.

Published methods do not provide uniform coverage over all benchmark families. In particular, several JSSP methods cover the standard non-DMU families but omit individual subsets, while the principal published FJSP baselines report results mainly on Brandimarte MK01-10 and the three Hurink variants. The main paper therefore restricts direct method-to-method discussion to benchmark settings with corresponding values. Appendix M retains the full evaluation matrices, including families for which one or more comparison methods do not report matching results. Missing entries are treated as unavailable evidence and never as wins for KARMS.

For NDS-S, we follow the model-selection protocol of the original method rather than selecting a checkpoint independently for each test instance. The official validation results select the 20×20 JSSP model among the four released JSSP models and the 20×10 FJSP model with up to 15 operations per job among the two released FJSP models. For each benchmark instance, $C_{\mathrm{NDS-S}}$ is the final makespan recorded by the corresponding selected model after the approximately 60 s search budget. No per-instance minimum across alternative NDS-S checkpoints is used. Gaps are then recomputed using the same C_{ref} values as the KARMS evaluation.

For PRISMA-S, the published per-instance makespans are likewise converted to gaps using the same instance reference values as the KARMS evaluation; the reported schedules and inference outputs themselves are unchanged.

The standalone CP baseline is distinct from both forms of CP embedded inside KARMS. Construct lookahead solves short restricted subproblems to rerank a small set of actor decisions, while Destroy-to-Improve uses CP only to reconstruct a selected local region of an incumbent schedule. The standalone CP baseline instead optimizes the complete scheduling instance.

Two CP result configurations are kept conceptually separate. The complete JSSP benchmark table uses the 3,600 s CP comparison values, whereas the paired statistical analysis uses a separate 1,800 s full-instance CP evaluation on the same JSSP instances as KARMS. Classical FJSP likewise uses the 1,800 s full-instance CP evaluation. Consequently, the 3,600 s JSSP comparison row is not used in the paired significance tests reported in Appendix L.7.

Table 16: Full-instance CP configurations used for different experimental purposes. These solves are separate from Construct lookahead and local Destroy-to-Improve repair.

Result set	Time limit	Use
JSSP benchmark CP	3,600 s	Complete comparison matrix
JSSP paired CP	1,800 s	Statistical analysis
FJSP paired CP	1,800 s	Comparison/statistics

For JSSP and FJSP, solution quality is measured against the reference makespan associated with each benchmark instance. We use the general term *reference value* rather than assuming that every stored value is a proven optimum. Classical FJSP and Hurink references follow the reference collections used by the final evaluation, including FJSPLib values where applicable.

Taillard JSSP requires particular care because the provenance of its stored reference values is not identical to that of all other classical JSSP families, and historical publications may rely on different reference collections. Comparative statements are therefore made from the gaps reported under the corresponding evaluation protocol rather than by assuming that every publication necessarily used an identical reference file.

L.6 Evaluation Metrics and Runtime

The primary scheduling objective is the makespan $C_{\max}$. For an instance with reference value C_{ref}, the relative gap is $\mathrm{Gap}(\%) = 100\,\frac{C_{\mathrm{method}} - C_{\mathrm{ref}}}{C_{\mathrm{ref}}}$, where C_{method} is the makespan returned by the evaluated method. Negative gaps are retained when a method improves on the stored reference rather than being clipped to zero.

For schedule-improvement experiments, we additionally report the paired reduction in reference gap, $\Delta\mathrm{Gap} = \mathrm{Gap}_{\mathrm{initial}} - \mathrm{Gap}_{\mathrm{final}}$, where a positive value indicates that the final schedule improves on its paired initializer.

Family-level JSSP and FJSP results are arithmetic means of the per-instance gaps in the corresponding family. Overall benchmark summaries use instance-level averaging unless a family-level macro average is explicitly identified. FFSP results are averaged independently over the 1,000 fixed instances at each problem size.

For KARMS, local runtime covers the executed scheduling pipeline, including neural evaluation, schedule construction, enabled CP lookahead, candidate generation, Destroy-to-Improve repair, and

Table 17: Evaluation metrics used throughout the experiments.

Metric	Definition
Makespan	$C_{\max}$
Reference gap	$100(C - C_{\text{ref}})/C_{\text{ref}}$
Gap reduction	$\text{Gap}_{\text{initial}} - \text{Gap}_{\text{final}}$
W/T/L	Improved / unchanged / worsened paired schedules
Runtime	Wall-clock evaluation time under the stated timing scope
Speedup	$T_{\text{reference}}/T_{\text{method}}$ when timing scopes are comparable

final validation. The reported value is therefore an end-to-end evaluation quantity rather than only neural forward time or only local-search time.

Published runtimes require a different interpretation. Existing methods differ in hardware, sampling multiplicity, search horizon, augmentation, and in which stages are included in the reported time. For example, an iterative improvement method may report only refinement time whereas another source may include construction and decoding. We therefore retain published runtimes as computational context but avoid direct speedup claims unless both execution scope and timing definition are sufficiently comparable.

Table 18: Interpretation of runtime measurements used in the experiments.

Method group	Runs/instance	Timing source	Interpretation
KARMS	1	Local evaluation	End-to-end local timing
Standalone CP	1	Local/full-instance solve	Direct under stated budget
Published JSSP/FJSP methods	Method-specific	Original source	Context only
Published FFSP methods	Method-specific	Original source	Context only

Component-level runtime audit. To identify where end-to-end inference time is spent, we additionally profile the complete KARMS pipeline on all 242 JSSP and 258 FJSP benchmark instances. Table 19 separates learned policy computation from bounded CP reasoning during construction and complete-schedule improvement.

Across all 500 instances, bounded CP reasoning accounts for 78.1% of mean E2E time: 65.7% is spent in Construct lookahead and 12.3% in local complete-schedule repair. In comparison, the learned Construct and Improve policy computations together account for 21.2%. The dominant cost is therefore the bounded constraint reasoning rather than the learned graph and selection modules.

The stage-level split leads to the same conclusion. Construction accounts for 84.4% of mean E2E time over the full evaluation, whereas the complete Destroy-to-Improve stage accounts for approximately 14.9%. On JSSP alone, 86.2% of E2E time is spent before Destroy-to-Improve. This also explains why reported runtimes from improvement-only methods should not be interpreted as directly comparable to KARMS's end-to-end measurements.

Table 20 reports the runtime decomposition on th 50 representative JSSP instances in the main paper, CP reasoning accounts for 81.3% of mean E2E time, compared with 18.0% for the learned Construct and Improve policies. Construct lookahead is the dominant cost (58.3%), indicating that runtime is driven primarily by bounded constraint reasoning rather than neural inference.

L.7 Statistical Analysis

Statistical inference is restricted to comparisons for which per-instance outcomes are available on exactly the same evaluation instances and under the same reference definition. Aggregate family means from published papers are therefore not converted into artificial paired samples. This prevents differences in benchmark coverage or aggregation from being mistaken for instance-level statistical evidence.

For a baseline b and KARMS, the paired gap difference on instance i is $\Delta\text{Gap}_i = \text{Gap}_{b,i} - \text{Gap}_{\text{KARMS},i}$, so that positive values favor KARMS. Pairwise comparisons use the two-sided

Table 19: Component-level runtime audit over the complete JSSP/FJSP evaluation. Component entries report mean wall-clock seconds, with the fraction of mean end-to-end (E2E) time in parentheses. "Other" combines validation and residual overhead. Percentages are ratios of component means to mean E2E time and may differ slightly from 100% due to rounding.

Set	N	Construct policy	Construct CP-LH	Improve policy	Improve CP repair	Other	E2E
JSSP	242	28.60 (18.9%)	102.09 (67.3%)	3.93 (2.6%)	15.85 (10.5%)	1.17 (0.8%)	151.65
FJSP (all)	258	7.82 (17.9%)	26.52 (60.6%)	1.06 (2.4%)	8.10 (18.5%)	0.26 (0.6%)	43.75
Classical FJSP	138	11.56 (20.1%)	37.15 (64.6%)	0.87 (1.5%)	7.63 (13.3%)	0.32 (0.6%)	57.53
Hurink FJSP	120	3.53 (12.6%)	14.30 (51.2%)	1.28 (4.6%)	8.63 (30.9%)	0.18 (0.6%)	27.92
All	**500**	**17.88 (18.6%)**	**63.10 (65.7%)**	**2.45 (2.6%)**	**11.85 (12.3%)**	**0.70 (0.7%)**	**95.98**

Table 20: Runtime decomposition on the 50 JSSP instances corresponding to the representative settings reported in the main paper.

