LIFTING UAV SCHEDULES: DIFFUSION POLICIES FOR ITERATIVE IMPROVEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Unmanned aerial vehicle (UAV) scheduling is a challenging sequential decision problem with tightly coupled operational constraints. Even small local edits can propagate through payload, energy, and timing dependencies, making high-quality schedules difficult to obtain through one-shot generation. We instead cast it as a sequential refinement process and propose **LIFT**, a refinement-aligned one-step diffusion policy for **L**earning schedule improvement via **I**mitation and PPO **F**ine-**T**uning. LIFT predicts structured edits on a schedule-dependent candidate graph, decodes them into feasible successor schedules, and optimizes them for long-horizon improvement. Training follows a refinement-aligned two-stage design: it first learns edit-level behavior from expert improvement trajectories, and then applies PPO fine-tuning through a one-step denoiser with tractable conditional action log-probabilities. Trained only on small-scale instances, LIFT achieves the lowest mean total arrival time across all evaluated benchmark configurations and generalizes to substantially larger problems without retraining. Together, these results reposition diffusion in combinatorial optimization: from a one-shot solution generator to a reinforcement learning policy for sequential solution refinement.

1 INTRODUCTION

Unmanned aerial vehicle (UAV) scheduling is becoming central to autonomous logistics, from last-mile delivery to emergency response and disaster relief (Betti Sorbelli, 2024; Dorling et al., 2016; Zhang et al., 2021), with real-world deployments accelerating demand for scalable optimization methods (UPS, 2020; Amazon, 2020; Wing, 2026; Zipline, 2026). We focus on *load-dependent UAV scheduling*, where a feasible schedule must jointly determine customer service order, depot assignment, vehicle usage, payload evolution, battery feasibility, and arrival time (Pasha et al., 2022; Torabbeigi et al., 2020). The key challenge lies in *load-dependent dynamics* (Dorling et al., 2016; Zhang et al., 2021): even a small local edit can cascade through the affected routes, reshaping objective value and feasibility far beyond the edited segment. As a result, high-quality schedules rarely emerge from isolated local changes; they are built through a *chain of interdependent refinements*.

Existing approaches only partially fit this sequential-refinement setting. Classical heuristics such as LKH-3 (Helsgaun, 2017), ALNS (Ropke & Pisinger, 2006), and VNS (Mladenović & Hansen, 1997) perform instance-wise search without learning reusable improvement behavior across problems. Learning-to-construct methods aim to produce a complete schedule in a single pass. For load-dependent UAV scheduling, this one-shot strategy becomes increasingly difficult as problem size grows, because the solver must identify a high-quality feasible schedule from a rapidly expanding solution space. Learn-to-improve methods are closer in spirit, as they iteratively improve the current schedule. However, many representative methods (Wu et al., 2021; Ma et al., 2023; Guo et al., 2025; Hottung & Tierney, 2020) define actions through prescribed operator templates, restricting structural flexibility: the policy selects or parameterizes a prescribed search rather than directly generating a structured edit tailored to the current schedule. Recent LLM-based solvers (Zhang et al., 2026; Liu et al., 2026; Ye et al., 2024; Liu et al., 2024) pursue greater flexibility through automated heuristic generation, but the resulting procedures are non-differentiable and prompt-sensitive.

Diffusion-based combinatorial optimization is attractive for iterative refinement because denoising resembles improvement: transforming an imperfect structure into a better one. Existing methods, however, mostly use diffusion at the *solution level*, generating or regenerating complete solutions (Sun

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

& Yang, 2023; Li et al., 2023; Vignac et al., 2023). For example, GenSCO (Li et al., 2025), a cutting-edge diffusion-based method and the closest baseline to our setting, regenerates perturbed complete solutions rather than choosing local edits inside the refinement loop. Iterative refinement instead requires diffusion to act as a stochastic policy over structured schedule edits. Each sampled edit is decoded into a feasible successor schedule and evaluated by its realized improvement. To optimize this policy, we use Proximal Policy Optimization (PPO) (Schulman et al., 2017), a popular algorithm for fine-tuning diffusion models in many domains (Black et al., 2024; Fan et al., 2023; Ren et al., 2025). However, PPO-based policy-ratio updates require tractable action log-probabilities, which are difficult to obtain for standard multi-step diffusion models such as GenSCO, since the model output depends on latent reverse-chain variables (Ho et al., 2020; Austin et al., 2021).

