gemini

Decentralized and Robust Scheduling for Flexible Job Shops using Multi-Agent Reinforcement Learning: A Comprehensive Critique and Implementation Framework

1. Introduction: The Stochastic Imperative in Industry 4.0 Scheduling

The domain of manufacturing scheduling is currently navigating a profound transition, moving from deterministic, static optimization paradigms toward dynamic, cyber-physical responsiveness. At the heart of this shift lies the Flexible Job Shop Scheduling Problem (FJSSP), a combinatorial optimization challenge that arguably represents the most faithful mathematical abstraction of modern high-mix, low-volume production environments.[1] Unlike the classical Job Shop Scheduling Problem (JSSP), which enforces a rigid sequence of machines for each job, the FJSSP introduces routing flexibility—the capability to process an operation on any one of a set of compatible machines.[2] This flexibility, while theoretically increasing the potential for throughput optimization, exponentially expands the search space, rendering traditional exact methods like Mixed Integer Linear Programming (MILP) computationally intractable for real-time applications.[3]

Historically, the industry has relied on centralized solvers, typically employing meta-heuristics such as Genetic Algorithms (GA) or Simulated Annealing (SA), to generate predictive schedules.[2] These approaches operate under the assumption of a "frozen" environment. However, the shop floor is inherently stochastic. Machine breakdowns, tool degradation, stochastic processing times, and urgent order insertions define the operational reality.[5] In such environments, a static schedule computed by a central "brain" exhibits fragility; a single disruption can render the entire plan infeasible, necessitating computationally expensive rescheduling or sub-optimal "right-shifting" of tasks.[2] This phenomenon, known as "schedule nervousness," undermines the theoretical optimality of centralized solutions.

The research proposal under review, *"Decentralized and Robust Scheduling for Flexible Job Shops using Multi-Agent Reinforcement Learning (MARL),"* correctly identifies this fragility as a critical bottleneck. It proposes a shift toward a "Self-Healing" architecture governed by decentralized agents.[2] While the motivation is sound, the proposed mechanism—a Multi-Agent System (MAS) relying on the Contract Net Protocol (CNP)—represents a heuristic

approach that predates the modern deep learning revolution. This report argues that while CNP offers decentralized execution, it lacks the *predictive* and *generalization* capabilities of modern Deep Reinforcement Learning (DRL).

To achieve true robustness—defined not just as stability but as the minimization of performance degradation under uncertainty—research must leverage the structural priors of the manufacturing environment. Recent advances in Graph Neural Networks (GNNs) allow us to model the shop floor as a dynamic heterogeneous graph, capturing the complex topological dependencies between jobs and machines.[8] When coupled with Multi-Agent Reinforcement Learning algorithms like MAPPO (Multi-Agent Proximal Policy Optimization) or QMIX, these systems can learn policies that are not merely reactive but *proactive*, anticipating bottlenecks and preserving slack for critical operations.[1]

This document serves as a comprehensive research report and implementation roadmap. It critically evaluates the student's proposal, establishes a state-of-the-art (SOTA) theoretical framework based on Heterogeneous Graph MARL, and provides a rigorous technical plan for executing a proof of concept on constrained hardware (NVIDIA RTX 4070). The goal is to elevate the proposed research from a heuristic comparison to a contribution at the frontier of Industrial Artificial Intelligence.

2. Theoretical Foundations and Literature Review

2.1. The Flexible Job Shop Scheduling Problem (FJSSP)

Formally, the FJSSP involves a set of n jobs $\mathcal{J} = \{J_1, \dots, J_n\}$ and a set of m machines $\mathcal{M} = \{M_1, \dots, M_m\}$. Each job J_i consists of a sequence of operations $O_{i,1}, O_{i,2}, \dots, O_{i,k_i}$. The critical distinction of FJSSP is that each operation $O_{i,j}$ can be processed on a subset of compatible machines $\mathcal{M}_{i,j} \subseteq \mathcal{M}$, with a processing time $p_{i,j,k}$ dependent on the chosen machine M_k.[1]

The optimization objective is typically to minimize the Makespan $C_{max} = \max_i C_i$, where C_i is the completion time of job J_i. However, in the context of Industry 4.0, the objective function becomes multi-dimensional, incorporating:

- **Total Tardiness:** $\sum \max(0, C_i - D_i)$, where D_i is the due date.
- **Robustness:** The stability of the schedule performance metric Z (e.g., makespan) in the face of stochastic perturbations ξ. This is often formulated as minimizing the expected

value $\mathbb{E}[Z(\xi)]$ or a risk measure like CVaR.[4]

2.2. The Limitations of the Contract Net Protocol (CNP)

The student's proposal relies on the Contract Net Protocol (CNP) for decentralization. Introduced by Smith (1980), CNP is a negotiation-based interaction protocol where tasks are auctioned to agents.[2]

- **Mechanism:** A Manager (Job) broadcasts a Task Announcement. Contractors (Machines) evaluate their capability and submit Bids. The Manager evaluates bids and issues an Award.
- **Critique in Stochastic Environments:** While CNP effectively distributes decision-making, its standard implementation relies on *greedy heuristics* for bidding (e.g., "bid based on current queue length").[2] This myopic view fails to capture second-order effects; a machine might have a short queue but be prone to failure, or accepting a task might block a critical path for a higher-priority job arriving later. CNP agents typically do not *learn* from long-term consequences; they simply execute pre-programmed logic.[11]
- **Communication Overhead:** In dense scheduling environments, the message-passing overhead of CNP (Announcement $\rightarrow$ Bid $\rightarrow$ Award for every operation) scales poorly, potentially introducing latency that rivals centralized solvers.[11]

2.3. The Advent of Deep MARL and GNNs

Current SOTA research (2024-2025) has largely moved beyond heuristic MAS toward Deep MARL, specifically utilizing Graph Neural Networks (GNNs) for state representation.[1]

- **Heterogeneous Graph Representation:** The shop floor is modeled as a graph where nodes represent operations and machines, and edges represent precedence and compatibility constraints. This structure is size-agnostic, allowing models trained on small instances to generalize to larger shops.[4]
- **End-to-End Learning:** Unlike CNP, where bidding logic is hand-coded, MARL agents parameterized by Deep Neural Networks (DNNs) learn to map states to actions (bids/dispatching decisions) by maximizing a cumulative reward signal. This allows the emergence of sophisticated strategies, such as "withholding capacity" to accommodate high-priority rush orders.[1]
- **Coordination Mechanisms:** Algorithms like QMIX (Value Decomposition) and MAPPO (Centralized Critic) allow agents to act decentrally during execution while leveraging global information during training, resolving the non-stationarity problem inherent in independent learning.[1]

3. Critical Analysis of the Research Proposal

The uploaded proposal [2] outlines a comparison between a centralized Genetic Algorithm (GA) and a decentralized MAS-CNP. The following critique identifies gaps and opportunities for elevation.

3.1. Strengths

- **Digital Twin Concept:** The proposal correctly identifies the need for a simulation environment (Digital Twin) to test robustness. The use of SimPy for discrete-event simulation is an industry-standard choice for this purpose.[2]
- **Problem Formulation:** The focus on "Cost of Robustness" is highly relevant. Quantifying the trade-off between theoretical optimality (efficiency) and operational stability (resilience) is a key research question in Operations Research.[6]
- **Stochastic Modeling:** Incorporating Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) via Poisson distributions ensures the evaluation environment reflects reality, moving beyond academic "toy" problems.[2]

3.2. Weaknesses and Theoretical Gaps

- **Lack of Learning in "MARL":** The proposal is titled "MARL," but the methodology section describes a heuristic CNP where machines bid based on "current queue length and repair status".[2] There is no mention of a policy network, value function, or gradient update step. Without a learning mechanism, this is a heuristic MAS, not a Reinforcement Learning system. This discrepancy is fatal to the novelty of the research.
- **Outdated Baseline:** Comparing against a static GA is insufficient. Static algorithms are known to fail in dynamic environments. A rigorous baseline must be a *reactive* heuristic, such as dynamic dispatching rules (e.g., Shortest Processing Time - SPT, Earliest Due Date - EDD) or composite rules.[17] The MAS must prove it can outperform these standard reactive policies, not just a static scheduler that invalidates upon the first disruption.
- **Simplistic Bidding Logic:** The proposed bidding logic ($t_{start} + t_{proc}$) minimizes local completion time. It does not account for global bottlenecks, machine degradation probability, or future job arrivals. This local greediness often leads to suboptimal global makespans.[11]
- **Insufficient Robustness Metrics:** The proposal suggests "Stability Metric: variance in makespan".[2] While useful, variance treats positive deviations (finishing early) and negative deviations (finishing late) equally. In manufacturing, only lateness is penalized. Furthermore, variance does not capture the *catastrophic tail risk*.

3.3. The Novelty Gap

To align with 2025 standards, the research must transcend CNP. Recent literature demonstrates that GNN-based DRL agents significantly outperform both heuristics and traditional solvers in dynamic scheduling.[8] The integration of **Heterogeneous Graph Neural Networks (HGNN)** with **MAPPO** is the current frontier.[10] Implementing this architecture

would shift the project from a reproduction of 1990s MAS concepts to a cutting-edge AI research contribution.

4. Proposed SOTA Architecture: Heterogeneous Graph MARL

We propose a complete architectural overhaul to replace the heuristic CNP with a learning-based system. This section details the **Heterogeneous Graph Neural Network (HGNN) + MAPPO** framework.

4.1. Heterogeneous Graph State Representation

We define the state s_t at time t as a heterogeneous graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$.

4.1.1. Node Sets ($\mathcal{V}_t$)

The graph contains two distinct types of nodes, reflecting the duality of the shop floor:

1. **Operation Nodes (O_{ij}):** Each active operation is a node.
   - *Features:* $\mathbf{x}_{O_{ij}} =$.
   - *Status:* One-hot vector {Waiting, Processing, Completed}.
2. **Machine Nodes (M_k):** Each resource is a node.
   - *Features:* $\mathbf{x}_{M_k} =$.
   - *HealthState:* Categorical variable {Functional, Degraded, Broken}.

4.1.2. Edge Sets ($\mathcal{E}_t$)

Relationships are encoded as directed edges:

1. **Precedence Edges ($O_{ij} \rightarrow O_{i,j+1}$):** Enforce the technological sequence of a job.
2. **Compatibility Edges ($O_{ij} \leftrightarrow M_k$):** Connect operations to capable machines. The edge weight w_{ijk} represents the processing time of operation O_{ij} on machine M_k.
3. **Machine Queue Edges ($O \rightarrow O'$):** Dynamic edges representing the current sequence of jobs in a machine's buffer.

This representation is **isomorphic** to the scheduling problem. When a machine breaks, its node feature $\mathbf{x}_{M_k}$ updates to "Broken." Through message passing, this information

propagates to all connected Operation nodes (O_{ij}), informing them that this machine is effectively "infinite cost," forcing the policy to learn alternative routing automatically.[1]

4.2. Heterogeneous Graph Neural Network (HGNN) Encoder

To process this graph, we employ a relational GNN (e.g., Heterogeneous Graph Transformer or GATv2). Standard GCNs are insufficient because they assume a single node/edge type.