Component	Mean (s)	E2E share
Construct policy	7.89	14.4%
Construct CP lookahead	31.90	58.3%
Improve policy	1.95	3.6%
Improve local CP repair	12.58	23.0%
Other	0.41	0.7%
End-to-end	**54.73**	**100%**

Wilcoxon signed-rank test with Pratt handling of zero differences. When several related hypotheses are tested within the same comparison family, the resulting p-values are adjusted with Holm's procedure at $\alpha = 0.05$.

We additionally report matched-pairs rank-biserial correlation r_{rb} as a nonparametric effect size, with positive values defined in favor of KARMS. Paired mean differences are summarized with nonparametric bootstrap confidence intervals using 20,000 instance-level resamples and a fixed random seed. When three matched methods are available on the same instances, a Friedman omnibus test is applied before pairwise follow-up comparisons. No significance test is constructed from published aggregate means alone.

Paired comparison with NDS-S. Unlike most published learning-based baselines, NDS-S provides official per-instance outputs that can be matched exactly to the KARMS evaluation instances. We use the validation-selected NDS-S models (`jssp20x20` for JSSP and `fjssp20x10_op15` for FJSP) and recompute all NDS-S gaps using the same A03 instance reference values as KARMS. For each instance, we define $\Delta_i = \mathrm{Gap}_{\mathrm{NDS-S},i} - \mathrm{Gap}_{\mathrm{KARMS},i}$, so positive values favor KARMS. Pairwise tests use a two-sided Wilcoxon signed-rank test with Pratt handling of zero differences. The six related comparisons are corrected jointly using Holm's procedure. We additionally report Pratt rank-biserial correlation and a 95% bootstrap confidence interval for the paired mean difference using 20,000 resamples and seed 20260916.

Table 21: Paired comparison between KARMS and NDS-S using identical instances and the same A03 reference values. Δ is $\mathrm{Gap}_{\mathrm{NDS-S}} - \mathrm{Gap}_{\mathrm{KARMS}}$ in percentage points, so positive values favor KARMS. W/T/L denotes lower/equal/higher gap for KARMS.

Set	N	KARMS	NDS-S	Δ	95% CI	W/T/L	p_{Holm}	r_{rb}
JSSP-242	242	12.171	14.712	+2.542	[1.484, 3.649]	116/10/116	0.107	+0.171
JSSP-162	162	9.001	9.411	+0.410	[−0.495, 1.328]	61/10/91	0.340	−0.086
Hurink-120	120	4.288	2.874	−1.414	[−2.167, −0.656]	31/10/79	< 0.001	−0.441
MK01-10	10	9.502	3.535	−5.967	[−9.986, −2.355]	1/3/6	0.125	−0.745
Barnes	21	4.155	3.027	−1.129	[−2.550, 0.306]	7/2/12	0.329	−0.398
Dauzère-Pérès	18	10.349	8.069	−2.280	[−4.556, −0.083]	7/0/11	0.329	−0.392

After Holm correction, only the Hurink-120 comparison remains statistically significant, with NDS-S obtaining lower gaps than KARMS (2.874% versus 4.288%, $p_{\mathrm{Holm}} < 0.001$, $r_{\mathrm{rb}} = -0.441$). On the common JSSP-162 subset, the mean gaps are close (9.001% versus 9.411%) and the paired difference is not significant ($p_{\mathrm{Holm}} = 0.340$). Over all 242 JSSP instances, KARMS has the lower mean gap (12.171% versus 14.712%), but the Wilcoxon comparison does not remain significant after multiplicity correction ($p_{\mathrm{Holm}} = 0.107$).

For JSSP-242, the bootstrap confidence interval for the paired mean difference excludes zero even though the Holm-corrected Wilcoxon test is not significant. These summaries address different aspects of the paired differences: the bootstrap interval concerns the arithmetic mean magnitude, whereas the Wilcoxon test is rank-based and is the multiplicity-controlled inferential criterion used here. We therefore do not claim a statistically significant JSSP-242 advantage. The same conservative interpretation is applied to MK01-10, Barnes, and Dauzère-Pérès, none of which remains significant after Holm correction.

Comparison with standalone CP. Table 22 compares KARMS with the locally evaluated standalone CP solver using the common 1,800 s solver configuration. CP obtains substantially lower gaps on both JSSP and classical FJSP. The paired differences are significant after Holm correction and have very large rank-biserial effect sizes in favor of CP. The same conclusion holds on the 162-instance JSSP subset used for common-method comparisons.

Table 22: Paired comparison with standalone CP. W/T/L denotes instances where KARMS is better/equal/worse.

Set	N	Gap: CP $\rightarrow$ KARMS	W/T/L	p_{Holm}	r_{rb}
JSSP	242	$1.58 \rightarrow 12.17$	0/13/229	< 0.001	-0.997
FJSP	138	$1.02 \rightarrow 12.39$	0/18/120	< 0.001	-0.982

These results should be interpreted together with runtime: standalone CP is given a much larger per-instance search budget, whereas KARMS targets a different quality-time operating point rather than matching the final solution quality of long-running CP.

Comparison with external greedy initializers. We also test whether Destroy-to-Improve produces systematic gains over the complete greedy schedules supplied by the external DANIEL and ReSched models. These comparisons concern the external greedy parent schedules used as initializers and are distinct from the sampling-based DANIEL and ReSched results reported elsewhere in the paper.

Across the combined 120 Hurink instances, KARMS significantly reduces the gap relative to both initializers, with large rank-biserial effects. The result is consistent across all three Hurink subsets for DANIEL. For ReSched, the improvement is significant on *edata* and *rdata*, while the smaller improvement on *vdata* is not significant after Holm correction. On the 15 Brandimarte instances, the observed improvements are positive but are not statistically significant.

Table 23: Paired improvement over external greedy parent schedules. Positive ΔGap and positive r_{rb} favor KARMS.

Set	Initializer	ΔGap	W/T/L	p_{Holm}	r_{rb}
Hurink-120	DANIEL-greedy	$+6.43$	108/2/10	< 0.001	$+0.93$
Hurink-120	ReSched-greedy	$+5.50$	94/7/19	< 0.001	$+0.80$
Brandimarte-15	DANIEL-greedy	$+2.06$	9/2/4	0.505	$+0.29$
Brandimarte-15	ReSched-greedy	$+2.04$	11/0/4	0.505	$+0.35$

For the Hurink subsets, Friedman tests over KARMS, DANIEL-greedy, and ReSched-greedy reject the equal-performance hypothesis for *edata*, *rdata*, *vdata*, and the combined 120-instance set ($p < 0.001$ in all cases). Pairwise post-hoc comparisons show significant improvements over DANIEL-greedy on all three subsets. Against ReSched-greedy, the improvements are significant on *edata* and *rdata*, whereas the *vdata* comparison is not significant after Holm correction ($p_{\mathrm{Holm}} = 0.121$).

No paired significance test is reported for L2S, TBGAT, ScheduleNet, RL-GNN, L2D, HGNN, MIStar, Residual Scheduling, PRISMA-S, the sampling-based DANIEL and ReSched results, or the published long-budget CP results when only aggregate values are available. NDS-S is treated separately above because official per-instance outcomes can be matched exactly to the KARMS evaluation instances and normalized with the same reference values.

M Complete Benchmark Results

The main paper reports compact comparisons restricted to benchmark settings with meaningful shared coverage. This appendix preserves the complete evaluation matrices used in the study, including benchmark families for which one or more published methods do not provide corresponding results. Such missing entries indicate unavailable comparison evidence and are not interpreted as wins for KARMS.

The purpose of these tables is therefore twofold. First, they make the full benchmark coverage and the provenance of the compact main-paper comparisons transparent. Second, they document how the frozen KARMS pipeline behaves on additional problem families and scales even when a direct method-to-method comparison is unavailable.