Motivated by these gaps, we propose **LIFT** (**L**earning schedule improvement via **I**mitation and PPO **F**ine-**T**uning), a refinement-aligned one-step diffusion policy for UAV scheduling. At each refinement step, LIFT observes the current feasible schedule, constructs a schedule-dependent candidate graph, and samples ordered ADD/DEL edge edits. A feasibility-preserving decoder then realizes the sampled edit as a feasible successor schedule. In this way, diffusion is used as a stochastic structured-edit policy inside the refinement loop, rather than as a solution-level generator. LIFT is trained through two stages designed for this sequential refinement process. Imitation pretraining learns from improvement trajectories of a state-of-the-art iterative solver (Guo et al., 2025), and PPO fine-tuning then optimizes the pretrained policy for long-horizon refinement beyond one-step teacher matching. To make PPO tractable, LIFT uses a one-step discrete denoiser to predict the clean edit directly from the current schedule and a sampled noisy edit. This removes latent reverse-chain variables from likelihood evaluation and produces conditional action log-probabilities that can be re-evaluated during policy-ratio updates.

Our work makes three key contributions. **First**, we introduce an *edit-level refinement* for load-dependent UAV scheduling, where the policy directly samples structured ADD/DEL edits on the current feasible schedule. **Second**, we introduce refinement-aligned imitation pretraining, where edit-level expert transitions teach the diffusion model how to act properly in the refinement loop. **Third**, we design a PPO-compatible one-step diffusion policy with tractable edit likelihoods, enabling long-horizon RL fine-tuning. Empirically, LIFT achieves the lowest mean total arrival time across all evaluated benchmark configurations, including substantially larger instances outside the training scale. More broadly, our results reposition diffusion as a trainable policy class for sequential combinatorial improvement, not merely a generator of complete solutions.

2 UAV Scheduling and Iterative Improvement

Load-dependent UAV Scheduling We consider a UAV scheduling problem defined on a set of depots D, customers C, and drones K. A *feasible schedule* must visit every customer exactly once. Each dispatched drone performs at most one closed route that starts from and returns to the same depot, while satisfying flow continuity, depot dispatch limits, payload capacity, and battery constraints (Guo et al., 2025; Pasha et al., 2022; Torabbeigi et al., 2020). Figure 1 presents an example schedule with depot-anchored closed routes. The problem is load-dependent: the flight speed depends on the current payload, and the corresponding energy consumption depends on both travel distance and payload. Therefore, assigning a customer to a different drone can change the load profile, requiring re-evaluation of the affected routes.

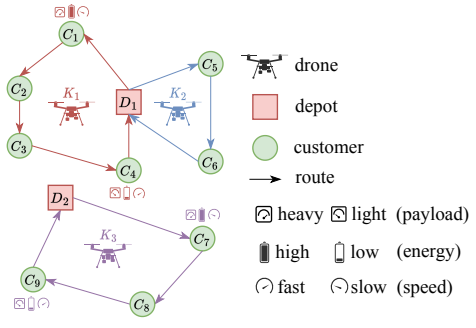

Figure 1: A feasible UAV schedule with depot-anchored closed routes for all drones.