For a node v of type $\tau(v)$, the update rule at layer l aggregates messages from neighbors $\mathcal{N}(v)$ grouped by relation type r:

$$\mathbf{h}_v^{(l+1)} = \text{GRU}\left(\mathbf{h}_v^{(l)}, \sum_{r \in \mathcal{R}} \sum_{u \in \mathcal{N}_r(v)} \alpha_{vu}^{(l)} \mathbf{W}_r \mathbf{h}_u^{(l)}\right)$$

Where α_{vu} is the attention coefficient, allowing the model to weigh the importance of different neighbors (e.g., focusing more on a "Broken" machine node than a "Functional" one further down the line).[1] This encoder compresses the entire shop floor state into a set of node embeddings $\mathbf{H}_v$.

4.3. Decentralized Policy via MAPPO (CTDE)

We adopt the **Multi-Agent Proximal Policy Optimization (MAPPO)** framework, which adheres to Centralized Training, Decentralized Execution.[1]

- **Agents:** We propose a **Dual-Agent** setup:
  1. **Job Agents:** Observe their own operation embeddings. Action: *Select Operation* to release.
  2. **Machine Agents:** Observe their own machine embeddings. Action: *Dispatch Job* from queue.
- **Decentralized Actors:** Each agent a has a policy network $\pi_{\theta_a}(u_a|o_a)$ that takes its local observation o_a (derived from the GNN embedding of its specific node) and outputs an action.
- **Centralized Critic:** A value network $V_\phi(s)$ takes the *global* state s (a pooling of all node embeddings) and predicts the expected return. This critic is used *only* during training to reduce variance and guide the actors.

Why MAPPO over QMIX? QMIX enforces a monotonicity constraint on the joint value function, which limits its ability to represent complex coordination strategies where an agent

might need to sacrifice local utility for global gain (e.g., a machine staying idle to wait for a high-priority job).[18] MAPPO typically outperforms QMIX in cooperative scheduling tasks with continuous/stochastic dynamics.[10]

4.4. Reward Shaping for Robustness

To align the agents with the goal of robustness, we design a dense reward function:

$$R_t = R_{\text{makespan}} + R_{\text{utilization}} + R_{\text{robustness}}$$

- $R_{\text{makespan}} = -(C_{max}(t) - C_{max}(t-1))$: Penalize increase in estimated completion time.
- $R_{\text{robustness}} = -\lambda \sum_k \mathbb{I}(M_k \text{ is Degraded}) \cdot \text{Load}_k$: Penalize assigning load to degrading machines. This term explicitly teaches agents to avoid risky resources *before* they fail, a behavior termed "predictive maintenance scheduling".[19]

5. Robustness Framework: Beyond Variance

To provide the "faultless metrics" requested, we must move beyond simple descriptive statistics. We introduce formal risk measures from financial engineering and reliability theory applied to scheduling.

5.1. Surrogate Robustness Measures (SRM)

Simulation is expensive. SRMs allow us to estimate robustness during the learning process without running thousands of Monte Carlo rollouts for every step. We focus on **Resilience-based SRM (SRM_R)**[20]:

$$SRM_R = \sum_{j \in \mathcal{J}} \sum_{i \in O_j} \frac{\text{Slack}_{ij}}{\mathbb{E}[p_{ij}]} \cdot \omega_i$$

Here, Slack_{ij} represents the allowable delay for operation i before it affects the makespan (Total Float). Dividing by expected processing time $\mathbb{E}[p_{ij}]$ normalizes this buffer relative to task duration. ω_i weighs the operation by the reliability of the assigned machine. Maximizing SRM_R encourages the scheduler to place "buffers" of time around unreliable machines, creating a structural immunity to breakdowns.

5.2. Conditional Value at Risk (CVaR)

Variance σ^2 treats positive and negative deviations symmetrically. In manufacturing, finishing early is fine; finishing late is costly. **CVaR** focuses on the tail risk.[22]

$$\text{CVaR}_\alpha = \mathbb{E}$$

We propose measuring $\text{CVaR}_{0.95}$, which quantifies the expected makespan in the worst 5% of breakdown scenarios. A robust scheduler minimizes this value, ensuring that even catastrophic failure cascades do not cause total system collapse.

5.3. Stability Metric (Nervousness)

Stability measures the deviation between the *predictive* schedule (generated at $t = 0$) and the *realized* schedule.

$$\text{Stability} = \sum_{i \in \mathcal{J}} |C_i^{\text{realized}} - C_i^{\text{predicted}}| + w \cdot \text{Reallocations}$$

This metric penalizes frequent changes in machine assignment ("Reallocations"), which incur hidden costs in material transport and setup changes.

6. Implementation Roadmap and Hardware Optimization

Designing a DRL system for FJSSP on an NVIDIA RTX 4070 (8GB VRAM) requires careful resource management. Graph datasets can grow large, and carrying gradients for multi-agent policies is memory-intensive.

6.1. Environment: The Digital Twin (SimPy)

We build a custom OpenAI Gym (Gymnasium) interface wrapping a SimPy core.

- **Agent Integration:** The SimPy environment pauses at decision points (Machine becomes free, Job arrives). It yields an observation to the RL agent, waits for an action, and then resumes the discrete-event simulation.
- **Stochasticity:**
  - **Failures:** Modeled as a Poisson process. We use a generator failure_process(env, machine) that interrupts the machine resource.
  - **Degradation:** A Hidden Markov Model (HMM) runs alongside each machine. State transitions (Good $\rightarrow$ Degraded $\rightarrow$ Fail) are probabilistic. The agent observes

"Degraded" signals (e.g., vibration data proxy) but not the exact time of failure.[23]

6.2. PyTorch Geometric (PyG) Implementation

We leverage PyG for efficient graph operations.

- **Data Structure:** HeteroData object stores node features x_dict and edge indices edge_index_dict.
- **Batching:** Crucial for 8GB VRAM. We cannot train on a single massive graph if the number of operations $N > 5000$.
  - **Strategy:** Use NeighborLoader or HGTLoader to sample subgraphs for training. This allows training on graphs larger than GPU memory by only loading the necessary k-hop neighborhood for the gradients being computed.[24]

6.3. RTX 4070 Optimization Strategy

The RTX 4070 is powerful (Tensor Cores) but VRAM-limited (8GB). We implement specific optimizations [18]:

1. **Automatic Mixed Precision (AMP):** Use torch.cuda.amp to run the HGNN forward pass and gradient computation in FP16 (half-precision). This effectively doubles the available VRAM and utilizes the 4070's Tensor Cores for 2-3x speedup.
2. **Gradient Checkpointing:** For deeper GNNs (e.g., >3 layers), we use torch.utils.checkpoint. This discards intermediate activations during the forward pass and recomputes them during the backward pass. It trades compute (cheap on 4070) for memory (expensive).
3. **Pinned Memory:** Use pin_memory=True in the PyG DataLoaders to accelerate transfer from host RAM to GPU VRAM.
4. **Flash Attention:** If using Transformer-based GNNs (HGT), enable Flash Attention (available in PyTorch 2.0+) to reduce the $O(N^2)$ memory complexity of attention mechanisms to linear $O(N)$.

6.4. Training Loop (Plan-v1)

- **Phase 1 (Pre-training):** Train the HGNN encoder using Imitation Learning (Behavior Cloning) on optimal solutions generated by a CP-SAT solver (e.g., OR-Tools) on small instances. This "warm starts" the policy, preventing the random exploration phase from taking weeks.[27]
- **Phase 2 (MAPPO Fine-tuning):** Switch to RL. Train on the SimPy environment with stochastic breakdowns enabled. Use Curriculum Learning: start with 0% failure rate, gradually increase to 20%.[1]

7. Experimental Design and Validation

7.1. Benchmarking Datasets

To ensure reproducibility and rigorous comparison, we utilize standard benchmarks, augmented with stochastic parameters:

- **Brandimarte (Mk01-Mk10):** Standard medium-scale instances.
- **Taillard (ta01-ta80):** Large-scale instances to test scalability.
- **Stochastic Extensions:** We augment these datasets by assigning exponential failure rates (λ) to machines based on the literature.[20]

7.2. Evaluation Protocol

We conduct a comparative analysis against three baselines:

1. **GA (Baseline from Proposal):** Centralized, re-optimizes only upon failure.
2. **Composite Dispatching Rules (PDRs):** A dynamic heuristic combining SPT (Shortest Processing Time) and WINQ (Work in Next Queue). This is the *true* industrial baseline for dynamic shops.
3. **HGNN-MAPPO (Proposed):** The deep learning approach.

Table 1: Proposed Metrics for Comparison

Metric Category	Metric Name	Definition	Goal
Efficiency	Normalized Makespan	C_{max}/C^*_{ma} (relative to best known lower bound)	Min
Stability	Nervousness	% of operations changing machine assignment	Min
Robustness	CVaR (95%)	Mean makespan of the worst 5% simulation runs	Min
Reliability	Successful Completion Rate	% of episodes where deadline	Max

		constraints are met	
Computation	Inference Time	Time to generate one action (ms)	Min (<100ms)

7.3. Comprehensive Ablation Studies

To isolate the contribution of each component, we perform the following ablations [28]:

1. **Graph Structure:** Replace HGNN with a standard MLP (flattened state). *Hypothesis:* Performance drops significantly on large instances, proving the necessity of topological information.
2. **Observation Radius:** Restrict GNN message passing to 1-hop vs 2-hop vs Full Graph. *Hypothesis:* 2-hop offers the best trade-off between local responsiveness and global awareness.
3. **Coordination Mechanism:** Compare MAPPO (CTDE) vs IPPO (Independent PPO). *Hypothesis:* MAPPO converges faster due to the centralized critic stabilizing the non-stationary environment.
4. **Robustness Reward:** Compare training with $R_{makespan}$ only vs. $R_{makespan} + R_{robustness}$. *Hypothesis:* The latter yields higher makespan in ideal conditions but significantly lower CVaR in disrupted conditions (The "Cost of Robustness").

8. Conclusion

The transition from static, centralized scheduling to decentralized, intelligent execution is a prerequisite for the realization of Industry 4.0. While the student's proposal correctly identifies the fragility of genetic algorithms and the need for self-healing systems, its reliance on the Contract Net Protocol limits its potential to simple heuristic negotiation.

By adopting the **Heterogeneous Graph MARL** framework detailed in this report, the research can leapfrog intermediate technologies. The proposed architecture—leveraging the structural priors of GNNs, the coordination power of MAPPO, and the rigorous risk quantification of CVaR/SRM—offers a robust, scalable, and computationally efficient solution. The implementation roadmap provided ensures that this advanced system is not just a theoretical construct but a deployable proof of concept, optimized for modern hardware constraints and validated against rigorous benchmarks. This approach does not merely solve the scheduling problem; it redefines the standard for robust autonomy in manufacturing.

References within the text refer to the following synthesized research clusters:

- [2]: User Proposal Document
- [1]: Graph Neural Networks for FJSSP & State-of-the-Art Reviews (2024-2025).
- [1]: Multi-Agent Reinforcement Learning (MAPPO, QMIX) in Scheduling.
- [5]: Robustness Metrics, Surrogate Measures (SRM), and CVaR.
- [3]: Contract Net Protocol (CNP) and its limitations.
- [18]: Hardware Optimization (PyTorch, Mixed Precision, GNN memory).
- [4]: Benchmark results (Taillard, Brandimarte).