M.1 Complete JSSP Results

The complete JSSP evaluation contains 242 instances from the ABZ, DMU, FT, LA, ORB, SWV, Taillard, and YN benchmark families. The main paper reports a compact set of representative competitive settings, whereas Table 25 provides the complete size-level comparison used in the evaluation.

The compared learning-based methods also differ in their test-time role. L2D, RL-GNN, ScheduleNet, and ReSched construct schedules directly, whereas L2S, TBGAT, and NDS-S improve complete schedules. NDS-S performs iterative neural deconstruction followed by heuristic reconstruction under a 60 s search budget, while KARMS combines learned construction, bounded CP lookahead, and CP-backed complete-schedule improvement. Table 24 summarizes this distinction; detailed baseline configurations and inference protocols are given in Appendix L.5.

Table 24: Inference role of the principal learning-based methods in the JSSP comparison.

Method	Role	Test-time procedure
L2D	Construct	Sequential learned dispatching
RL-GNN	Construct	Sequential learned scheduling
ScheduleNet	Construct	Learned agent-task scheduling
ReSched	Construct	Sequential operation-machine scheduling
L2S	Improve	Learned local search from an initial complete schedule
TBGAT	Improve	Topology-aware learned local search
NDS-S	Improve	Neural deconstruction followed by heuristic reconstruction (~ 60 s)
KARMS	Construct + Improve	Learned construction + bounded CP lookahead + CP-backed complete-schedule repair

NDS-S provides matching coverage on all 242 JSSP instances using its validation-selected 20×20 JSSP model and the reported approximately 60 s improvement budget. After recomputing its gaps with the same instance reference values used by KARMS, NDS-S obtains a mean gap of 14.71% over the complete 242-instance evaluation and 9.41% over the common 162-instance non-DMU subset. On the latter subset, KARMS obtains 9.00%, behind L2S-5000 (6.90%) but ahead of NDS-S (9.41%), L2S-500 (11.05%), ScheduleNet (16.46%), RL-GNN (20.30%), and L2D (24.64%). The paired KARMS-NDS-S difference on this subset is not statistically significant after Holm correction ($p_{\mathrm{Holm}} = 0.340$; Appendix L.7).

The 80 DMU instances broaden the evaluation toward larger JSSP configurations and provide a direct additional comparison with NDS-S, which obtains a mean gap of 25.45% on this family under the common reference convention. Over all 242 JSSP instances, KARMS has the lower mean gap

Table 25: Complete JSSP benchmark results. Each available cell reports mean Gap (%) / reported Time. "-" denotes an unavailable corresponding result in the source used for that method. NDS-S gaps are recomputed from its official validation-selected model outputs using the same instance reference values as KARMS; its reported search budget is approximately 60 s per instance. Published timing scopes may differ from KARMS's end-to-end runtime; see Appendix L.6.

(a) Taillard, ABZ, and FT

	Taillard								ABZ		FT		
Method	15×15	20×15	20×20	30×15	30×20	50×15	50×20	100×20	10×10	20×15	6×6	10×10	20×5
CP	0.1 / 7.7m	0.2 / 0.8h	0.7 / 1.0h	2.1 / 1.0h	2.8 / 1.0h	0.0 / 0.4h	2.8 / 0.9h	3.9 / 1.0h	0.0 / 0.8s	1.0 / 1.0h	0.0 / 0.1s	0.0 / 4.1s	0.0 / 4.8s
L2D	24.7 / 0.4s	30.0 / 0.6s	28.8 / 1.1s	30.5 / 1.3s	32.9 / 1.5s	20.0 / 2.2s	23.9 / 3.6s	12.9 / 28.2s	14.8 / 0.1s	24.9 / 0.6s	14.5 / 0.1s	21.0 / 0.2s	36.3 / 0.2s
RL-GNN	20.1 / 3.0s	24.9 / 7.0s	29.2 / 12.0s	24.7 / 24.7s	32.0 / 38.0s	15.9 / 1.9m	21.3 / 3.2m	9.2 / 28.2m	10.1 / 0.5s	29.0 / 7.3s	29.1 / 0.1s	22.8 / 0.5s	14.8 / 1.3s
ScheduleNet	15.3 / 3.5s	19.4 / 6.6s	17.2 / 11.0s	19.1 / 17.1s	23.7 / 28.3s	13.9 / 52.5s	13.5 / 1.6m	6.7 / 7.4m	6.1 / 0.7s	20.5 / 6.6s	7.3 / 0.2s	19.5 / 0.8s	28.6 / 1.6s
L2S-500	9.3 / 9.3s	11.6 / 10.1s	12.4 / 10.9s	14.7 / 12.7s	17.5 / 14.0s	11.0 / 16.2s	13.0 / 22.8s	7.9 / 50.2s	2.8 / 7.4s	11.9 / 10.2s	0.0 / 6.8s	9.9 / 7.5s	6.1 / 7.4s
L2S-5000	6.2 / 1.5m	8.3 / 1.7m	9.0 / 1.9m	9.0 / 2.0m	12.6 / 2.4m	4.6 / 2.8m	6.5 / 3.8m	3.0 / 8.4m	1.4 / 1.3m	8.6 / 1.7m	0.0 / 1.1m	5.6 / 1.2m	1.1 / 1.2m
TBGAT-1000	6.1 / 24.9s	8.7 / 28.7s	9.0 / 34.1s	10.9 / 33.7s	14.0 / 37.3s	7.5 / 46.9s	9.4 / 47.5s	4.9 / 1.4m	1.1 / 17.9s	10.1 / 25.3s	-	-	-
TBGAT-5000	4.6 / 2.1m	6.3 / 2.3m	7.0 / 2.7m	6.8 / 2.7m	9.7 / 2.9m	2.6 / 3.9m	5.0 / 3.9m	1.2 / 6.7m	0.8 / 1.5m	6.6 / 2.1m	-	-	-
NDS-S	4.02 / 60s	6.82 / 60s	6.62 / 60s	11.22 / 60s	12.00 / 60s	20.72 / 60s	15.33 / 60s	25.03 / 60s	0.16 / 60s	9.45 / 60s	0.00 / 60s	2.69 / 60s	4.29 / 60s
ReSched	15.7 / -	19.7 / -	16.3 / -	21.5 / -	22.5 / -	16.1 / -	15.6 / -	9.6 / -	-	-	-	-	-
KARMS	3.5 / 35.5s	6.9 / 56.1s	9.2 / 1.4m	12.9 / 1.7m	17.1 / 2.8m	14.0 / 3.8m	17.6 / 7.2m	11.0 / 21.1m	4.5 / 7.1s	9.8 / 1.1m	9.1 / 1.3s	13.3 / 7.1s	3.2 / 22.9s