The objective is to find the optimal schedule s^* that minimizes the total arrival time across all customers $J(s^*)$:

$$s^* = \arg\min_s J(s) = \arg\min_s \sum_{k \in K} \sum_{i \in C} t_i^k(s), \tag{1}$$

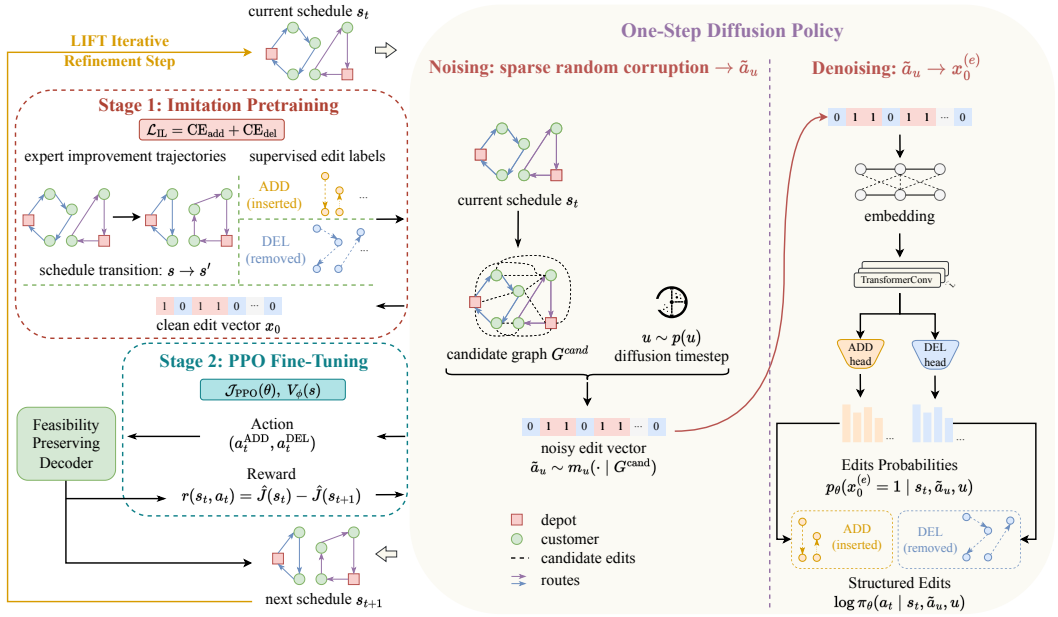

Figure 2: Overview of LIFT. Starting from a feasible schedule, LIFT constructs a schedule-dependent candidate graph, samples ordered ADD/DEL edits with a one-step diffusion policy, and decodes them into a feasible successor schedule for iterative refinement. Training first imitates expert improvement trajectories and then applies PPO fine-tuning with decoded rewards and replayed action log-probabilities.

where $t_i^k(s)$ denotes the arrival time of drone k at customer i according to schedule s. The full mathematical formulation and the drone parameter settings used in the experiments are provided in Appendix C and Appendix E.2, respectively.

Iterative Improvement as Structured Edit We formulate iterative schedule refinement as a *Markov Decision Process* (MDP) $(\mathcal{S}, \mathcal{A}, P, r)$ over feasible schedules. At each refinement step, the state $s_t \in \mathcal{S}$ is the current feasible schedule, and the action $a_t \in \mathcal{A}$ is a structured edit to be performed on s_t. The state-transition P is implemented by the decoder introduced in Section 3.3, which maps the structured edit on s_t to a feasible successor schedule s_{t+1}. Consistent with the objective of minimizing the total arrival time $J(s)$ in equation 1, we define the stepwise reward as

$$r(s_t, a_t) = \hat{J}(s_t) - \hat{J}(s_{t+1}), \tag{2}$$

where $\hat{J}(s) = J(s)/J(s_0)$ is the total arrival time normalized by that of the initial schedule s_0, keeping reward magnitudes comparable across scales. Because the normalization in $\hat{J}$ is fixed per instance, the undiscounted rewards telescope, $\sum_{t=0}^{T-1} r(s_t, a_t) = \hat{J}(s_0) - \hat{J}(s_T)$, turning the total improvement into dense per-step feedback while leaving the undiscounted objective unchanged. For long-horizon credit assignment, generalized advantage estimation (GAE) (Schulman et al., 2015) allows an edit with a small or negative immediate reward to receive a positive advantage when it enables larger subsequent gains.