Works cited

1. Multi-agent reinforcement learning for flexible shop scheduling problem: a survey - Frontiers, accessed February 10, 2026, https://www.frontiersin.org/journals/industrial-engineering/articles/10.3389/fieng.2025.1611512/full
2. plan-v1.md
3. Intelligent Scheduling in Open-Pit Mining: A Multi-Agent System with Reinforcement Learning - MDPI, accessed February 10, 2026, https://www.mdpi.com/2075-1702/13/5/350
4. learning to schedule job-shop problems: representation and policy learning using graph - arXiv, accessed February 10, 2026, https://arxiv.org/pdf/2106.01086
5. Resilience-Based Surrogate Robustness Measure and Optimization Method for Robust Job-Shop Scheduling - MDPI, accessed February 10, 2026, https://www.mdpi.com/2227-7390/10/21/4048
6. Robustness Measures and Robust Scheduling for Job Shops - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/publication/245315073_Robustness_Measures_and_Robust_Scheduling_for_Job_Shops
7. Evaluation of the Robustness for Integrated Production Scheduling and Maintenance Planning Problem - MDPI, accessed February 10, 2026, https://www.mdpi.com/2076-3417/13/4/2260
8. Multi-Agent Reinforcement Learning for Job Shop Scheduling in Dynamic Environments, accessed February 10, 2026, https://www.mdpi.com/2071-1050/16/8/3234
9. Applying Multi-agent Reinforcement Learning and Graph Neural Networks to Flexible Job Shop Scheduling Problem - Seoul National University, accessed February 10, 2026, https://snu.elsevierpure.com/en/publications/applying-multi-agent-reinforcement-learning-and-graph-neural-netw
10. Multi-Agent Reinforcement Learning for Extended Flexible Job Shop Scheduling - MDPI, accessed February 10, 2026, https://www.mdpi.com/2075-1702/12/1/8
11. Task Assignment of the Improved Contract Net Protocol under a Multi-Agent System - MDPI, accessed February 10, 2026, https://www.mdpi.com/1999-4893/12/4/70
12. (PDF) Task Assignment of the Improved Contract Net Protocol under a

Multi-Agent System, accessed February 10, 2026, https://www.researchgate.net/publication/332148538_Task_Assignment_of_the_Improved_Contract_Net_Protocol_under_a_Multi-Agent_System
13. Flexible job-shop scheduling via graph neural network and deep reinforcement learning - Institutional Knowledge (InK) @ SMU, accessed February 10, 2026, https://ink.library.smu.edu.sg/context/sis_research/article/9200/viewcontent/2022_TII_songwen.pdf
14. Job Shop Scheduling Benchmark: Environments and Instances for Learning and Non-learning Methods - arXiv, accessed February 10, 2026, https://arxiv.org/pdf/2308.12794
15. The available workers of the machines | Download Scientific Diagram - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/figure/The-available-workers-of-the-machines_tbl3_338163240
16. Robust scheduling for multi-objective flexible job-shop problems with random machine breakdowns | Request PDF - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/publication/257356195_Robust_scheduling_for_multi-objective_flexible_job-shop_problems_with_random_machine_breakdowns
17. Graph Neural Networks for Job Shop Scheduling Problems: A Survey - arXiv, accessed February 10, 2026, https://arxiv.org/html/2406.14096v1
18. Optimizing Memory Usage in PyTorch Models - MachineLearningMastery.com, accessed February 10, 2026, https://machinelearningmastery.com/optimizing-memory-usage-pytorch-models/
19. (PDF) Deep Reinforcement Learning-Based Job Shop Scheduling of Smart Manufacturing, accessed February 10, 2026, https://www.researchgate.net/publication/367634944_Deep_Reinforcement_Learning-Based_Job_Shop_Scheduling_of_Smart_Manufacturing
20. Flow chart of the ABC algorithm. | Download Scientific Diagram - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/figure/Flow-chart-of-the-ABC-algorithm_fig2_228756416
21. Optimization of Schedule Stability and Efficiency Under Processing Time Variability and Random Machine Breakdowns in a Job Shop Environment | Request PDF - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/publication/264371319_Optimization_of_Schedule_Stability_and_Efficiency_Under_Processing_Time_Variability_and_Random_Machine_Breakdowns_in_a_Job_Shop_Environment
22. Optimizing the service level of regular criteria for the stochastic flexible job-shop scheduling problem - ROADEF 2024, accessed February 10, 2026, https://roadef2024.sciencesconf.org/510348/document
23. Fault-Resilient Manufacturing Scheduling with Deep Learning and Constraint Solvers, accessed February 10, 2026, https://www.mdpi.com/2076-3417/15/4/1771
24. [2511.09261] A Distributed Training Architecture For Combinatorial Optimization -

arXiv, accessed February 10, 2026, https://arxiv.org/abs/2511.09261
25. Optimizing Memory and Retrieval for Graph Neural Networks with WholeGraph, Part 1, accessed February 10, 2026, https://developer.nvidia.com/blog/optimizing-memory-and-retrieval-for-graph-neural-networks-with-wholegraph-part-1/
26. How to Improve the Efficiency of Your PyTorch Training Loop | Towards Data Science, accessed February 10, 2026, https://towardsdatascience.com/improve-efficiency-of-your-pytorch-training-loop/
27. Graph Neural Networks for Job Shop Scheduling Problems: A Survey - arXiv, accessed February 10, 2026, https://arxiv.org/pdf/2406.14096
28. Attention-Based Multi-Agent RL for Multi-Machine Tending Using Mobile Robots - MDPI, accessed February 10, 2026, https://www.mdpi.com/2673-2688/6/10/252
29. [2207.04117] Ablation Study of How Run Time Assurance Impacts the Training and Performance of Reinforcement Learning Agents - arXiv, accessed February 10, 2026, https://arxiv.org/abs/2207.04117
30. Dual Operation Aggregation Graph Neural Networks for Solving Flexible Job-Shop Scheduling Problem with Reinforcement Learning | Request PDF - ResearchGate, accessed February 10, 2026, https://www.researchgate.net/publication/391480285_Dual_Operation_Aggregation_Graph_Neural_Networks_for_Solving_Flexible_Job-Shop_Scheduling_Problem_with_Reinforcement_Learning
31. REDUCE MEMORY LOADED INTO GPU WHEN TRAINING MODEL : r/pytorch - Reddit, accessed February 10, 2026, https://www.reddit.com/r/pytorch/comments/o8b51t/reduce_memory_loaded_into_gpu_when_training_model/
32. arXiv:2504.16106v1 [cs.DS] 17 Apr 2025, accessed February 10, 2026, https://arxiv.org/pdf/2504.16106

gpt

Critique and Improved Research Plan for Decentralized MARL-Based FJSSP Scheduling

Introduction

Flexible Job Shop Scheduling Problems (FJSSP) involve assigning a sequence of operations for each job to available machines, with the flexibility that an operation can be processed by more than one machine type. The user's current research plan targets a **decentralized, robust scheduling approach** for FJSSP using Multi-Agent Reinforcement Learning (MARL). In this approach, machines and jobs are modeled as agents negotiating task assignments via a **Contract Net Protocol (CNP)** style bidding system. The plan aims to handle **dynamic, stochastic disruptions** (e.g. machine failures) by simulating the shop floor in Python (SimPy), and to benchmark performance against a centralized Genetic Algorithm (GA) baseline. The end goal is a proof-of-concept that is rigorously tested with solid metrics, ablation studies, and comprehensive evaluation.

This report provides a detailed critique of the current plan and recommends improvements. We identify weaknesses and gaps, highlight strengths to build upon, and suggest modern state-of-the-art methods (2023–2026) that can enhance the novelty and effectiveness of the approach. We also propose architectural and algorithmic improvements, simulation and implementation strategies, and a robust evaluation framework. The suggestions emphasize **computational tractability** given the hardware constraint (single NVIDIA RTX 4070, 8 GB VRAM) and aim to ensure the research outcomes are both academically rigorous and practically relevant.

Critique of the Current Plan

Weaknesses and Gaps in the Current Approach

The current plan has promising elements but also exhibits several weaknesses that could hinder achieving a truly **robust and novel solution**:

- **Reliance on Contract Net Protocol (CNP) without Learning:** Using a classical negotiation scheme (CNP) provides a structured decentralization, but by itself it may not fully exploit learning. CNP is an older heuristic approach; without integration of learning it can be suboptimal and slow in complex, dynamic scenarios. If agents simply follow fixed bidding rules, the system might not adapt or improve over time. Modern AI schedulers have surpassed basic CNP in performance by learning from data. **Gap:** It's unclear how MARL is combined with CNP – if the plan is to have agents learn bidding strategies or just use CNP for task allocation. This needs clarification and likely a more learning-driven coordination mechanism.

- **Unclear MARL Algorithm and Coordination Mechanism:** The plan does not specify which MARL algorithm or training paradigm will be used. MARL in scheduling is challenging due to **non-stationarity** (agents concurrently affecting the environment) and credit assignment issues. A naïve independent learning approach (each agent learns in isolation) often fails because agents treat others as part of a moving environment. **Gap:** There's no mention of using modern MARL techniques like centralized training (CTDE) or value decomposition to handle these issues. Without this, learning may be unstable or converge to poor policies.

- **State and Observation Representation Not Detailed:** The plan doesn't describe how the scheduling state will be represented for each agent. In FJSSP, state spaces are large (machine statuses, job queues, operation sequences, etc.). A poor choice of state features or observation could limit the MARL's effectiveness. Many prior works used very high-dimensional handcrafted features or graphs. **Gap:** Failing to incorporate structured representations (e.g. graph-based state) could be a weakness, as recent research shows that carefully designed state representations (even minimalistic ones with proper structure) greatly improve performance. The plan should articulate how each machine (or job) agent perceives the environment (local observation vs. global information) and ensure important context (like upcoming operations, machine workloads, failure status) is available to the agents.

- **Action Space and Decision Scope Not Specified:** Similarly, the plan doesn't clarify each agent's action space. Does a **machine agent** decide which job to process next among those bidding for it, or does a **job agent** choose a machine? Or is there a two-stage negotiation (job announces, machines bid)? The CNP suggests a manager-contractor structure, but it's not stated who plays manager. This ambiguity could hide complexity: if each machine agent independently selects a job to process, conflicts must be resolved (multiple machines might choose the same job). Conversely, if job agents choose machines, machine capacity conflicts need handling. **Gap:** Without a clear definition, designing the MARL reward and ensuring feasibility (no two machines doing the same job operation, etc.) is difficult. The plan should define the agent roles more concretely and consider a mechanism (learning-based or rule-based) for conflict resolution – for example, a priority rule when two machines want the same task.

- **Limited Baseline and Benchmarking Scope:** While a GA baseline is good, relying solely on GA might be insufficient. GA is a *static, centralized* optimizer; it may not adapt well to real-time changes (you would need to re-run GA whenever a failure or new job occurs, which is slow). Furthermore, GA performance can depend on tuning and may not represent modern heuristics. **Gap:** The plan doesn't mention other baselines like **dispatching rules** (e.g. shortest processing time, earliest due date), **simulation-based heuristics**, or even a *centralized RL* approach. In absence of multiple baselines, it's hard to judge if MARL+CNP is truly superior or just beating an ill-suited GA in a dynamic setting. A wider benchmarking (including rule-based scheduling and possibly an optimal solution on small instances) is needed for rigorous evaluation.