(b) LA, SWV, ORB, and YN

	LA								SWV			ORB	YN
Method	10×5	15×5	20×5	10×10	15×10	20×10	30×10	15×15	20×10	20×15	50×10	10×10	20×20
CP	0.0 / 0.1s	0.0 / 0.2s	0.0 / 0.5s	0.0 / 0.4s	0.0 / 21.0s	0.0 / 12.9m	0.0 / 13.7s	0.0 / 30.1s	0.1 / 0.8h	2.5 / 1.0h	1.6 / 0.5h	0.0 / 4.8s	0.5 / 1.0h
L2D	14.3 / 0.1s	5.5 / 0.1s	4.2 / 0.2s	21.9 / 0.1s	24.6 / 0.2s	24.7 / 0.4s	8.4 / 0.7s	27.1 / 0.4s	41.4 / 0.3s	40.6 / 0.6s	30.8 / 1.2s	31.8 / 0.1s	22.1 / 0.9s
RL-GNN	16.1 / 0.2s	1.1 / 0.5s	2.1 / 1.2s	17.1 / 0.5s	22.0 / 1.5s	27.3 / 3.3s	6.3 / 11.3s	21.4 / 2.8s	28.4 / 3.4s	29.4 / 7.2s	16.8 / 51.5s	21.8 / 0.5s	24.8 / 11.0s
ScheduleNet	12.1 / 0.6s	2.7 / 1.2s	3.6 / 1.9s	11.9 / 0.8s	14.6 / 2.0s	15.7 / 4.1s	3.1 / 9.3s	16.1 / 3.5s	34.4 / 3.9s	30.5 / 6.7s	25.3 / 25.1s	20.0 / 0.8s	18.4 / 11.2s
L2S-500	2.1 / 6.9s	0.0 / 6.8s	0.0 / 7.1s	4.4 / 7.5s	6.4 / 8.0s	7.0 / 8.9s	0.2 / 10.2s	7.3 / 9.0s	29.6 / 8.8s	25.5 / 9.7s	21.4 / 12.5s	8.2 / 7.4s	12.4 / 11.7s
L2S-5000	1.8 / 1.2m	0.0 / 1.2m	0.0 / 1.2m	0.9 / 1.3m	3.4 / 1.3m	2.6 / 1.4m	0.0 / 1.7m	5.9 / 1.5m	17.8 / 1.4m	17.0 / 1.7m	17.1 / 2.1m	3.8 / 1.3m	8.7 / 1.9m
TBGAT-1000	1.6 / 3.1s	0.0 / 1.9s	0.0 / 3.3s	1.8 / 18.2s	3.5 / 21.4s	4.0 / 22.8s	0.0 / 6.3s	5.3 / 24.5s	19.2 / 31.2s	20.1 / 34.3s	18.5 / 59.5s	5.7 / 20.8s	-
TBGAT-5000	1.5 / 9.8s	0.0 / 9.2s	0.0 / 16.1s	1.4 / 1.6m	2.4 / 1.8m	2.1 / 52.0s	0.0 / 16.9s	4.4 / 2.0m	11.1 / 2.5m	15.7 / 2.7m	14.5 / 4.9m	4.5 / 1.7m	-
NDS-S	0.00 / 60s	0.00 / 60s	0.00 / 60s	0.30 / 60s	2.28 / 60s	3.84 / 60s	2.06 / 60s	3.06 / 60s	17.97 / 60s	20.03 / 60s	15.97 / 60s	2.24 / 60s	10.39 / 60s
ReSched	-	-	-	-	-	-	-	-	-	-	-	-	-
KARMS	4.2 / 2.3s	0.6 / 4.7s	0.6 / 8.6s	2.6 / 7.1s	4.9 / 13.5s	4.1 / 32.2s	1.1 / 53.1s	3.5 / 35.4s	11.1 / 31.6s	18.4 / 52.8s	12.5 / 2.1m	4.6 / 9.6s	11.1 / 1.2m

(c) DMU

Method	20×15	20×20	30×15	30×20	40×15	40×20	50×15	50×20
NDS-S	12.59 / 60s	10.96 / 60s	27.40 / 60s	18.72 / 60s	29.45 / 60s	35.00 / 60s	30.85 / 60s	38.62 / 60s
KARMS	10.1 / 58.2s	11.2 / 1.4m	17.5 / 1.8m	17.9 / 2.8m	18.4 / 2.8m	25.6 / 4.6m	21.7 / 4.1m	26.3 / 6.2m

Notes. "-" indicates that the method does not report the corresponding result in the comparison source used here. NDS-S gaps are recomputed from the official validation-selected model outputs using the same instance reference values as KARMS; its search budget is approximately 60 s per instance. On DMU, matching results are available for KARMS and NDS-S, whereas the other listed learned methods do not provide corresponding values. The CP row is reported as a full-instance reference; the paired statistical CP analysis uses the separate configuration described in Appendix L.7.

(12.17% versus 14.71%), although the paired Wilcoxon comparison does not remain significant after Holm correction ($p_{\mathrm{Holm}} = 0.107$). Across the 86 size-level comparisons with the construct-only baselines L2D, RL-GNN, ScheduleNet, and ReSched, KARMS obtains a lower mean gap in 79 cases (91.9%).

M.2 Complete FJSP Results

The FJSP evaluation covers the Brandimarte, Hurink, Barnes, Dauzère-Pérès, Behnke, Fattahi, and Kacem benchmark families. The main paper restricts direct method-to-method comparison to Brandimarte MK01-10 and the three Hurink variants, where the principal published methods provide matching results. Table 27 reports the broader family-level evaluation.

The compared methods differ substantially in their inference role and test-time procedure. HGNN, DANIEL, Residual Scheduling, ReSched, and PRISMA-S construct schedules directly, whereas NDS-S and MIStar improve complete schedules. KARMS combines both roles through learned construction, bounded CP lookahead, and CP-backed complete-schedule repair, reusing the same frozen models across the evaluated FJSP families. Table 26 summarizes these differences.

Beyond the four main shared-coverage groups, NDS-S also provides matching results for Barnes and Dauzère-Pérès, while PRISMA-S provides matching results for Dauzère-Pérès. Behnke, Fattahi, Kacem, and Brandimarte MK11-15 remain additional KARMS evaluation coverage among the methods listed here and are not interpreted as baseline wins.

Table 26: Inference roles and test-time procedures of the principal learning-based methods in the FJSP comparison.

Method	Role	Test-time procedure
HGNN-samp	Construct	Heterogeneous-graph DRL with 100 sampled trajectories
DANIEL-samp	Construct	Dual-attention DRL with 100 sampled trajectories
Residual RS+100	Construct	Residual-state scheduling with 100 constructed solutions
ReSched-samp	Construct	Transformer-based scheduling with 100 sampled trajectories
NDS-S	Improve	Learned neural deconstruction followed by heuristic reconstruction
PRISMA-S	Construct	CP-guided training with sampling-based constructive inference
MIStar-400	Improve	Memory-enhanced improvement search with 400 iterations
KARMS	Construct + Improve	Deterministic construction + bounded CP lookahead + CP-backed repair

Table 27: Complete FJSP results by benchmark family. Each available cell reports mean Gap (%) / reported Time. "-" denotes an unavailable corresponding result in the source used for that method. Published timing scopes may differ from KARMS's end-to-end runtime.

Method	MK01-10	Hurink-e	Hurink-r	Hurink-v	Barnes	Dauzère	Behnke	Fattahi	Kacem	Brandimarte-15
n	10	40	40	40	21	18	60	20	4	15
CP 1800s	1.0 / 24.1m	0.0 / 15.0m	0.2 / 23.3m	-0.0 / 10.7m	-	-	-	-	-	-
HGNN-samp	13.6 / 1.3s	16.3 / 1.4s	11.4 / 1.4s	3.3 / 1.4s	-	-	-	-	-	-
DANIEL-samp	13.0 / 1.3s	14.4 / 1.4s	12.1 / 1.4s	3.8 / 1.4s	-	-	-	-	-	-
Residual RS+100	15.4 / -	7.9 / -	4.7 / -	0.7 / -	-	-	-	-	-	-
ReSched-samp	6.6 / -	8.1 / -	5.0 / -	0.8 / -	-	-	-	-	-	-
PRISMA-S	10.01 / 2.0s	7.72 / 1.9s	6.83 / 2.0s	1.69 / 2.1s	-	7.22 / 3.4s	-	-	-	-
NDS-S	3.535 / 60.03s	2.286 / 60.03s	2.709 / 60.04s	3.627 / 60.04s	3.027 / 60.04s	8.069 / 60.05s	-	-	-	-
MIStar-400	3.0 / 1.3m	6.8 / 1.4m	2.4 / 1.4m	0.2 / 1.5m	-	-	-	-	-	-
KARMS	9.5 / 18.8s	5.7 / 20.3s	5.2 / 26.1s	2.0 / 29.3s	4.2 / 16.2s	10.3 / 1.1m	17.7 / 1.5m	6.0 / 0.7s	32.1 / 3.1s	8.3 / 28.4s

On the four main shared-coverage groups, MIStar-400 obtains the lowest listed learned gap on MK01-10 (3.0%), Hurink-r (2.4%), and Hurink-v (0.2%), while NDS-S is strongest on Hurink-e (2.286%). Across all 120 Hurink instances, KARMS achieves an instance-weighted mean gap of 4.29%, compared with 2.87% for NDS-S, 3.13% for MIStar-400, 4.43% for Residual RS+100, 4.63% for ReSched-samp, 5.42% for PRISMA-S, 10.10% for DANIEL-samp, and 10.33% for HGNN-samp. NDS-S and PRISMA-S additionally provide direct overlap on Barnes and/or Dauzère-Pérès, while Behnke, Fattahi, Kacem, and Brandimarte MK11-15 primarily document the broader evaluation coverage of the same frozen KARMS models. The paired comparison with NDS-S is significant only on Hurink-120 after Holm correction ($\Delta = -1.414$ percentage points, $p_{\mathrm{Holm}} < 0.001$, $r_{\mathrm{rb}} = -0.441$); the corresponding MK01-10, Barnes, and Dauzère-Pérès comparisons do not remain significant after correction.