3 Proposed Approach

LIFT instantiates the iterative schedule refinement process. An overview of the refinement pipeline is shown in the lower part of Figure 2. Starting from a current feasible schedule, LIFT first constructs a schedule-dependent candidate graph that exposes plausible local edits (Section 3.1). A one-step diffusion policy then operates on it to predict structured ADD/DEL edits (Section 3.2). These sampled edits are realized by a feasibility-preserving decoder, which maps them to a feasible successor

schedule (Section 3.3). The policy itself is trained with a two-stage refinement-aligned pipeline that combines imitation pretraining with PPO fine-tuning (Section 3.4).

At inference time, the learned LIFT policy is embedded in a *multi-branch refinement search algorithm* (Section 3.5). This design is inspired by classical and recent local-search frameworks for routing and scheduling (Cordeau & Maischberger, 2012; Mladenović & Hansen, 1997; Guo et al., 2025; Li et al., 2025), in which parallel search improves exploration, Tabu memory reduces short-term cycling, and perturbation helps escape local stagnation. In LIFT, these classical search components are retained because they are well matched to iterative refinement, while hand-crafted search operators are replaced by diffusion-guided schedule edits.

3.1 Schedule-Dependent Candidate Graph

Any feasible UAV schedule can be represented as a directed *schedule graph*, where the node set $V = D \cup C$ consists of all depots D and customers C, and each directed edge (i, j) indicates that a drone travels directly from node i to node j. Under this graph representation, each depot-anchored closed route corresponds to a directed path to be followed by a drone in the schedule. Hence, schedule refinement is achieved by editing a collection of directed edges.

Treating every possible directed edge as a candidate edit would create a large action space at each refinement step, although most such edges are unlikely to yield useful local improvements. LIFT therefore constructs a sparse *candidate graph* $G^{\mathrm{cand}} = (V, E^{\mathrm{cand}})$ based on the current schedule graph $G^{\mathrm{cur}} = (V, E^{\mathrm{cur}})$ to expose a compact set of plausible local edits. The candidate edge set retains the current schedule edges and augments them with sparse local alternatives:

$$E^{\mathrm{cand}} \;=\; E^{\mathrm{cur}} \;\cup\; E^{\mathrm{knn}} \;\cup\; E^{\mathrm{2hop}} \;\cup\; E^{\mathrm{dep}}, \tag{3}$$

where E^{knn} and E^{2hop} provide customer-to-customer alternatives around the current schedule, while E^{dep} introduces additional depot-customer connectivity. A lightweight validity filter further removes clearly inadmissible candidate edges. Detailed edge construction and filtering rules are deferred to Appendix D.2.

3.2 One-Step Diffusion Policy

LIFT parameterizes its diffusion policy with a one-step denoiser. At each refinement step t, the policy samples an action a_t that specifies the structured edit for the decoder (Section 3.3). To support PPO fine-tuning, the probability of each sampled action must be re-evaluable during policy updates. A standard multi-step reverse chain in diffusion models makes this cumbersome because the final action depends on latent reverse-chain variables (Ho et al., 2020; Austin et al., 2021). LIFT avoids this difficulty by replacing the multi-step reverse chain with one denoising step that directly predicts the structured edit used by the decoder, as illustrated in Figure 2.

Concretely, LIFT samples a discrete diffusion timestep $u \sim p(u)$ and then generates a sparse *binary edge edit vector* $\tilde{a}_u \in \{0,1\}^{|E^{\mathrm{cand}}|}$ (Appendix D.3). Conditioned on the current state s_t, the noisy edge edit vector $\tilde{a}_u$, and the diffusion timestep u, the denoiser directly predicts the probability that each candidate edge $e \in E^{\mathrm{cand}}$ should be selected:

$$p_\theta(x_0^{(e)} = 1 \mid s_t, \tilde{a}_u, u), \qquad e \in E^{\mathrm{cand}}, \tag{4}$$

where $x_0 \in \{0,1\}^{|E^{\mathrm{cand}}|}$ is the clean edit vector, and $x_0^{(e)} \in \{0,1\}$ indicates whether edge e is selected for edit.