- **Insufficient Emphasis on Training Robustness and Ablation:** The user aims for *faultless metrics and comprehensive testing*, but the plan doesn't detail **how** the training

and evaluation will ensure that. For instance, MARL can be stochastic – results vary by random seed. There's no mention of running multiple training runs or statistical significance testing. Also, while ablation studies are mentioned, the plan doesn't outline which components would be ablated. **Gap:** Without planning these details, there's a risk of missing critical analyses (like what if the negotiation protocol is removed, or what if failures are not included during training, etc.). A robust plan would specify important ablations (e.g., "no communication" scenario, "no RL (only CNP)" scenario, different reward functions) to validate each design choice.

- **Potential Performance and Scalability Issues:** The combination of SimPy simulation with MARL could be slow. Each training episode might simulate many time-steps, especially if many jobs and long processing times are involved. If the plan doesn't account for this, training could be prohibitively long or limited to trivial scenarios. Similarly, CNP involves messaging between agents; if there are many agents (machines/jobs), negotiation overhead or communication complexity might scale poorly. **Gap:** No discussion is given on how many machines/jobs the approach should handle, or how to ensure simulation speed (e.g., by simplifying time scales or parallelizing). Without addressing this, the proof-of-concept might remain at toy-scale. The 8 GB GPU can handle moderately sized neural networks, but the CPU might become the bottleneck due to simulation. The plan should include strategies for efficient simulation (e.g., event skipping, parallel episodes) and possibly use function approximation to avoid brute-force bidding by enumerating all options every time.

- **Handling of Stochastic Failures Not Elaborated:** Introducing machine failures (stochastic breakdowns) is a key novelty for robustness, but the plan is vague on *how* this is implemented and tackled. Will the RL agents get any warning or information about a failure (like a machine health state or just a sudden machine-unavailable event)? How will jobs in progress on a failed machine be handled (restarted or delayed)? **Gap:** If these details are not ironed out, the simulation may become inconsistent or the RL agents may not learn effectively (they could treat failures as random chaos unless the MDP is designed to include them in the state). Robust scheduling research often models machine health or failure probabilities explicitly in the state/reward. The plan should incorporate such elements (e.g., a feature for machine's failure status or time to repair) to help agents anticipate or react to failures. Additionally, without careful reward design, agents might "learn" to ignore failures (if, say, the reward only emphasizes throughput, they might not plan for breakdowns).

In summary, **the current plan's gaps** include an undefined learning algorithm/architecture, lack of clarity in agent design and negotiation integration, minimal baseline coverage, and missing details on state, action, and failure modeling. These need to be addressed to avoid a situation where the MARL either does not outperform simple heuristics or the experiments lack credibility.

Strong Points Worth Keeping and Expanding

Despite the above gaps, the plan has several **strong aspects** that should be retained and built upon:

- **Decentralized Multi-Agent Perspective:** Treating machines (and/or jobs) as autonomous agents is well-aligned with real-world smart manufacturing systems. Decentralization increases robustness by avoiding a single point of failure and enables scalability by distributing decision-making. The use of MARL fits the trend of leveraging AI for distributed scheduling in Industry 4.0. Notably, prior work has shown that a distributed learning-based scheduler can outperform centralized approaches and handle disruptions better. This is a core strength of the approach – **preserve the multi-agent design** as it can inherently adapt to local events (like a machine breakdown) more quickly than a centralized scheme.

- **Contract Net Protocol (CNP) as a Starting Coordination Mechanism:** CNP is a logical way to structure the interaction between agents (who needs a task done vs. who can do it). It ensures feasibility (one machine per operation) through negotiation and could prevent conflicts. The plan's idea to use CNP provides an initial scaffold for agent communication. Importantly, similar concepts (bidding, auctions, negotiations) have been combined with MARL in recent studies – for example, using a bidding mechanism to resolve conflicts among machine agents proved effective in a dynamic scheduling scenario. **Keep this negotiation aspect**, but consider learning enhancements (discussed later) rather than a purely fixed protocol.

- **Use of SimPy and Stochastic Simulation:** Implementing the environment in a discrete-event simulation (DES) framework like SimPy is a strong choice for realism. It allows modeling time progression, machine operation processing times, queues, and random breakdown events precisely. Many RL environments for scheduling are either simplified (turn-based) or custom-coded; using SimPy can simulate true parallelism of machines and time delays accurately. This will make the experiments more **credible** and results more applicable to real manufacturing settings. Also, introducing **stochastic failures and repairs** in the simulation is an excellent idea to test robustness – it moves the research beyond static benchmark puzzles to a realistic dynamic environment. Ensure to **expand on this** by varying failure rates and patterns in experiments to fully test the system's robustness.

- **Comparison with a Metaheuristic (GA) Baseline:** Including a GA baseline shows the intent to benchmark against established methods from operations research. GAs and other metaheuristics (ACO, PSO, etc.) have a rich history in FJSSP optimization. A well-tuned GA can be quite competitive for static scheduling. If the MARL approach can outperform GA, that's a significant result. The plan to compare with GA is a **strong point to keep**, but it should be expanded to a broader benchmarking suite (as noted earlier). It's also good that the baseline is *centralized*, providing a contrast to the decentralized MARL; this can highlight the benefits of the proposed method in dynamic conditions (GA might struggle with frequent rescheduling, whereas MARL agents can continuously adjust).

- **Focus on Rigorous Evaluation (Metrics, Ablation, Testing):** The user emphasizes *faultless metrics*, ablation studies, and comprehensive testing, which is an excellent research philosophy. It indicates an understanding that thorough experimentation is needed for academic work. This mindset should be maintained. In the improved plan, we will detail specific metrics, ablations, and tests – but it's worth noting the strength that the

user is prepared to go beyond just showing one or two favorable graphs. This thoroughness, when properly executed, will make the research outcomes much more convincing.

- **Hardware Feasibility Awareness:** A subtle but important point is the user's awareness of hardware constraints (8 GB VRAM). This suggests they will be cautious about model sizes and computation. It's a strength because it will guide the design towards efficient algorithms suitable for a single GPU, likely avoiding overly large models or brute-force methods that wouldn't run in practice. By acknowledging this, the plan can focus on *computationally tractable* methods (e.g., use smaller neural networks, limit parallelism) which we will also consider in the improvements.

In summary, **the strong points to retain** are the decentralized MARL approach with agent negotiation, the realistic SimPy-based simulation with failures, the inclusion of a GA baseline, and the intent for rigorous evaluation. These create a solid foundation. The following sections will build on these strengths while addressing the earlier gaps, by incorporating **state-of-the-art techniques and architectural improvements** to ensure the project's success and novelty.

Modern State-of-the-Art Methods and Architectures (2023–2026)

The field of intelligent scheduling and MARL has advanced rapidly. Incorporating recent **state-of-the-art (SOTA)** methods will greatly improve the research plan's quality, novelty, and chances of success. Below, we outline key approaches from 2023–2026 that are relevant, along with how they could be applied or adapted to this project:

- **Graph Neural Network (GNN) Encodings for Scheduling:** Recent works model the job shop state as a graph to capture complex relationships. For FJSSP, a **heterogeneous graph** can represent machines and operations as different node types, with edges for feasible assignments or precedence constraints. GNNs (graph convolution or attention networks) then compute embeddings for these nodes, which are used by the agents' decision policies. This approach excels because it can handle variable numbers of jobs/machines and explicitly model which operations are pending and which machines can do them. For example, **MAC-Sched (2024)** represented the shop floor as a heterogeneous graph and achieved superior performance over 18 heuristic rule combinations. In another study, a **type-aware MARL** used a heterogeneous graph with a meta-path RNN for machine states and graph attention for operation states. *Implication:* Integrating a GNN into the MARL agents can significantly enhance their situational awareness and decision quality, which is especially useful under dynamic conditions. It's a modern technique to strongly consider.

- **Transformer-based Neural Architectures:** Transformers have made inroads into scheduling due to their ability to handle sets/sequences flexibly. A very recent example is **ReSched (ICLR 2026)**, which uses a Transformer with dot-product attention for FJSSP. By simplifying state representation to a few essential features (and encoding job operation relations via a small graph), ReSched's transformer-based policy outperformed

classical dispatching rules and prior DRL methods. Transformers can capture global interactions (e.g., how scheduling one job might affect another) through attention mechanisms, and they naturally handle variable input lengths (number of jobs/machines) up to a limit. *Implication:* Using a **Transformer encoder** to process the state (or the list of available operations) for decision-making could improve the learning efficiency and outcome. This is a 2026-level idea that could be novel in a multi-agent context (most existing ones are single-agent). It must be balanced with complexity (a transformer might require more memory, but a shallow one can fit in 8 GB VRAM). Given the success reported, this is a promising direction.

- **Centralized Training, Decentralized Execution (CTDE) Paradigm:** Modern MARL often adopts CTDE to handle coordination without sacrificing decentralization at run-time. Under CTDE, during training the agents share information or have a central critic, but during execution each agent uses its own policy with local observations. Algorithms such as **QMIX** (value decomposition network) and its successors (QTRAN, QPLEX, Qatten) decompose a global team reward into individual value functions that each agent can learn. These methods have been very successful in cooperative multi-agent tasks by solving the credit assignment problem. In scheduling, where the goal (e.g., minimize makespan) is a shared team objective, such value-decomposition methods could greatly help agents learn *jointly optimal* decisions rather than greedy local ones. Similarly, actor-critic methods like **MADDPG** (Multi-Agent DDPG) or **COMA** (Counterfactual Multi-Agent policy gradient) use a centralized critic that can access global state during learning to compute better gradients for each agent. *Implication:* Incorporating CTDE will likely improve training stability and performance. For example, a centralized critic could observe the entire shop status (all machine queues, all jobs) to judge the quality of the joint action at each step, guiding each agent's policy updates. This addresses the non-stationarity issue by stabilizing learning. Adopting frameworks like QMIX or MADDPG (with caution on continuous actions) is in line with recent MARL literature.

- **Hierarchical and Heterogeneous Agent Frameworks:** Another state-of-the-art idea is to use **hierarchical MARL**, which means structuring the decision process into multiple layers. In scheduling, a natural hierarchy is: (1) choose which job (or operation) to schedule next, and (2) assign it to a machine. In fact, one 2025 study created a **"job selection – machine assignment"** multi-agent framework with two types of agents (one type picks jobs, another assigns machines). They reported that this approach with tailored state and action spaces for each agent type led to improved solution quality and even handled bi-objective optimization (profit and makespan) effectively. Hierarchy can also come in as a high-level scheduler vs. low-level executors. For instance, a high-level agent might allocate a batch of tasks or reallocate jobs upon failures, while lower-level agents handle immediate scheduling. *Implication:* Introducing hierarchy can reduce complexity each agent faces and create a novel angle. The plan could implement two layers: e.g., a **dispatcher agent** (or a job-agent coalition) that decides which job should go next into the system (perhaps using a learned heuristic or rule), and the **machine agents** that then bid or decide who processes that job. This separation of concerns can make learning easier (each sub-problem is simpler) and also mirror how human schedulers often operate (first deciding an order of jobs, then allocating machines). It's a modern concept that adds novelty, especially if combined with MARL (most prior works

are either single-agent or non-learning hierarchical rules).