M.3 Results by Problem Scale

The training curriculum contains synthetic JSSP and FJSP instances only up to 30×20. To examine scale transfer separately from the historical Construct ablation, Table 28 summarizes representative JSSP size groups ranging from 10×10 to 100×20. The final two rows lie beyond the reported training-size range.

The comparison uses raw Construct output and the final KARMS schedule. Because the final schedule is reached after the complete post-construction pipeline, the reported reduction includes all enabled post-construction processing and should not be attributed exclusively to Destroy-to-Improve.

Table 28: Raw Construct and final KARMS gaps across representative JSSP sizes. The final two rows are beyond the reported training-size range.

Size	N	Construct Gap (%)	Final Gap (%)	ΔGap
10×10	18	18.34	4.55	13.79
15×15	15	19.53	3.50	16.03
20×20	24	22.97	10.34	12.64
30×20	20	30.89	17.49	13.40
50×20	20	27.40	21.99	5.41
100×20	10	11.66	10.99	0.66

The post-construction pipeline remains useful beyond the training range, although the magnitude of the reduction decreases on the largest reported size. At 50×20, the mean gap falls by 5.41 percentage points , whereas at 100×20 the reduction is only 0.66 points. This suggests that transfer to larger schedules remains possible without retraining, but the amount of recoverable local improvement depends on the structure and quality of the incumbent rather than increasing automatically with problem size.

N Ablation and Component Analysis

This appendix distinguishes historical component progression from controlled ablation and diagnostic evidence. Historical Construct experiments document how the constructive policy changed as insertion, structured training, knowledge augmentation, and bounded CP lookahead were introduced. Controlled diagnostics then examine candidate ranking, budgeted operation selection, learned selection versus heuristic alternatives, the CP reconstruction backend, and the two memory sources, while the FFSP experiment provides a stage-wise view of construction, lookahead, and complete-schedule improvement after transfer to a different shop structure.

We use the term *ablation* conservatively. A comparison is interpreted causally only when the evaluated variants share the same starting schedules and differ in the intended component under an otherwise matched protocol. Historical development runs, candidate-ranking examples, and transfer analyses are labeled accordingly rather than presented as one-factor-at-a-time causal evidence.

N.1 Construct Design Progression

The Construct policy was developed progressively rather than by increasing network capacity alone. Table 29 summarizes the role of the main changes. The original RL policy is append-only: once an operation-machine action is selected, the operation is assigned to the best feasible position at the current end of that machine sequence and cannot exploit an earlier idle gap. Insertion actions remove this restriction by allowing a selected operation to occupy an earlier feasible gap. Knowledge augmentation then enriches the scheduling state with structural and heuristic context, while curriculum training and distillation expose the policy progressively to more difficult scheduling decisions. The Knowledge Graph stage denotes the mature knowledge- and memory-augmented Construct configuration used before bounded CP lookahead is applied.

N.2 Construct-Only Ablation

Table 30 reports representative historical Construct-only results at the major validated milestones. Because these models were developed sequentially, this experiment is best interpreted as a component progression rather than a strict one-factor-at-a-time ablation. In particular, the table shows how the policy changes as insertion-capable construction, structured training, the mature Knowledge-augmented representation, and finally CP lookahead are introduced. The final two sizes, 50×20 and 100×20, are outside the reported training range of up to 30×20.

The main reduction occurs when the simple constructive policy is combined with the structured training procedure: curriculum and distillation sharply improve the schedule produced before any complete-schedule repair. The subsequent knowledge-augmented stage provides a smaller addi-

Table 29: Progressive design of the Construct policy. The rows describe conceptual additions

Stage	Main change
Append-only RL baseline	Selects an operation-machine action and places it at the best feasible end position of the selected machine sequence.
+ Feasible insertion	Allows the selected operation to use an earlier feasible idle gap instead of being restricted to the machine tail.
+ Knowledge augmentation	Adds structural scheduling information together with heuristic and memory-based context to the learned decision process.
+ Curriculum and distillation	Trains progressively across increasingly difficult synthetic JSSP/FJSP instances and transfers guidance from stronger scheduling decisions.
Full knowledge model	Uses the mature knowledge-augmented graph and memory representation adopted by the final Construct policy.
+ CP lookahead	Re-evaluates a small set of highly ranked actor decisions through short bounded CP subproblems before committing the action.

Table 30: Construct-only ablation across problem sizes on 10 unseen synthetic instances and not classical benchmarks. Entries are reported mean makespans. The horizontal separation marks sizes beyond the reported training range. CP is shown only as a reference and is not a learned Construct component.

Size	Append-only RL	Insertion RL no curriculum	Curriculum + distill.	Full knowledge	+ CP lookahead	CP reference
6×6	837	1082	520	498	**478**	478
10×10	1441	2537	980	1005	**913**	834
15×15	2197	4159	1468	1471	**1376**	1241
20×20	5485	5095	2011	2003	**1920**	1599
30×20	6848	6612	2410	**2327**	2382	1961
50×20	10825	13390	3562	3509	**3472**	2995
100×20	21951	26354	6121	6118	**6039**	5593

tional refinement at full training budget; the controlled study below shows that its contribution becomes substantially larger when the available training budget is reduced. Short CP lookahead improves six of the seven reported sizes and is especially useful on the larger 50×20 and 100×20 instances; at 30×20, the underlying Knowledge-augmented policy is slightly better without lookahead. This variation is expected from a bounded lookahead mechanism and motivates treating CP as a selective decision aid rather than as a replacement for the learned policy.

N.3 Knowledge Data Efficiency, Cross-Scale Generalization, and Seed Robustness

To isolate the contribution of knowledge augmentation, we train matched Construct policies with and without the additional heuristic, structural, and memory-based knowledge signals under 25%, 50%, and 100% of the training budget. All other Construct components and training conditions are kept matched, and evaluation uses deterministic Construct inference only, without CP lookahead or Destroy-to-Improve. The 10×10, 20×20, and 30×20 settings are within the reported training range, whereas 50×20 and 100×20 test cross-scale generalization.

Table 31: Relative reduction in mean makespan obtained by the knowledge-augmented Construct policy over its matched no-knowledge counterpart. Positive values favor Knowledge. The first three columns are within the training-size range; the last two are unseen larger scales.

Training budget	10×10	20×20	30×20	50×20	100×20
25%	8.8%	10.7%	12.8%	14.9%	13.2%
50%	4.0%	1.1%	2.3%	2.4%	3.9%
100%	1.9%	0.2%	1.6%	3.0%	4.5%

In the evaluated runs, the knowledge-augmented variants achieve lower mean makespan at every tested budget and size . The effect is strongest under the 25% budget, where the reduction ranges from 8.8% to 14.9%, suggesting that structured scheduling knowledge is particularly useful when learning exposure is limited. The advantage remains positive at the full training budget, showing that the mean advantage remains positive even at the full training budget.

The benefit also persists beyond the training-size range. At 50×20 and 100×20, Knowledge improves mean makespan at every budget, with gains of 3.0% and 4.5%, respectively, even at the full training budget. Together, these results support the intended role of knowledge augmentation as an inductive bias: it improves Construct under reduced training budgets and remains useful when the same learned policy is applied to substantially larger unseen instances.

Training-seed robustness. We additionally assess sensitivity to Construct training randomness using three independently trained Construct models with the same four-cycle training recipe and seeds 17, 42, and 20260828. Each model is evaluated on the same fixed 55-instance set. To isolate variation due to Construct training, the Destroy-to-Improve model is kept fixed across all runs. We report performance after greedy Construct, after bounded CP lookahead, and after the complete pipeline with the same frozen Improve model.