To instantiate this conditional distribution in equation 4, the denoiser is implemented as a message-passing neural network on the candidate graph G^{cand}. Each node $i \in V$ has a feature vector $\mathbf{f}_i = [\tilde{x}_i, \tilde{y}_i, q_i, \mathbb{I}_{i \in D}]$, where $\tilde{x}_i$ and $\tilde{y}_i$ are node coordinates, q_i is demand, and $\mathbb{I}_{i \in D}$ indicates whether i is a depot. Each candidate edge $e = (i, j) \in E^{\mathrm{cand}}$ has a feature vector $\mathbf{f}_e = [b_e, d_e]$, where $b_e = \mathbb{I}_{e \in E^{\mathrm{cur}}}$ indicates whether e belongs to the current schedule, and d_e is the Euclidean distance between i and j.

The network also takes the sampled diffusion timestep u and the noisy edit value $\tilde{a}_u^{(e)}$ as inputs. Node features are first projected into hidden embeddings. For each candidate edge, we combine its edge features, noisy edit value, and timestep embedding to form the edge input used in message passing.

These node and edge inputs are processed by stacked TransformerConv layers (Shi et al., 2021), producing final node embeddings $\mathbf{h}_i$. Meanwhile, the edge indicator b_e splits candidate edges into two sets:

$$E^{\mathrm{DEL}} = \{e \in E^{\mathrm{cand}} : b_e = 1\}, \quad E^{\mathrm{ADD}} = E^{\mathrm{cand}} \setminus E^{\mathrm{DEL}}. \tag{5}$$

Edges in E^{DEL} are candidates to be removed from the current schedule, while edges in E^{ADD} are candidates to be added to the schedule. Two separate neural heads score these two sets of edges:

$$\ell_e^{(d)} = f_d([\mathbf{h}_i \| \mathbf{h}_j]), \quad \forall e = (i,j) \in E^d, \quad d \in \{\mathrm{ADD}, \mathrm{DEL}\}. \tag{6}$$

The sigmoid value $\sigma(\ell_e^{(d)})$ gives the predicted probability that edge e should be selected, and is used for per-edge imitation learning. For PPO fine-tuning and inference, the logits are converted into positive Plackett-Luce weights:

$$w_e^{(d)} = \exp\left(\ell_e^{(d)}/\tau_d\right), \quad e \in E^d, \quad d \in \{\mathrm{ADD}, \mathrm{DEL}\}, \tag{7}$$

where τ_d is the sampling temperature for set d.

The sampled action is an ordered pair $a_t = (a_t^{\mathrm{ADD}}, a_t^{\mathrm{DEL}})$. For each $d \in \{\mathrm{ADD}, \mathrm{DEL}\}$, the ordered list $a_t^d = (e_1^d, \ldots, e_{k_d}^d)$ is sampled without replacement from E^d, where $k_d = \min(K_d, |E^d|)$ and K_d is the edit budget. We implement this using Gumbel-TopK (Kool et al., 2019b): independent Gumbel noise is added to $\log w_e^{(d)}$, and the top-k_d perturbed edges are selected in order. This gives an ordered sample from the Plackett-Luce distribution (Plackett, 1975). The corresponding action selection log-probability is

$$\log \pi_\theta(a_t \mid s_t, \tilde{a}_u, u) = \sum_{d \in \{\mathrm{ADD},\mathrm{DEL}\}} \sum_{j=1}^{k_d} \log \frac{w_{e_j^d}^{(d)}}{\sum_{e \in C_j^d} w_e^{(d)}}, \tag{8}$$

where C_j^d is the set of remaining candidates before the j-th draw from $d \in \{\mathrm{ADD}, \mathrm{DEL}\}$. The resulting ordered edge lists $(a_t^{\mathrm{ADD}}, a_t^{\mathrm{DEL}})$ are then passed to the decoder, which produces a feasible successor schedule.

3.3 Feasibility-Preserving Decoder

The diffusion policy produces ordered ADD and DEL edge lists, which are not themselves a feasible UAV schedule. Applying them directly may violate route continuity, depot anchoring, payload capacity, or battery constraints. LIFT therefore uses a decoder to convert the selected edges into a feasible successor schedule.