- **Learned Communication and Negotiation:** While the contract net protocol is a fixed communication pattern, state-of-the-art MARL research explores **learnable communication** – allowing agents to send messages to each other and learn what to communicate. Frameworks like *CommNet, RIAL/DIAL, or recent Graph Attention Communication* could enable machines to, say, broadcast their availability or request help when overloaded. In scheduling context, explicit messaging could be used for agents to warn others of a breakdown or to cooperate by exchanging tasks. For example, one could train agents with a communication channel where a machine agent can ask if others can take over its job when it expects to go down. This is advanced and not widely applied to FJSSP yet, but aligns with cutting-edge MARL. *Implication:* If time and complexity allow, exploring a **communication-enhanced MARL** (where the contract net emerges as a learned behavior rather than a fixed protocol) would be highly novel. Agents could use an attention mechanism to attend to other agents' states (a form of implicit negotiation). However, given this is complex and 8 GB VRAM is a limit, this should be considered only after establishing a working basic MARL; it's mentioned here for completeness of SOTA methods.

- **Robust and Distributional RL Techniques:** Robust scheduling can benefit from RL techniques that handle uncertainty better. **Distributional RL** (learning the distribution of returns, not just the mean) can be useful when outcomes (makespan, delays) have high variance due to randomness. There has been exploration of distributional RL in scheduling (e.g., to predict the whole distribution of job completion times). Also, **risk-sensitive or robust RL** approaches (like using percentile of reward instead of expectation, or adding penalty for variance) could be applied so that the learned policy is not just good on average, but avoids catastrophic failures. Some 2026 works mention using independent learners with distributional value functions for FJSSP. *Implication:* The plan could incorporate robustness at the algorithmic level: e.g., use **Double DQN (DDQN)** or **Dueling DQN** (for stability) as these are known to perform well in scheduling contexts, or use **policy gradient with entropy regularization** to encourage exploration (ensuring agents don't get stuck in brittle strategies). While not a separate architecture, making the RL training robust (through techniques like reward shaping, domain randomization, etc.) is part of modern best practices.

- **Multi-Objective Optimization and Metrics:** Modern scheduling research is often multi-objective – for example, balancing efficiency with energy or due date adherence. The user's goal is robustness (which can be seen as another objective alongside productivity). State-of-art methods sometimes use **reward shaping or lexicographic optimization** to handle this. One approach (Wang *et al.* 2025) combined a game-theoretic reward (Nash bargaining) to unify multiple objectives in a MARL training. In our context, robustness (e.g., minimal disruption from failures) and performance (makespan/tardiness) could be two objectives. *Implication:* Consider using **multi-objective MARL** concepts: either train a single policy with a weighted reward sum (tune weights via ablation), or even output a set of Pareto-optimal policies (though that's advanced). At minimum, ensure the reward function captures the different priorities (more in the reward design section below).

Incorporating these contemporary techniques will ensure the research is not reinventing old wheels and is competitive with the latest developments. Next, we integrate these into specific suggestions for architectural and algorithmic improvements to the user's plan.

Architectural Improvements to Enhance Novelty

To make the solution novel and effective, we recommend several architectural enhancements that build on the plan's base while leveraging the SOTA concepts above:

1. Hybrid Learning and Negotiation Mechanism

Improve the Contract Net Protocol with Learning: Instead of using CNP as a fixed rule-based negotiation, turn it into part of the learning environment. For example, *machines as agents* and *jobs as tasks to be allocated* can still interact via bidding, but the bidding behavior can be learned. One idea is to give each machine agent a policy that decides on a **bid value or priority for a job** (or decides which job to bid on). The highest bid "wins" the job as per CNP, but the strategy of bidding high or low is learned through RL to maximize long-term reward (e.g., bidding higher on jobs that are critical for overall performance). This retains the conflict-resolution nature of CNP but removes arbitrary heuristics – agents learn how to bid optimally.

Another approach is the **dispatcher-agent idea** mentioned earlier: implement a *two-agent type system* where one agent (or a centralized dispatcher module) announces available tasks and machine agents bid. The dispatcher could itself be an RL agent that decides which task to announce next (resolving tie situations or sequencing tasks intelligently). This way, the negotiation protocol is preserved but augmented with learned decision-making at each step.

If sticking closer to classical CNP, you might consider an **Adaptive or Ensemble CNP**, where multiple heuristics for bidding are available and the agent learns to choose among them. For instance, a machine agent could have options like "bid shortest processing time", "bid earliest due date", "bid random" etc., and the RL learns which strategy yields the best outcome in different states. This is analogous to what Gui *et al.* (2024) did by having agents learn weights for scheduling rules at decision points. They effectively *learned to select among heuristic rules* using MADDPG, within a CNP-based MAS, and showed improved performance over any single rule. Adopting a similar approach would enhance novelty (combining rule-based wisdom with RL adaptation) and likely yield a robust scheduler.

Why this is novel: Classic CNP hasn't typically been combined with deep RL until recently. By doing so, you contribute to the emerging area of *learning-enhanced negotiation*. It provides a bridge between traditional MAS scheduling and end-to-end learned policies. Moreover, this approach naturally handles dynamic changes: if a machine fails, it will stop bidding/announcing, and the others dynamically take over the tasks – a behavior which CNP already covers. The learning overlay will further ensure the allocation decisions leading up to and after such events are optimized from experience.

2. State Representation with Graphs or Simplified Features

Implement a Graph-Based State Encoder: Representing the FJSSP state as a graph and using a GNN to encode it can dramatically improve how much context each agent has. One design: for each decision time, construct a bipartite graph between *waiting operations* and *machines*. Connect an operation node to a machine node if that machine can process that operation (technologically capable and not currently busy or will be free soon). Also connect operation nodes sequentially if they belong to the same job (precedence constraints). A GNN can compute embeddings for machines and operations that capture not only direct connections but also the broader context (e.g., how many other machines can do this operation, how many operations remain in this job, etc.). The machine agent's policy network can then take its machine-node embedding (which now contains rich info about its options and state) as input to decide on actions. Likewise, if job/operation agents exist, their embeddings would inform their actions.

This approach is supported by recent research successes: a graph encoding was part of MAC-Sched and helped generalize to different factory configurations; a graph attention network in Lv *et al.* (2025) effectively guided schedule repair decisions under breakdowns. If a full GNN is too complex to implement from scratch, a simpler route is to use a **graph-based feature extraction** – e.g., count of capable machines per operation, workload on each machine, etc., which is essentially a summary of the bipartite graph. The key is to avoid a massive flat state vector and instead use structured representations that can scale and generalize.

Leverage Transformer if Possible: If implementing a GNN is cumbersome, an alternative is to use a **Transformer encoder** on a set of inputs representing the state. For example, create a list of "tokens" such as machine states (with features like machine id, current workload, status) and job operation states (features like job id, operation id, remaining processing time, etc.). Add positional or type embeddings to distinguish machine vs operation tokens, and feed this set into a Transformer. The Transformer's self-attention will learn relationships (which operations are vying for which machine, etc.) in a flexible way. The output embeddings can be assigned back to each entity, and an agent can pick its relevant output. ReSched's success with just four essential features per operation and a transformer suggests that even a simplified input, when fed through a powerful architecture, can outperform brute-force feature engineering.

Given the 8 GB VRAM constraint, you would use a *small transformer*: e.g., 2–4 attention heads, a few layers, and limit the token count by only including currently relevant operations (like those waiting in queue, not all operations of all jobs if some are not ready yet). This keeps computation reasonable. The novelty here is high – applying transformers in MARL for scheduling is cutting-edge 2025–2026 tech, and it could set your work apart. Just ensure to justify it by citing something like ReSched and articulate how it simplifies state representation (fewer handcrafted features, the model learns what's important).

3. Centralized Training and Critic Designs

Adopt CTDE with a Centralized Critic or Value Decomposition: As discussed, one strong improvement is to train the agents with a **centralized critic**. In practice, you could implement this by having a global critic network that takes the joint state (maybe the concatenation of all machine states, or a global graph embedding) and outputs a value (expected global reward) or

Q-value for the joint action. During training, agents would still execute their decentralized policies, but the critic's feedback (like TD-error or policy gradient advantage) considers the whole system, which helps align agent behaviors with the global objective (e.g., minimize overall makespan or total tardiness). There are good libraries and examples for MADDPG (which uses centralized critic) that you could adapt. If using value-based methods, you might implement **QMIX**: have each agent output Q-values for its actions; use a mixing network that combines these Q's into a global Q and train that to predict the team reward. The monotonicity constraint of QMIX (global Q is an increasing function of each local Q) often holds in scheduling when using team reward, but if not, more advanced QTRAN/QPLEX can be considered.

For example, suppose the reward is negative makespan (so higher reward is shorter schedule). This is a shared reward obtained at episode end. Independent Q-learners would struggle since no single agent directly sees how its early actions contributed to final makespan. But a value decomposition can distribute this reward to agents' Q-values in a way that they learn cooperative policies. By citing the MARL survey, you can justify this approach academically.

Parameter Sharing and Heterogeneous Critic: In FJSSP, all machines might follow similar decision logic (choose a task), so you can **share the policy network parameters among all machine agents** to reduce the number of learnable parameters (this is common in MARL to improve sample efficiency and symmetry). However, if you have different types of machines (say some are faster, some are specialized), you might include type-specific parameters or inputs (e.g., machine type ID as part of state). For the critic, since it's centralized, it can be a single network. If you implement job-selection agents and machine agents separately (heterogeneous agents), you can have one critic for all or separate critics per role. A recent work used a hyper-network to allow different parameters for different agent types while still training in a centralized way – a sophisticated method to ensure each type's value function is appropriately adjusted. Depending on complexity, you might not go as far as hyper-networks, but at least acknowledging heterogeneity (two agent classes) and possibly training them with their own critics or carefully designed shared critic will be valuable.

Overall, adopting CTDE will likely require more implementation effort (you have to manage the training loop manually or use an existing MARL framework), but it is *worth it*. It directly addresses a known weakness (coordination and credit assignment) and brings your work to modern standards where purely independent MARL is generally considered insufficient for cooperative tasks.

4. Reward Function and Objective Shaping

Design a Robust, Multi-Faceted Reward Structure: In scheduling, the reward mechanism is crucial and tricky – a sparse reward (like only giving a reward at episode end for makespan) makes learning slow. The plan should use **reward shaping** to guide agents at intermediate steps. For instance, every time an operation completes, you could give a small reward equal to (some baseline time – actual completion time) to encourage faster completions. Or penalize the time a machine stays idle while jobs are waiting (to encourage keeping machines busy). If robustness is a goal, include a penalty for missed deadlines or for each time a machine goes down with work incomplete (to encourage agents to maybe prefer machines that are healthy or to reassign tasks proactively if a machine is likely down – though predicting that might be

beyond what they can do without prognostics).