For this controlled study, each training run uses four curriculum cycles with the same 12 phases per cycle. Each phase contains five sampled training instances and five held-out validation instances. From the second cycle onward, two rehearsal instances are additionally included per phase. This gives 240 sampled training instances and 240 validation evaluations per seed, together with approximately 72 rehearsal instances across the last three cycles. Training uses 11 rollout workers, PPO batch size 4, and three anchor instances per curriculum bucket. The matched runs are repeated with the same seeds,

Table 32: Training-seed robustness on the same 55-instance evaluation set. Construct is trained independently for each seed, while the Improve model is kept fixed. Values are mean reference gaps (%); lower is better.

Seed	Construct	+ CP lookahead	Final KARMS
42	21.00	17.78	5.60
17	22.51	18.58	7.27
20260828	16.61	14.43	4.01
Mean $\pm$ std.	20.04 ± 3.06	16.93 ± 2.20	5.63 ± 1.63

Although performance varies across independently trained Construct models, the same qualitative behavior is preserved in all three runs: CP lookahead improves the initial schedule, and the frozen Destroy-to-Improve stage provides a further substantial reduction. The final gaps remain within a relatively narrow range of 4.01%-7.27%, providing evidence that the reported pipeline behavior is not specific to a single Construct initialization. The seed study varies Construct while keeping the final Improve model fixed; variability across independently trained Improve models remains future work

N.4 Post-Construction Pipeline Contribution

The historical Construct ablation above concerns the production of the initial schedule. The main paper additionally compares raw Construct output with final KARMS output over all 500 established JSSP/FJSP benchmark evaluations. Across these instances, the mean reference gap decreases from 22.71% after raw construction to 10.34% in the final schedule.

This difference should be interpreted as the contribution of the complete post-construction inference pipeline rather than as an isolated Destroy-to-Improve effect. Bounded CP lookahead can already modify Construct decisions before the complete schedule is passed to the improvement stage. A strict Destroy-to-Improve comparison would instead start from the incumbent immediately after CP lookahead and compare it with the schedule returned by the improvement stage.

The distinction is useful for interpreting the architecture. Construct makes global scheduling decisions incrementally, CP lookahead locally re-evaluates a small set of uncertain construction choices,

Table 33: Controlled operation-selection ablation on the same 55 instances. All variants use the same Construct and Improve checkpoints, Candidate Head, and local CP configuration; only the released operation set changes.

Budget	Operation selection	Mean Gap (%)	Mean time (s)
$\rho = 0.85,\ B_{\max} = 250$	Full proposed set	7.30	7.09
	Random-B	6.54	5.17
	Learned Top-B	**5.47**	**4.00**
$\rho = 0.25,\ B_{\max} = 8$	Full proposed set	7.78	6.97
	Random-B	13.71	2.37
	Learned Top-B	**5.92**	**2.30**

and Destroy-to-Improve operates only after the complete schedule reveals its realized critical blocks, bottleneck machines, low-slack regions, and other structures. The FFSP component analysis in Appendix N.9 reports these stages separately, while the controlled heuristic replacement experiment below is designed to isolate the learned selection and CP-reconstruction responsibilities more directly.

N.5 Budgeted Operation-Selection Ablation

Complementing the candidate-ranking diagnostic reported in the following subsection, we isolate the subsequent *what* decision: which operations within the selected candidate should be released to CP. We use the same fixed 55-instance subset as the memory ablation, containing Taillard 15×15, ORB 10×10, LA 15×15, Brandimarte MK01-10, and Hurink edata LA01-20. All variants use the same Construct and Improve checkpoints, Candidate Head, and local CP configuration (15 s, one worker, fixed seed); only the operation-set rule is changed.

We evaluate two repair-budget settings. The first matches the reported KARMS configuration, with $\rho = 0.85$ and $B_{\max} = 250$. The second is a tighter stress-test setting, with $\rho = 0.25$ and $B_{\max} = 8$, designed to make explicit subset selection more restrictive. Under each budget, Learned Top-B selects operations using the learned Operation-Selection Transformer, Random-B selects the same number of operations uniformly at random, and Full proposed set releases all proposed operations before closure. The subset was fixed before running the component ablations to provide a computationally manageable mixture of JSSP/FJSP instances, multiple benchmark families, and in- and out-of-distribution schedule structures; it was not selected based on ablation outcomes.

The deployment-matched setting already shows a clear benefit from learned operation selection. With $\rho = 0.85$ and $B_{\max} = 250$, Learned Top-B reduces the mean gap to 5.47%, compared with 6.54% for Random-B and 7.30% when all proposed operations are released. It also reduces mean Improve time to 4.00 s, compared with 5.17 s and 7.09 s, respectively. Thus, the learned selector improves both repair quality and computational cost under the same operation-budget setting used by the reported pipeline.

The tighter $\rho = 0.25$, $B_{\max} = 8$ setting provides a complementary stress test in which explicit subset selection is more restrictive. Under this budget, Learned Top-B remains substantially better than Random-B (5.92% versus 13.71%) at essentially identical runtime, and also improves over releasing the full proposed set. Together, the two settings show that the gain is not explained only by releasing fewer operations: the learned scores identify subsets that are more useful for subsequent CP reconstruction.

N.6 Candidate-Ranking Diagnostic

To better understand the role of learned candidate selection, we additionally inspect a controlled diagnostic in which different ranking policies operate on exactly the same candidate banks and use the same frozen repair backend. This experiment is not intended as a benchmark-level ablation; instead, it separates limitations of candidate generation from limitations of candidate ranking.

The diagnostic reveals two distinct limitations. In some schedules, the fixed candidate bank already contains an improving region but the learned top-1 decision does not select it; in other cases, an

Table 34: Representative candidate-ranking diagnostics. Oracle gain is the best relative makespan reduction available anywhere in the fixed candidate bank. A zero top-1 gain indicates that the selected candidate does not improve the incumbent.

Schedule	Oracle gain	Supervised top-1	PPO top-1	PPO top-3
JSSP 6×6 A	0.0113	0.0113	0	0
JSSP 6×6 B	0.0344	0	0	0.0344
JSSP 20×20	0.0185	0	0	0.0185

improving candidate remains within the highest-ranked set but below rank one. These are ranking limitations rather than failures of candidate generation. Conversely, when the best candidate in the entire bank has zero gain, improving the ranker alone cannot recover a better repair from the current bank.

The diagnostic therefore supports treating candidate generation and candidate ranking as separate components of Destroy-to-Improve. It is also consistent with the use of CP-outcome supervision in the final Candidate Head adaptation. Because the diagnostic contains only a small fixed set of schedules and historical ranking policies, it is used to explain failure modes rather than to establish benchmark-level superiority of one ranking objective over another.

N.7 Learned Selection and CP Repair versus Heuristic Alternatives

The previous analyses establish that post-construction processing improves the schedules produced by Construct, but they do not by themselves determine whether the gain comes from learned region selection, constraint-based reconstruction, or their combination. We therefore perform a paired 2×2 component ablation over the same 380 incumbent schedules: 242 JSSP and 138 classical FJSP instances. Every instance uses the same parent schedule, reference value, and evaluation configuration across all four variants.

The two factors are the destroy decision and the reconstruction backend.

Heuristic selection. The heuristic variant uses the same candidate-generation pipeline, schedule analysis, failure-aware filtering, repair budget, job-tail closure, acceptance rule, and improvement horizon as the learned variant. The learned Candidate Head and Operation-Selection Transformer are replaced by fixed, non-learned priority rules for selecting a candidate region and determining the operations to release. The resulting operation set is then processed by the same closure and reconstruction procedure. Thus, this baseline preserves the search space and repair protocol while replacing the learned where/what decisions with deterministic heuristic selection.

MWKR reconstruction. The heuristic reconstruction backend receives the same effective repair region as CP. Operations outside this region keep their incumbent machine assignments and start/end times, while released operations are rescheduled one at a time. At each step, feasible operation-machine choices are formed from the ready operations and a Giffler-Thompson-style conflict set is constructed around the earliest-finish pivot. MWKR selects the feasible choice with the largest remaining job workload, with ties resolved by earliest start time, processing time, and job identifier. For FJSP, the machine is selected jointly from the operation's eligible machines. The resulting schedule must contain every operation and is materialized through the same schedule interface as CP. The same acceptance rule is then applied: improving and schedule-changing equal-makespan solutions may become the next incumbent, while true no-ops and worsening solutions are rejected. All four variants use the same 20-round improvement horizon.