Given the current schedule s_t and the sampled action $a_t = (a_t^{\mathrm{ADD}}, a_t^{\mathrm{DEL}})$, the decoder identifies customers to be reassigned and reinserts them into feasible route positions. Specifically, each selected DEL edge (i, j) corresponds to an existing edge in the current schedule. Selecting (i, j) indicates that the connection from i to j should be removed, so its customer endpoints are added to the reassignment set. The decoder ranks these customers by their aggregated DEL support and considers several candidate subsets of the top-ranked customers. For each subset, the decoder removes the selected customers from s_t and reinserts them one by one into feasible route positions.

For each removed customer c, a feasible insertion position is defined by an existing edge (i, j) along a drone route. Namely, inserting c between i and j replaces (i, j) with (i, c) and (c, j) while preserving feasibility. Among all feasible positions for c, the decoder first keeps the top-M positions with the lowest insertion cost, where M is a fixed decoding budget, and then inserts c at the retained position with the strongest support from a_t^{ADD}, using insertion cost only to break ties. Insertion cost thereby ensures feasible and locally reasonable choices, while the diffusion-guided ADD scores provide the additional improvement. Each candidate subset thus yields one decoded schedule, and the decoder returns the feasible one with the lowest total arrival time in equation 1. If no feasible schedule is found, the action is rejected and $s_{t+1} = s_t$. Thus, every accepted successor is a feasible UAV schedule. The full decoding algorithm is provided in Appendix D.4.

3.4 TWO-STAGE REFINEMENT-ALIGNED TRAINING

We develop a new two-stage training pipeline in LIFT. The first stage employs imitation learning to teach the diffusion policy introduced in Section 3.2 how a strong solver iteratively improves a schedule. The second stage exploits PPO to further fine-tune the policy for long-horizon improvement under the reward in equation 2.

Imitation pretraining We collect expert improvement trajectories from GENIS, a state-of-the-art iterative refinement algorithm for UAV scheduling (Guo et al., 2025). Each state-transition in a collected trajectory contains a current schedule s and an improved successor schedule s'. After constructing the candidate graph G^{cand} of s, we label each candidate edge according to how GENIS changes the schedule:

$$y_e^{\text{ADD}} = \mathbb{I}_{e \in E(s')} \quad e \in E^{\text{ADD}}, \quad y_e^{\text{DEL}} = \mathbb{I}_{e \notin E(s')} \quad e \in E^{\text{DEL}}. \tag{9}$$

Here, an ADD label is positive if the edge appears in the successor schedule produced by GENIS, and a DEL label is positive if that edge in the current schedule is removed by GENIS.

During pretraining, the noisy edit mask is sampled from the same fixed sparse prior used later in PPO fine-tuning and inference (Appendix D.3). We do not generate the noisy input by modifying the edits made by GENIS, which is inaccessible during practical use of LIFT. This keeps the noisy-input distribution consistent across all stages of LIFT. Since positive edit labels are sparse, we retain all positive labels and subsample negative labels separately for the ADD and DEL edges. Consequently, the imitation loss is defined as the sum of the following two cross-entropy losses:

$$\mathcal{L}_{\text{IL}} = \text{CE}_{\text{ADD}} + \text{CE}_{\text{DEL}}. \tag{10}$$

PPO fine-tuning After imitation pretraining, PPO is further utilized to enhance the diffusion policy's ability to improve UAV schedules over a sequence of refinement steps. At any step m, the policy samples an action a_m using the one-step denoiser in Section 3.2. The corresponding edge edit vector and diffusion step $(\tilde{a}_{u_m}, u_m)$ are stored so that the action log-probability in equation 8 can be easily recomputed during PPO updates. The policy is updated with the clipped PPO objective plus an entropy bonus:

$$\mathcal{J}_{\text{PPO}}(\theta) = \mathbb{E}_m \left[\min\Big(\rho_m(\theta)\hat{A}_m, \text{clip}(\rho_m(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_m \Big) \right] + \lambda\, \mathbb{E}_m[\mathcal{H}_m], \tag{11}$$

where $\rho_m(\theta)$ is the likelihood ratio, $\hat{A}_m$ is the advantage estimated by GAE, and $\mathcal{H}_m$ is the policy entropy.