Given multiple objectives (throughput, meeting due dates, robustness to failures), a **composite reward** can be formulated. A weighted sum is simplest: e.g., `Reward = -α * makespan - β * total tardiness - γ * downtime impact`. One of the cited papers did exactly this by introducing a "cooperative cost" into the reward to blend objectives. You might not have a direct metric for robustness, but you can approximate it by penalizing very long delays caused by failures or the number of times jobs have to be rescheduled. Make sure to test different weight combinations (this can be part of ablation) to see which yields the best result. Alternatively, use an **adaptive or scenario-based reward:** for example, during training randomly switch the weightings so the policy doesn't overfit to one trade-off (though this is more advanced).

Also, consider **shaping by potential-based functions** (from reinforcement learning theory): For example, you can add a bonus for every operation finished (like +1) to expedite learning (the agent gets some reward for each completed task, not just final outcome). This won't change the optimal policy theoretically if done right, but will make learning faster by providing denser feedback.

Finally, ensure the reward is **global or appropriately assigned**. If each machine gets an individual reward (e.g., based on its utilization or the job it finished), they might optimize locally and not align with global optimum. A shared reward (team reward) fosters cooperation but can dilute feedback. A compromise could be: give a global reward every scheduling tick (like negative sum of waiting times of jobs, so everyone cares about overall delay) *plus* a small local reward for each action (like machine gets a reward for finishing a job quickly). This mix can help balance local vs global. Experiment with this structure – it's often necessary in MARL to get the right incentives.

Reference modern practices: As noted in one reference, a multi-agent framework used *adaptive state representations and reward functions* for different agent types to achieve dual objectives. This hints that you may even tailor the reward per agent type. For example, a job-selection agent might be rewarded on how well due dates are met (since it decides which job is prioritized) whereas a machine agent might be rewarded on utilization or processing efficiency. Both ultimately influence makespan and robustness, but the division of responsibilities can make learning more effective. This is a nuanced improvement that can set your work apart by demonstrating a deep understanding of reward design in MARL.

5. Hierarchy and Modularity for Novelty

If time and scope allow, implementing a **hierarchical controller** could be a standout feature. One possible architecture:

- A **High-Level Scheduler** (could be a learned policy or a heuristic module) that periodically evaluates the system and can take actions like re-routing jobs, triggering a rescheduling when a disruption occurs, or allocating preventive maintenance. This is somewhat analogous to a production supervisor. For example, when a machine breaks down, the high-level agent could decide which jobs to reassign to other machines or whether to wait (incorporating some downtime planning). This agent could be trained via RL as well (observing a summary of system state, deciding an action that affects the

environment, like moving all jobs from machine A to B if A broke). This is venturing into **meta-RL or hybrid optimization**, but it addresses robustness explicitly.

- The **Low-Level Scheduler** would be the existing MARL of machine agents that handle the fine-grained task assignments during normal operation. The high-level agent would only intervene on big events (new urgent job arrival, machine breakdown, etc.), effectively providing a new layer of decision that current plan doesn't have. This is similar to *hierarchical reinforcement learning (HRL)* where one agent's action sets goals or parameters for others. It's complex but could be simplified: for instance, the high-level could just choose which rule the system should follow in the next interval (like switching the objective from makespan to tardiness if many jobs are running late, etc.), which the lower agents then heed.

While full HRL might be ambitious, **even a simpler dual-agent-type system (job selector vs machine executor)**, as discussed, will add novelty. The referenced 2025 framework where one agent type handled job selection and another handled machine assignment saw improved robustness and even yielded better Pareto fronts for conflicting objectives. Emulating that structure would differentiate your work from single-layer MARL approaches.

Note: Keep the hierarchy design modular – if something fails, you can fall back to a single layer. For example, if you can't get a job-selection RL agent working well, you could default to a heuristic dispatcher and still use MARL at machine level (or vice versa). This flexibility ensures you can still demonstrate a working system and perhaps compare hierarchical vs non-hierarchical in ablation.

By implementing these architectural improvements – enhanced negotiation via learning, graph/transformer state encoding, CTDE training, sophisticated reward shaping, and possibly hierarchical decision layers – the research will stand on the frontier of 2020s scheduling optimization. Each of these adds novelty: e.g., *"MARL with learned bidding for FJSSP," "Graph-based representation for dynamic job shop," "hierarchical multi-agent scheduling"* – all of which are cutting-edge topics as evidenced by very recent publications.

Importantly, these must be balanced with **feasibility**. It's not necessary to implement every single suggestion; rather, pick a combination that you have resources to execute. For instance, one might combine graph state + CTDE + learned bidding in one coherent approach. Or transformer + independent MARL (if CTDE is too hard) + robust reward. The key is to incorporate enough SOTA elements to ensure novelty and strong performance, without exceeding the project's capacity.

Next, we focus on the **algorithmic and simulation** improvements to support these architectural choices.

Algorithmic and Simulation Enhancements

Simulation Environment Improvements

Your choice of **SimPy** for simulation provides a realistic temporal environment. To integrate this with RL: treat each scheduling decision as a **time-step** in the MDP. Concretely, you can advance SimPy's clock until a point where a decision is required – e.g., when a machine becomes free or a new job arrives (whichever comes first). At that point, pause the simulation and have the relevant agents observe the state and choose actions (which job to process, or which machine to request). Then apply those decisions (assign the job to the machine, start processing) and resume SimPy until the next decision point. This will create an episode consisting of a sequence of decision points, ending when all jobs are completed (or a certain time horizon is reached). This approach ensures the RL interacts with the sim only at key moments, not at every simulated time unit, which is efficient.

Handling Concurrent Decisions: If multiple machines free up at once, you could have them make decisions simultaneously (multi-agent step). This is where potential conflicts arise (two machines might pick the same job) – implementing the negotiation protocol or a conflict resolution rule is necessary. One way: when multiple machines are free, treat it as a *single joint action* selection problem – either by a bidding process or by ordering the decision (one machine gets priority to choose first based on a criterion, as MAC-Sched did by prioritizing machines with fewer options). This joint decision could be done sequentially in SimPy (one machine chooses, then the next sees the updated state and chooses, etc.), which breaks the simultaneity but simplifies conflict handling. The order can be randomized or fixed by priority to avoid bias.

Stochastic Failures in SimPy: Model each machine as a SimPy process that can go through "working" and "broken" states. For example, for each operation processed, you can schedule a potential breakdown event: if a random draw indicates a failure will occur during processing, preempt it at that time, mark the machine as broken, and require repair time. Use SimPy's event handling to interrupt the machine's process and start a repair timer. During this breakdown, any job that was on the machine might need to wait or be reassigned (this is tricky – perhaps simply delay it until repair is done, or you can create a decision point where the job is put back to the queue of remaining operations because the machine broke). Make sure to include these details in the environment logic.

The agents should have some observation of this event. At least, they should perceive that machine X is now unavailable. If you maintain a machine status in the state (e.g., a binary indicator for each machine "up/down"), then after a breakdown this indicator flips and agents can react (like avoiding assigning new tasks to it). If the breakdown happens while processing, you might penalize that outcome to represent lost work or delay.

Simulation Speed: SimPy can handle thousands of events per second easily, but RL might need many episodes. To speed things up:

- Keep the time horizon limited (maybe simulate, say, an 8-hour shift or until a set of jobs complete).

- Use vectorized simulations if possible (multiple SimPy envs in parallel processes). This might be doable with Python's `multiprocessing` or using a library like Ray. For example, running 8 parallel envs could use 8 CPU cores and keep the GPU busy with larger batch updates, making better use of the RTX 4070.

- If parallelism is hard, consider simplifying processing: e.g., scaling all times down so that events happen more frequently (makes episodes shorter in wall-clock time). Since absolute time doesn't matter, a job taking 5 hours vs 5 minutes in simulation is the same logically, but shorter durations mean the sim finishes faster (just ensure to scale failure rates accordingly).

Integration with RL Algorithms: You might leverage existing RL frameworks for parts of this. OpenAI Gym or the newer Gymnasium allows custom envs – you can wrap your SimPy simulation in a Gym interface (with `reset`, `step` functions). In the step function, you would advance the simulation to next decision, collect observations and reward. There are examples (as per search results) of Gym environments for job shop, and even one specifically using SimPy for scheduling. Reusing those or OR-Gym's FJSSP environment (if available) could save time. However, customizing is fine given your specific needs (failures, MARL). Also, consider using **PettingZoo** library for multi-agent environments if you want a standardized MARL API – it could manage multiple agents' actions per step nicely. But writing your own loop might give more flexibility.

Computational Considerations and Algorithm Selection

Given the 8 GB GPU and presumably a decent CPU, here are targeted suggestions to keep things tractable:

- **Neural Network Sizes:** Use shallow networks for agents. Often in scheduling, even a 2-layer MLP with 64–128 neurons can suffice, or a GNN with 128-d embeddings. The state space, if encoded well, doesn't need extremely high dimensional hidden layers like image processing would. This keeps memory and computation lower. Monitor GPU memory usage and adjust batch sizes accordingly. An RTX 4070 can easily handle tens of thousands of parameters and batch sizes of hundreds, so you have headroom – the main limitation is to avoid overly complex models like a huge transformer.

- **Algorithm Efficiency:** Off-policy algorithms (like DQN, DDQN) are sample-efficient and can reuse experiences (experience replay), which is good if simulation is slow. On-policy methods (like PPO) require more fresh samples but are stable. Considering results: one study found DDQN outperformed PPO in a similar multi-agent scheduling scenario. This suggests a well-tuned DQN variant is both efficient and effective. You could start with **DDQN with replay** for each agent (or a joint replay buffer for all), which uses less on-policy samples. If using CTDE, then a joint experience replay or centralized learner will be needed (which is fine).

  For continuous actions (if you go for something like bidding with a numeric value), you'd need DDPG/MADDPG. Those can be finicky but with proper normalization and not too large action space, they can work. However, many scheduling actions are inherently discrete (choose job A, B, or C), so you might not need continuous actions at all except perhaps for representing a bid value. A discrete approach might be simpler: e.g., each machine agent's action is an index of which job to take from the list of available ones (with an option to do nothing or idle if appropriate). If the action space is large (say 20 jobs waiting), use techniques like **action masking** (to disallow invalid choices) and

maybe **heuristic pruning** (only allow, say, top 5 candidate jobs based on some heuristic priority to reduce branching). This was suggested by an approach that did search-space reduction for JSSP with RL. It's a pragmatic way to keep the action space manageable for the neural net.

- **Hyperparameters:** Use the GPU for neural net training, but the simulation runs on CPU. So make sure to overlap these if possible (e.g., while the GPU is optimizing on a batch, the CPU can run the next episode in parallel threads). This can be done with async frameworks or simpler, by collecting a bunch of episodes (rollouts) first, then training, etc. A typical setup could be: run N episodes, collect transitions, then do M gradient updates, and repeat. This might be easier than a fully online single-step update approach in a DES environment.

- **Failure of an Episode:** In dynamic environments, sometimes things can go very wrong (e.g., all machines break down and jobs never finish). Decide on a clear **termination condition** and maybe a timeout (if an episode exceeds some time, end it and give a large negative reward to discourage that scenario). This prevents the agents from learning to exploit weird loopholes like stalling indefinitely to avoid negative reward.