This gives four matched variants: learned selection + CP, heuristic selection + CP, learned selection + MWKR, and heuristic selection + MWKR. The first variant corresponds to the learned-selection/constraint-repair path evaluated in this controlled experiment.

For every variant, ΔGap is measured relative to the same parent schedule, $\Delta\mathrm{Gap} = \mathrm{Gap}_{\mathrm{parent}} - \mathrm{Gap}_{\mathrm{final}}$, so positive values denote improvement. The common mean parent gap over the 380 schedules is 22.40%.

Table 35: Controlled 2×2 ablation over the same 380 parent schedules. Mean, median, and P10-P90 are instance-level final reference-gap statistics; family-macro gap gives equal weight to the 14 benchmark families. ΔGap is the mean reduction from the common parent gap. Improvement percentages and W/T/L are measured relative to the common parent schedule.

Selection	Repair	Mean gap	Median gap	P10-P90	Family-macro gap	ΔGap	Improved % (W/T/L)
Learned	CP	**13.52**	**11.72**	**1.19-25.00**	**12.07**	**8.89**	**91.84 (349/31/0)**
Heuristic	CP	20.34	18.39	5.40-40.40	18.97	2.07	65.00 (247/133/0)
Learned	MWKR	20.96	19.29	5.81-41.72	19.51	1.44	46.84 (178/202/0)
Heuristic	MWKR	20.96	18.75	5.81-41.77	19.68	1.45	38.68 (147/233/0)

The interaction between the two components is substantial. Holding CP fixed, replacing learned selection with heuristic selection increases the mean final gap by 6.82 percentage points . Holding learned selection fixed, replacing CP with MWKR reconstruction increases it by 7.44 points. In contrast, CP improves over MWKR by only 0.62 points when both use heuristic selection, while learned and heuristic selection have essentially the same mean gap under MWKR reconstruction. The paired difference-of-differences between the two factors is 6.83 points, with a 95% bootstrap confidence interval of $[6.08, 7.58]$. This indicates that the two components are complementary: learned selection is most useful when the selected region is subsequently reconstructed by CP, and CP benefits substantially from receiving regions chosen by the learned policy.

Table 36 shows that this conclusion is not driven by only one problem class. The effect is especially pronounced on JSSP, where learned selection + CP reduces the mean gap by 9.81 points from the common parent and improves 228 of 242 schedules. On classical FJSP, the same variant reduces the mean gap by 7.27 points and improves 121 of 138 schedules.

Table 36: Controlled component ablation separated by problem class. Each cell reports Final Gap / ΔGap

Selection	Repair	JSSP ($N = 242$)	FJSP ($N = 138$)
Learned	CP	**13.47 / 9.81**	**13.59 / 7.27**
Heuristic	CP	22.40 / 0.88	16.72 / 4.14
Learned	MWKR	22.78 / 0.50	17.77 / 3.09
Heuristic	MWKR	22.86 / 0.42	17.61 / 3.25

Because all four variants are evaluated on the same parent schedules, we can also compare their final gaps directly at the instance level. A Friedman test over the four matched variants rejects equality of their distributions ($\chi^2_F = 636.64$, $p < 0.001$). We then compare learned selection + CP with each alternative using two-sided Wilcoxon signed-rank tests with Pratt handling of ties and Holm correction, following the statistical protocol of Appendix L.7.

Table 37: Paired comparisons against learned selection + CP. Mean difference is $\mathrm{Gap}_{\mathrm{alternative}} - \mathrm{Gap}_{\mathrm{Learned+CP}}$, so positive values favor learned selection + CP. W/T/L denotes lower/equal/higher final gap for learned selection + CP on the paired instances. Bootstrap intervals use 20,000 paired resamples.

Alternative	Mean diff. [95% CI]	W/T/L	r_{rb}
Heuristic + CP	$6.82\,[6.10, 7.55]$	278/94/8	0.967
Learned + MWKR	$7.44\,[6.69, 8.20]$	313/54/13	0.950
Heuristic + MWKR	$7.44\,[6.68, 8.19]$	320/50/10	0.960

All three Wilcoxon comparisons remain significant after Holm correction ($p_{\mathrm{Holm}} < 0.001$). The effect sizes are also large, with rank-biserial correlations between 0.950 and 0.967 in favor of learned selection + CP. These results provide a direct controlled complement to the broader Construct-to-final analysis: the advantage of Destroy-to-Improve is not reproduced by replacing either its learned region-selection mechanism or its CP reconstruction backend with the corresponding heuristic alternative.

Table 38: Runtime of the controlled component ablation. All variants are executed by the same evaluation runner on the same paired instances.

Variant	Mean time (s)	Median time (s)
Learned + CP	16.96	19.16
Heuristic + CP	4.73	3.25
Learned + MWKR	63.56	28.12
Heuristic + MWKR	53.35	24.79

The matched runtime comparison provides additional context for the quality results. Heuristic selection + CP is the fastest variant, but its mean gap reduction is only 2.07 points compared with 8.89 points for learned selection + CP. More importantly, both MWKR-repair variants are slower on average than learned selection + CP while producing substantially smaller improvements. The quality advantage of learned selection + CP is therefore not explained by a larger heuristic-repair runtime budget. We nevertheless treat runtime as a secondary diagnostic here, because the four variants execute different reconstruction procedures rather than identical computations.

No variant worsens the parent schedule in Table 35. As in the other repair experiments, this should not be interpreted as an independent robustness result because worsening proposals are rejected by the common acceptance rule.

N.8 Pattern and Repair Memory Ablation

We additionally isolate the contribution of Pattern Memory and Repair-Aware Memory while keeping the remaining KARMS pipeline unchanged. The four matched variants enable or disable the two memory sources independently on the same $N = 55$ evaluation instances. We reuse the fixed 55-instance component-ablation subset defined in Appendix N.5.

Table 39: Ablation of Pattern Memory (PM) and Repair-Aware Memory (RM). Lower mean gap is better. All variants use the same evaluation set and differ only in the enabled memory components.

Variant	PM	RM	Mean Gap (%)	Δ vs Full
Full KARMS	✓	✓	**3.98**	-
No Repair-Aware Memory	✓	-	4.23	+0.25
No Pattern Memory	-	✓	4.45	+0.47
Both memories removed	-	-	6.39	+2.41

Removing Repair-Aware Memory increases the mean gap from 3.98% to 4.23%, while removing Pattern Memory increases it to 4.45%. Removing both memories produces a substantially larger degradation to 6.39%, 2.41 percentage points above the full configuration. Removing either memory increases the mean gap in this evaluation, while removing both produces the largest degradation, suggesting complementary information from the two memory sources. Pattern Memory supports construction decisions from similar historical states, whereas Repair-Aware Memory provides complementary evidence about previous repair outcomes.

N.9 FFSP Component Transfer

FFSP provides a useful stage-wise transfer analysis because the same inference components can be examined sequentially on a different shop structure without FFSP-specific gradient training or fine-tuning. The benchmark definition and evaluation protocol are given in Appendix L.3. Table 40 follows the schedule through a simple MWKR initializer, the learned Construct policy, bounded CP lookahead, and the final complete-schedule improvement stage. Each row averages 1,000 fixed FFSP instances.

Construct already improves over the MWKR baseline. Starting from the KARMS Construct solution, bounded CP lookahead and Destroy-to-Improve provide additional reductions at all three FFSP scales. At 100 jobs, for example, mean makespan decreases from 116.02 under MWKR to

Table 40: Progressive component analysis on FFSP. Values are mean makespans over 1,000 instances per size.

Jobs	MWKR	Construct	+ CP lookahead	Final KARMS
20	37.42	32.80	31.15	**28.85**
50	67.87	65.06	63.88	**56.67**
100	116.02	110.12	108.57	**100.51**

110.12 after learned construction, 108.57 after bounded CP lookahead, and 100.51 after complete-schedule improvement. The same ordering is observed at 20 and 50 jobs. This progression shows that learned construction, short-horizon constraint reasoning, and complete-schedule repair remain complementary after transfer to the supported FFSP setting.