Imitation pretraining provides a reliable starting policy for effective schedule edits, while PPO fine-tuning further optimizes the policy for continued improvement beyond one-step expert matching. A separate critic $V_\phi(s)$ is trained to fit the return targets. The full PPO training procedure is summarized in Appendix D.5.

3.5 INFERENCE-TIME REFINEMENT SEARCH

At inference time, LIFT embeds the learned policy inside a multi-branch refinement search algorithm. Starting from an initial feasible schedule, each branch repeatedly builds the schedule-dependent candidate graph, samples several actions from the one-step diffusion policy, and decodes each with the decoder in Section 3.3, keeping the lowest-cost feasible successor schedule. The branch moves to this successor only when it improves the current schedule, while tracking the best feasible schedule found so far.

In our experiments, we run four branches in parallel to effectively exploit the stochasticity of diffusion sampling. Different sampling temperatures encourage different branches to explore varied sequences of actions from the same initial schedule. This follows the spirit of parallel iterated local search (Cordeau & Maischberger, 2012), but replaces hand-crafted operator choices with flexible schedule edits generated by the learned policy. Each branch also maintains a Tabu list to avoid repeated reassignment of the same customers in consecutive improvement steps.

If a branch fails to improve for 16 consecutive refinement steps, LIFT perturbs its best-so-far schedule and restarts refinement from the perturbed schedule. This *shake* operation utilizes standard

neighborhood perturbation operators (Mladenović & Hansen, 1997; Guo et al., 2025; Li et al., 2025) to help the search escape stagnation; a branch stops after 15 unproductive shakes. After all branches finish, LIFT returns the best feasible schedule found across all branches and refinement steps. The full inference procedure is provided in Appendix D.6.

4 EXPERIMENTS

4.1 EXPERIMENT SETUP

Benchmark, Training Data, and Implementation We evaluate LIFT on the load-dependent UAV scheduling benchmark introduced by GENIS (Guo et al., 2025), covering 50 to 360 customers, 2 to 9 depots, and 2 to 14 drones per depot. We use these benchmark instances strictly for evaluation. For training, we generate synthetic problem instances by first grouping the benchmarks into 13 classes based on $(|C|, |D|)$. For each class, we extract summary statistics of customer locations, depot locations, and demands, and then sample new synthetic instances with identical statistical properties (Appendix E.1). Unless otherwise stated, all experiments use 350 synthetic training instances from a 7-class subset covering $|C| \in [50, 160]$. The larger classes with $|C| > 160$ are reserved for out-of-training-scale evaluation. Following Guo et al. (2025); Meng et al. (2023), all training instances explicitly consider practical UAV operating constraints, with full details in Appendix E.2. All experiments are conducted on a server with $2\times$ NVIDIA H100 GPUs, and the key hyperparameters are summarized in Table 4.

Evaluation Metric Following Guo et al. (2025); Torabbeigi et al. (2020); Pasha et al. (2022), we report the total customer arrival time per instance, measured in seconds and defined in equation 1, which is the optimization objective of our load-dependent UAV scheduling problem.