- **Use of Existing Tools:** To implement MARL algorithms efficiently, consider using **Ray RLlib** – it has multi-agent support and can handle centralized critics. Stable Baselines3 is mostly for single-agent, but you can hack multi-agent by combining observations; however, RLlib or CleanRL or Mava (a MARL framework) might give more out-of-the-box support for MADDPG, QMIX, etc. Evaluate the learning curve vs writing it yourself. Given it's a research project, sometimes writing the training loop manually gives more insight and flexibility for customizations (like custom logging, complex reward calculation). But leveraging well-tested implementations can save time and avoid bugs, especially for things like replay buffers, parallel environment handling, etc.

Robustness and Testing During Training

To ensure the learned policy is **robust** to variations (like different failure scenarios), apply *domain randomization* during training: vary the seed for job arrival times, processing times, failure intervals every episode. This prevents the policy from overfitting to a particular sequence of events. Essentially, each episode should be a new random scenario drawn from a distribution (representing the space of possible shop floor conditions). The MARL will then learn a policy that works well on average across these, which typically generalizes better to unseen scenarios.

Monitor certain metrics as training goes, for example: average reward per episode, average makespan achieved, number of deadlines missed, etc., to see if they trend positively. Additionally, after training, test the policy on a set of **unseen scenarios** (with different random seeds or different job mixes) to evaluate generalization.

Another aspect: consider if partial observability is an issue. If each machine agent only knows its own status and maybe a local view of jobs, it might not know what other machines are doing – which is realistic (decentralized). This makes it a **Dec-POMDP** in theory. If the agents lack critical info (like a machine might not know if another machine is under maintenance unless

told), learning can suffer. Solutions include: allow some information sharing (e.g., a machine broadcast when it breaks, which could be simulated by having that info become visible to others), or include some global features in each agent's observation (like number of machines down at the moment, or total jobs remaining). Since it's a cooperative setting, providing a bit more info to agents (even if not fully global state) often helps coordination. This doesn't violate decentralization if it's info they realistically could get via a communication network on the shop floor (which in Industry 4.0 is plausible). So, to improve robustness, ensure agents aren't completely blind to important events outside their local sphere.

Finally, implement **checkpointing and testing**: regularly save model checkpoints and test them on a set of scenarios (without exploration noise) to see how performance is improving. This not only helps in picking the best model for final evaluation but is also critical in case training goes awry (you can revert to a stable checkpoint). It also lets you do early stopping if the policy converges or no longer improves.

Evaluation: Benchmarking, Testing, and Ablation

A comprehensive evaluation plan will strengthen the work significantly. Here's a structured approach:

Benchmarking with Multiple Baselines

To truly assess performance, compare the MARL approach against diverse methods:

- **Genetic Algorithm (GA) Baseline:** As planned, implement or use an existing GA for FJSSP. Ensure it's well-tuned (good population size, crossover/mutation rates) and perhaps run it multiple times for fairness (GAs are stochastic). Use it primarily on static scenarios (all jobs known upfront, no breakdown) to get a baseline schedule quality. In dynamic scenarios, you can simulate a *myopic GA scheduler* by periodically re-running GA when events occur (though this might be slow). If GA struggles with dynamic events, note that as a downside. The literature often shows GA yields good but not optimal results and can be slow for large problems, which your approach might overcome by continuous decision-making.

- **Dispatching Rule Heuristics:** These are fast, simple baselines that often perform decently in dynamic scheduling. Examples: *Shortest Processing Time (SPT)* first, *Earliest Due Date (EDD)* first, *Least Work Remaining*, *First Come First Serve*, etc., as well as combinations or more advanced rules (like ATC – Apparent Tardiness Cost rule). You can include one or two such rules (or even an ensemble where a rule is chosen depending on situation). They require almost no compute and can react immediately to changes (just pick next job according to rule). While they likely won't beat an optimized RL or GA on optimality, they are very *robust and fast*, so if your method can beat them consistently, that's a strong result. Moreover, rules often excel under certain conditions (e.g., SPT minimizes mean flow time well, EDD minimizes lateness), so it's illustrative to see where the MARL stands in comparison – perhaps it can combine the strengths of multiple rules adaptively.

- **Optimal Solver for Small Cases:** If possible, use an MILP or Constraint Programming solver on small static FJSSP instances (with, say, 5 jobs, 3 machines) to get the optimal makespan. This serves two purposes: (1) sanity check – if your approach is far from optimal on a trivial case, something's off; (2) gives a performance bound – you can report how close to optimal your approach gets. For dynamic cases, "optimal" is ill-defined, but at least for static baseline or simplified scenarios (no failures, all jobs known), it's nice to have. You might use the ILP models from literature (some references in that sciencedirect review [22] might have formulations). However, this is optional if time is short; heuristics and GA are usually sufficient to demonstrate effectiveness.

- **Centralized RL (Single-Agent) Baseline:** To isolate the benefit of decentralization, you could create a *single-agent RL* that observes the entire state (all machine and job info) and decides all assignments. This could be implemented as sequencing decisions: e.g., at each step, the single agent picks which machine should process which job next (this might need a custom action encoding, perhaps two actions: choose machine i and choose job j for it). Or more simply, a single agent could choose the next operation to start (implicitly assigning it to a specific machine as part of the action). This agent essentially mimics a dispatch rule but is learned. If you use a powerful architecture (like a transformer) for it, it might achieve good results. By comparing against this, you can see the *price (or benefit) of decentralization*. In some cases, centralized might perform a bit better since it has full information and can coordinate perfectly, but it's not realistic to implement on the shop floor. If your decentralized MARL is nearly as good or better under certain disturbances (like when comms delays or partial info make central control less effective), that's a noteworthy finding. Indeed, one 2021 study found that multiple AI schedulers (distributed) outperformed a centralized RL due to reduced communication and better handling of unexpected events. You might replicate that result, strengthening the case for MARL.

- **Prior Learning Approaches:** If any code or baseline is available from literature (e.g., the heuristic-guided Q-learning from Zhou *et al.* 2022, or the graph RL from 2023, etc.), and if it's feasible to implement or approximate, it could be insightful. However, this might not be necessary – often, comparing to well-known heuristics and a basic RL variant suffices. If a reviewer is aware of a particular prior work, they might ask "how does it compare?", so having at least references to those and a qualitative or quantitative comparison (even if just citing their results on standard benchmarks vs. yours) can be good. For instance, if ReSched reports a certain makespan on a standard dataset, and you test your approach on the same dataset, you could compare your numbers to theirs (with citation) to argue you are on par or better.

Metrics for Performance and Robustness

Ensure to measure and report **multiple metrics** to give a full picture:

- **Makespan:** The total time to complete all jobs (common objective in scheduling). This is critical if you claim optimization performance. For dynamic scenarios, makespan might be less meaningful if new jobs keep arriving, so use a cutoff time or a rolling horizon.

- **Mean Flow Time / Total Completion Time:** Sum of completion times of jobs, or average time a job spends in the system. This metric captures overall responsiveness, not just the last job done. It's important because an algorithm could minimize makespan by letting many jobs finish very late as long as one critical path is optimized; mean flow time penalizes holding any job up too long.

- **Tardiness and Lateness:** If you have due dates for jobs, calculate metrics like average tardiness (how late jobs are on average) or percentage of jobs that meet deadlines. Robust scheduling often cares about due-date performance.

- **Machine Utilization and Idle Time:** Especially relevant under failures – measure what fraction of time machines are productive vs. down or idle. A robust schedule might keep machines busy (except when broken) and quickly reassign work when there's an idle. Conversely, if an approach has machines waiting unnecessarily, that's inefficiency.

- **Throughput or Output Rate:** In a continuous job arrival scenario, how many jobs per hour are completed. This could be used if the environment is not a fixed set of jobs but a stream.

- **Recovery Metrics:** Since robustness is key, define some metrics around failure events:

    - **Mean Time to Recover (MTTR) from failure:** e.g., if a machine fails, how long on average until its work is taken over by others or it is back up and jobs complete. Compare this across methods – a good method might have lower MTTR due to quick rescheduling.

    - **Performance Degradation:** Run an experiment with no failures and record the metric (makespan, etc.), then run with failures (same jobs, plus some breakdowns) and see how much worse it gets (% increase in makespan, etc.). If your method is robust, the degradation percentage might be smaller than GA or others. This could be shown in a bar chart for clarity.

- **Stability of Schedule:** Possibly measure how much the schedule changes or how many times jobs are reassigned. While flexibility is good, too much change could be disruptive. Count, for example, the number of operations that started on one machine then had to move due to breakdown (lower is better). Or the number of *rescheduling actions* taken. A method that is overly reactive might constantly shuffle tasks; a robust method might make minimal necessary adjustments.

- **Computation Time:** If relevant, note the computational effort needed by each method (especially GA vs RL decision times). MARL once trained will make decisions in milliseconds, whereas GA might take seconds/minutes for large problems. This difference is part of the value proposition in a real-time setting. However, training time of MARL is huge (but that's offline), so you might mention it but it's not a run-time concern.

When presenting results, tables for numerical values and line/bar charts for trends are useful. For instance, you could have a table of results on static benchmark instances (with makespan,

etc., for each method), and separate graphs for dynamic cases (like makespan vs. time or job arrivals). If doing multiple trials (which you should for stochastic scenarios), include error bars or min/max ranges to show variability.

Given the academic nature, also perform **statistical tests** if you have enough samples – e.g., a t-test or Wilcoxon test comparing your MARL vs GA performance over 20 random scenarios, to show the improvement is statistically significant (p-value). This preempts reviewers' concerns about randomness. With multiple metrics, you might apply multivariate tests or just discuss each metric's significance.

Ablation Studies

Ablation experiments will validate the contributions of each component of your approach. Some key ablations to consider:

- **No Learning vs. Learning:** Run the system with the CNP negotiation but *without* the RL decisions (e.g., machines bid with a fixed heuristic like shortest processing time priority). Compare that with the full MARL system. This directly shows what RL adds. If the RL vastly outperforms static CNP, then you've demonstrated learning was beneficial. If not, maybe the negotiation is already good, but then highlight that RL brings adaptability to different conditions (maybe test on a scenario that the heuristic is not tuned for).

- **Independent vs. Centralized Training:** If you implemented a centralized critic (CTDE), ablate it by switching it off (train agents independently with only local rewards or with the same global reward but no critic). Likely, performance will drop or learning will be slower. This supports the use of CTDE and aligns with MARL theory that CTDE helps. Conversely, if by chance independent learning did fine, that's interesting too – maybe the problem structure helped. Either result is insightful.

- **State Representation Variants:** If you introduced a fancy state encoding (graph or transformer), test a simpler alternative. For example, ablate the graph by replacing it with a flat feature vector (like a sorted list of machine loads and job lengths). Show that the advanced representation yields better results or faster learning. This justifies the extra complexity. If using a transformer, you could ablate the attention by using a simple pooling of inputs or by limiting context, to show the transformer's benefit.

- **Reward Function Components:** As suggested, if your reward has multiple terms (throughput, robustness penalties, etc.), ablate each term. For example, train a policy without the failure-penalty term in the reward and see if it becomes less robust (maybe it optimizes throughput at the cost of being blindsided by failures). This will highlight that your reward shaping was essential to get the desired behavior. Alternatively, if you have a multi-objective scenario, you can try different weightings (not exactly ablation, but sensitivity analysis) to show the trade-offs and why your chosen weights are a good balance.