For completeness, Table 41 compares the final frozen KARMS pipeline with FFSP-specific baselines on the same MatNet FFSP test sets. MatNet and MatNet×128 follow Kwon et al. (2021), while ReSched-samp follows the corresponding FFSP evaluation of Xiao et al. (2026). Dedicated FFSP methods obtain lower mean makespans at all three scales, so this comparison is interpreted as cross-problem transfer rather than FFSP-specific superiority. Nevertheless, KARMS remains competitive on individual instances despite receiving no FFSP-specific gradient updates.

Table 41: FFSP transfer over 1,000 instances per scale. Bold marks the best mean makespan; KARMS superscripts give instance-level wins against MatNet / MatNet×128 / ReSched-samp, respectively.

Method	FFSP20	FFSP50	FFSP100
MatNet	27.30	51.50	91.50
MatNet ×128	25.40	**49.60**	**89.70**
ReSched-samp	**25.12**	49.65	90.80
KARMS	$28.85^{246/135/129}$	$56.67^{123/87/84}$	$100.5^{46/12/12}$

Although the mean gap to the dedicated FFSP methods increases with problem size, the instance-level wins show that the transferred policy is not uniformly dominated: KARMS improves on MatNet in 246, 123, and 46 instances at 20, 50, and 100 jobs, respectively, with additional wins over MatNet×128 and ReSched-samp. Together with Table 40, this indicates that the transferred pipeline remains operational and benefits from each of its learned and constraint-based stages, even though FFSP was absent from gradient training.

N.10 Repair of ReSched and DANIEL Schedules

We use 135 common FJSP instances: 15 Brandimarte instances and 40 instances from each of Hurink edata, rdata, and vdata. A source-specific adapter converts each external output into operation start times, machine assignments, and induced machine order. Before improvement, we validate operation coverage, processing durations, precedence, machine eligibility, machine non-overlap, and reconstructed makespan. The Destroy Policy receives no parameters, logits, embeddings, or training data from the external initializer. All 675 checkpoint-instance evaluations passed validation. Table 42 reports paired results. Worsening candidates are rejected by the acceptance rule, so the absence of losses is structural and is not presented as an independent safety result.

The improvement is not attributable to a selected ReSched seed: its checkpoint-level mean gap reductions range from 2.00 to 2.38 points, with a between-seed standard deviation of 0.19 points. Mean improvement is also positive for every checkpoint-family combination. The two DANIEL models are checkpoint/training-distribution variants rather than independent random seeds.

N.11 Construct-Only Comparison with External Initializers

To isolate initial-schedule quality from Destroy-to-Improve, we compare KARMS Construct with external DANIEL and ReSched schedules on the same 135 FJSP instances used in the external-repair experiment: 15 Brandimarte instances and 40 instances from each Hurink *edata*, *rdata*, and

Table 42: External FJSP schedules before and after frozen Destroy-to-Improve. Each checkpoint is evaluated on the same 135 instances. W/T/L denotes improved/unchanged/worsened. ΔGap is the paired mean reduction in BKS gap; "Macro" gives equal weight to each of the four benchmark families. The aggregate ReSched CI clusters repeated seeds by instance.

Initializer	Gap: initial $\rightarrow$ final	W/T/L	ΔGap [95% CI]	Macro ΔGap
ReSched seed 1	$10.27 \rightarrow 8.27$	86/49/0	$2.00\,[1.50, 2.53]$	2.05
ReSched seed 2	$9.85 \rightarrow 7.46$	84/51/0	$2.38\,[1.87, 2.95]$	2.39
ReSched seed 3	$9.85 \rightarrow 7.66$	79/56/0	$2.19\,[1.70, 2.72]$	2.20
ReSched aggregate	$9.99 \rightarrow 7.80$	249/156/0	$2.19\,[1.82, 2.58]$	2.21
DANIEL SD1	$10.82 \rightarrow 8.30$	94/41/0	$2.53\,[1.93, 3.18]$	2.51
DANIEL SD2-mix	$10.67 \rightarrow 7.22$	102/33/0	$3.46\,[2.84, 4.11]$	3.26

vdata subset. The comparison includes raw Construct, Construct with bounded CP lookahead, three ReSched checkpoints, and two DANIEL variants; no Destroy-to-Improve repair is applied.

Table 43: Initial-schedule quality on the common 135-instance FJSP set. Values are mean BKS gap (%) before any Destroy-to-Improve repair. Lower is better. KARMS aggregate values are instance-count-weighted averages over Brandimarte and Hurink *edata*, *rdata*, and *vdata*.

Method	Mean Gap (%)
KARMS Construct	14.96
KARMS Construct + CP lookahead	11.53
DANIEL SD1	10.82
DANIEL SD2-mix	10.67
ReSched seed 1	10.27
ReSched seed 2	9.85
ReSched seed 3	9.85

The methods differ in role and training regime. KARMS uses one knowledge-augmented Construct policy trained on a mixed JSSP/FJSP curriculum spanning multiple problem sizes rather than a separate constructor specialized for this FJSP benchmark collection. Raw Construct nevertheless remains reasonably close to the external initializers, while bounded CP lookahead reduces its mean gap from 14.96% to 11.53%, a 3.43 percentage-point reduction before complete-schedule improvement.

This is consistent with KARMS's intended use of knowledge augmentation: structural scheduling relations, heuristic signals, and memory context provide information that is not tied to a fixed number of jobs or machines, allowing the same Construct policy to operate across JSSP and FJSP and beyond its reported training range. The Construct progression in Appendix N.3 further shows that the mature knowledge-augmented representation remains effective on substantially larger unseen sizes, while bounded CP lookahead provides additional decision-level refinement.

Although the external initializers achieve somewhat lower gaps on this FJSP set, the comparison is not a controlled ranking of training efficiency because architectures, training distributions, and training budgets differ. Rather, it shows that a unified KARMS constructor can produce schedules close to dedicated FJSP initializers while supporting a broader range of problem classes and sizes. Construct is therefore used as a general-purpose transferable initializer: knowledge supplies reusable scheduling structure, bounded CP lookahead refines difficult local decisions, and Destroy-to-Improve subsequently exploits remaining schedule-specific improvement opportunities.

N.12 Ablation Summary

The component analyses support a staged interpretation of KARMS. Historical Construct experiments show that feasible insertion removes an important restriction of the original append-only policy, the structured curriculum produces the largest subsequent reduction in Construct makespan, the mature knowledge representation provides a smaller additional refinement, and bounded CP lookahead acts as an inference-time correction whose benefit varies with problem size.

After construction, the complete post-construction pipeline produces a further substantial reduction across the 500 established JSSP/FJSP evaluations. Candidate-ranking diagnostics identify two distinct failure modes: an improving region may exist but be ranked poorly, or the candidate bank may contain no improving region. This motivates separating candidate generation from candidate ranking rather than treating the Destroy Policy as a single decision. A separate budgeted operation-selection ablation isolates the subsequent *what* decision. At the deployed repair budget, Learned Top-B improves both gap and repair time relative to Random-B and full-region release; the tighter budget provides an additional stress test in which the advantage over random subset selection becomes even larger.

Transfer results provide complementary evidence. On FFSP, learned construction, bounded lookahead, and complete-schedule improvement each reduce mean makespan without FFSP-specific gradient training. The frozen improvement model also improves schedules generated independently by ReSched and DANIEL, showing that its usefulness is not restricted to schedules produced by the KARMS Construct policy.

The controlled heuristic-replacement experiment provides the clearest direct component test. Across 380 paired JSSP/FJSP incumbents, learned selection + CP reduces the common mean parent gap from 22.40% to 13.52%, compared with 20.34% for heuristic selection + CP and approximately 20.96% for either MWKR reconstruction variant. The factorial comparison also shows a strong interaction: learned selection provides a large benefit with CP, while its mean advantage largely disappears when CP is replaced by MWKR reconstruction; conversely, CP provides its largest gain when applied to regions selected by the learned policy. Together with the candidate-ranking diagnostic, this supports the architectural separation between learning *where and what* to modify and using constraint reasoning to determine *how* the selected region is reconstructed.