- **Communication/Negotiation Mechanism:** If you added a learned communication channel or a hierarchical dispatcher, ablate those. E.g., run the system without the dispatcher (each machine just picks greedily) to see the difference, or disable any

communication (if your agents share some info or embeddings) to see how performance changes. In Gui *et al.* (2024) they essentially compared a self-organizing MARL vs. classical methods to show the MARL's superiority – you can similarly compare *with vs. without* the MARL coordination.

- **Model Scale and GPU Usage:** Potentially, you might ablate model sizes (small network vs larger network) to demonstrate that you chose the smallest sufficient model to fit in 8 GB while still achieving good performance. If a much larger model doesn't improve much, that's a useful result (it means your approach is efficient). Or if it does improve, note the trade-off (maybe not feasible on given hardware). This is a more technical ablation but could be relevant if someone questions if your network capacity was enough.

For each ablation, measure the same metrics as before so you can quantifiably say e.g., *"Using graph embeddings improved average reward by X% and reduced variance by Y compared to a non-graph state"*. Present these perhaps in a separate table or a set of bar charts where each bar is a variant of your method. That visually drives home which components matter.

Realistic Stress Tests

For robustness claims, design some stress scenarios:

- **High Load Scenario:** e.g., 2× the usual number of jobs or very tight deadlines. See if MARL handles the pressure better than baselines (maybe by dynamically balancing load) or if it degrades gracefully.

- **High Failure Scenario:** e.g., machines have a mean-time-to-failure that is very low so that many breakdowns occur. Check if the schedule still runs efficiently or if certain methods fail to cope. Perhaps your approach will shine here by quickly reassigning tasks, whereas GA (if it had to be rerun frequently) might not keep up.

- **Edge Case Scenario:** e.g., a machine that is significantly faster than others (to see if agents learn to utilize it without all piling on and causing a bottleneck), or heterogeneous job sizes (mix of very short and very long tasks). This can reveal if the policy generalizes to imbalance.

- **Generalization Test:** Train on a certain number of machines/jobs, then test on a slightly different configuration (say one extra machine or a different mix of job types) if possible. A strong MARL approach, especially with graph/transformer, might generalize to a different size problem. Traditional methods or a fixed-size neural network might not. If you can demonstrate some generalization (even without retraining), that's a big plus and a hallmark of a good representation.

Throughout evaluation, **cite comparisons** to literature when available. For instance, if your results on a standard dataset are better than those reported by a 2024 paper, mention that (carefully, with citation). Or if not better, at least you can say *"comparable to X method within*

margin, but ours is decentralized/robust, offering other advantages." The goal is to situate your contribution relative to the state-of-art.

Finally, ensure the evaluation is clearly linked to your claims. If you claim "fault-tolerance", show evidence: e.g., *"Method A completed 95% of jobs on time despite 3 machine failures, whereas baseline B only 80%"*. If you claim "improves over GA", show the numeric improvement and maybe a schedule Gantt chart comparison illustrating qualitatively how the MARL makes smarter decisions (you can embed a Gantt chart image for a small instance to visually compare schedules – a powerful illustration if allowed).

By executing this robust evaluation plan, you'll demonstrate not only that your improved approach works, but *why* it works and under what conditions, which is the essence of solid academic research.

Revised Implementation Strategy

Bringing all the above together, we outline a revised implementation strategy step-by-step, emphasizing the integration of improvements while keeping in mind the hardware constraints:

Development Roadmap

1. **Environment Construction:** Start by building the SimPy-based FJSSP environment with essential features:
    - Represent machines as processes that can take tasks and possibly break down.
    - Represent jobs and operations, with operations waiting in queues until a machine processes them.
    - Implement random breakdown events for machines (with parameters for mean time between failures and repair time distributions).
    - Test the simulation standalone with a simple rule (like always take the first available job) to ensure it behaves realistically (jobs finish, failures cause delays, etc.). This will validate the environment before adding RL complexity.
2. **Observation and Action Design:** Decide on what each agent observes. Likely include:
    - For a machine agent: its own status (idle/busy, time until free, if broken, etc.), a summary of jobs it could process (like features of the next operation of each waiting job that can go to this machine), and perhaps limited global info (e.g., number of jobs waiting in the system or how many machines are down).
    - Normalize these features (0-1 range) to help neural networks. Possibly use a fixed-size observation (if graph/transformer not yet implemented) by limiting to a max number of jobs or using placeholder values if fewer.

- Define the action for a machine agent as an index representing “choose job i from the considered list, or choose to remain idle/pass”. Enforce feasibility: if an agent picks a job, remove it from others’ options for that decision cycle.

- If using a separate dispatcher agent, its observation would be global (like all machine workloads and job queue lengths) and action could be selecting which job to release next (or which machine gets priority).

3. **Baseline Integration Early:** Implement the GA and a couple of heuristic dispatchers in code. This allows you to generate some baseline schedules from the same environment. You can also use these to generate **expert data** if you consider pre-training the RL or for debugging (e.g., see how far the RL is from a decent heuristic in early training).

4. **Neural Network Model Selection:** Choose the model type for agents:

    - Initially, you might start with a simple feedforward network per agent (taking a flattened feature vector) to ensure the RL plumbing works. Once that is stable, upgrade to a more advanced representation (graph or transformer). This incremental approach helps isolate bugs – it’s easier to debug a basic DQN with a simple state than a complex GNN from the get-go.

    - For graph representation: consider using a library like PyTorch Geometric or DGL, which can simplify GNN implementation on GPU. You’d feed it the constructed graph of operations and machines and get embeddings.

    - For transformer: PyTorch’s nn.Transformer or the huggingface libraries can be used in a custom way. Ensure sequence lengths (number of tokens = machines + operations considered) are not too high to avoid memory blow-up. Perhaps cap at, say, 50 tokens (if more operations, maybe sample or prioritize some).

    - Sharing parameters: likely use one neural net shared by all machine agents (so you train one policy for all, benefiting from more data and symmetry). If job/dispatcher agent exists, that gets its own net.

5. **RL Algorithm and Training Loop:** Implement the training loop with the chosen algorithm:

    - For DQN/DDQN: you need experience replay. Use a buffer size sufficient for many experiences (maybe 100k or more transitions) but watch memory. Use target network updates for stability. Because environment steps are not at fixed intervals (they depend on event occurrences), you might want to store transitions in a time-step fashion (state, joint-action, reward, next state) where joint-action encompasses all agents’ actions at that decision step. You can train agents either jointly on these (if using global Q mixing) or separately (if independent, but that’s less ideal).

    - For Actor-Critic (like MADDPG or MAPPO): use policy and critic networks, possibly with parameter sharing for policies. Ensure the critic gets global state

input. Use frameworks if possible (e.g., Stable Baselines3 has single-agent PPO which you could hack into multi-agent if you combine observations, but multi-agent-specific frameworks are better).

- Include exploration strategy: e.g., ε-greedy for DQN or Gaussian noise for DDPG. In multi-agent, you might need all agents to explore, so consider decaying exploration over time and maybe use *parameter noise* or *centralized epsilon* (to avoid one agent always exploiting while others explore).

- Training might require a lot of episodes to converge, so be prepared to adjust learning rate, exploration decay, reward scaling, etc. Use the logging to tune these.

6. **Testing and Iteration:** After initial training runs, evaluate on some scenarios and compare to baselines. Likely you will need to iterate: if results aren't as good as expected, analyze where it fails:

    - If the agents aren't coordinating, maybe the reward needs tweaking or you truly need a centralized critic.

    - If the training is unstable, maybe reduce learning rate or use a different algorithm (e.g., PPO could be tried if DQN fails, since PPO is quite stable albeit less sample-efficient).

    - If failures confuse it, perhaps augment state with more info (like a countdown to repair for broken machines, so agents know if a machine will be back soon or not).

    - Use the ablation-style tests internally during development to see if each component is helping.

7. **Implement Advanced Features (if not yet):** Add the graph or transformer encoding once basic MARL works. Also implement the hierarchical layer if aiming for that. It might be wise to finalize a working flat MARL solution, then introduce the dispatcher agent on top and retrain or even train it separately (e.g., train machine-level policy first, then add a high-level agent that assumes machines will follow a certain policy).

8. **Final Training and Model Selection:** Train the final version on a range of conditions (variety of random seeds). Save the best performing model (maybe based on highest evaluation reward or shortest makespan on a validation set of scenarios). Because of stochasticity, you might train multiple models and pick a champion (if time permits).

9. **Evaluation Suite:** Run the saved model (no exploration noise) on the full set of test scenarios and benchmarks you've prepared:

    - Static benchmark instances (without failures) to compare makespan vs GA, etc.

    - Dynamic scenarios (with failures, new jobs) comparing with dispatch rules and

GA (if GA is run periodically).

- Edge cases and stress tests as discussed.
  Make sure to gather results over enough runs for statistical confidence.

10. **Ablation Experiments:** Modify the code to disable or change one component at a time and re-run a subset of evaluations. It might not be feasible to retrain everything from scratch for each ablation (that's time-consuming), so focus on key ones. For example, to ablate the graph, you could simply switch the state input to not use the graph and see how performance drops with the already trained model (though strictly speaking, the model should be retrained without graph to be fair). If possible, retrain smaller versions for ablation (maybe on a smaller problem instance) to demonstrate differences. This part can be done in parallel since ablations are independent experiments.

11. **Metrics Analysis:** Process the logs to compute all metrics. Use scripts to parse simulation outputs (SimPy might not directly give makespan, so you compute it as max completion time of all jobs). Compute averages, standard deviations. Plot graphs. Ensure citations to any external data (like if you put a line in a chart from a literature method, cite it).

12. **Documentation:** Throughout the process, maintain clear documentation of the assumptions (e.g., how breakdown is modeled, what each agent knows). This will be useful when writing the paper/report so you can clearly explain the environment and methodology. It will also help if you need to troubleshoot issues by referring back to these assumptions.

Handling Constraints

During implementation, always keep an eye on resource usage:

- Use `torch.cuda.memory_allocated()` and similar to see how much VRAM is used. If you hit close to 8 GB, consider optimizing (reducing batch size, smaller network, or using float16 precision if safe).

- CPU usage: SimPy is single-threaded per env. If using multiple envs, do multi-processing to leverage multi-core. But be mindful that Python multiprocessing duplicates memory – might need to keep the simulation lightweight.

- Training time: If each episode is a full simulation, it might take seconds. MARL might need thousands of episodes. This can be heavy. Use as high learning rate as you can without diverging to speed convergence, and consider any tricks to reduce variance (like reward normalization, if huge delays cause very large negative rewards, normalize them to a more stable scale).

- If time is a constraint in development, prioritize a subset of improvements that are most crucial: e.g., CTDE + good reward shaping might yield more impact than a transformer representation if you can't do both. It's okay to simplify some aspects as long as you

explain it – you can say, for instance, *“Due to computational limits, we used a simpler MLP instead of a full transformer, but this can be explored in future work.”* However, given the strong results reported by ReSched with a transformer, it would be great to include at least a basic attention mechanism if possible.

.