Baselines We compare LIFT with strong baselines across the main competing approaches. All methods use the same budget of at most 8,000 schedule evaluations, matching LIFT's default inference budget (Appendix F.2). **MLG**, a greedy constructor tailored to the minimum-latency objective (Silva et al., 2012), and **ACO** (Bell & McMullen, 2004), a classical metaheuristic, serve as non-learned references that provide continuity with prior scheduling and routing studies (Solomon, 1987). **ALNS** (Ropke & Pisinger, 2006), **VNS** (Mladenović & Hansen, 1997), and an **LKH-3**-style edge-exchange search (Helsgaun, 2017) are strong classical searches adapted to load-dependent UAV scheduling: payload, battery use, and arrival times are recomputed after every candidate move, while each method retains its original search logic. Implementation and adaptation details are given in Appendix E.4. **GENIS** (Guo et al., 2025) is the state-of-the-art learning-based neighborhood search method for UAV scheduling, combining dual-graph neural state encoding with online reinforcement learning for adaptive operator selection. **AFL** (Zhang et al., 2026) is a recent agentic LLM approach for complex routing problems, representing methods that generate or compose heuristic search procedures with large language models. **GenSCO** (Li et al., 2025) uses diffusion as a search operator inside a disruption-enhancement loop, and is the closest diffusion-based baseline to LIFT, since both use diffusion within an improvement-oriented search process.

4.2 MAIN RESULTS AND DISCUSSIONS

Table 1 reports the main results in total customer arrival time; for stochastic methods, each entry is the mean and standard deviation over 10 random seeds. LIFT achieves the lowest mean total arrival time on every evaluated instance, improving over GENIS, ALNS, VNS, LKH-3, MLG, AFL, GenSCO, and ACO by 3.5%, 3.6%, 4.0%, 7.6%, 9.5%, 10.2%, 13.1%, and 41.9% on average, respectively. Under the matched budget, no baseline is both faster and better: GENIS, VNS, GenSCO, and LKH-3 take 2.4–8.3 times longer to reach their best schedules yet still perform worse (Appendix F.2), and Figure 3 shows that LIFT reaches near-final solution quality using a much smaller fraction of the refinement budget than GENIS and GenSCO. LIFT also achieves the best makespan on 14 of the 15 configurations although it does not optimize makespan directly, and 99.98% of its sampled actions decode to feasible successor schedules, with non-improving edits leaving the schedule unchanged (Appendices F.1 and F.7). Representative LIFT schedules are shown in Appendix F.8.

Table 1: Main results on the 15 public benchmark instances in total customer arrival time (10^2 seconds; mean (standard deviation) over 10 seeds, MLG being deterministic). Lower is better; the best and second-best means are in bold and underlined. LIFT is significantly better than every baseline on every instance (Wilcoxon rank-sum test, $\alpha = 0.05$). Instance labels `C#_D#_K#` denote the numbers of customers, depots, and drones per depot.

Instance	MLG	ACO	ALNS	VNS	LKH-3	GENIS	AFL	GenSCO	LIFT
C50_D4_K4	48.45	65.87 (1.48)	47.92 (0.18)	47.98 (0.21)	49.78 (1.20)	47.86 (0.10)	48.97 (0.34)	52.25 (1.01)	**47.76 (0.00)**
C50_D4_K2	78.71	94.03 (2.83)	72.24 (0.85)	72.41 (1.21)	76.50 (1.17)	72.15 (1.58)	75.13 (0.86)	80.16 (1.33)	**70.90 (0.53)**
C75_D5_K3	81.78	110.04 (4.43)	77.35 (0.47)	77.42 (0.46)	81.07 (0.61)	77.28 (0.79)	78.95 (0.44)	87.53 (1.74)	**76.03 (0.46)**
C80_D2_K5	263.50	396.79 (15.38)	244.87 (0.94)	245.71 (0.77)	246.30 (1.20)	244.40 (0.66)	244.26 (0.97)	276.38 (6.32)	**239.56 (0.29)**
C100_D2_K8	168.95	259.49 (14.49)	150.86 (2.21)	151.12 (0.91)	151.78 (0.77)	150.65 (1.62)	177.24 (0.57)	161.80 (0.88)	**147.89 (0.89)**
C100_D2_K5	192.81	294.99 (7.12)	183.05 (0.82)	183.36 (1.04)	183.72 (2.08)	182.91 (1.80)	229.65 (1.00)	212.97 (3.74)	**175.03 (1.44)**
C100_D3_K6	122.20	198.23 (5.50)	117.62 (0.58)	117.95 (0.79)	120.33 (1.34)	117.48 (0.73)	120.56 (0.64)	122.54 (2.78)	**105.34 (0.59)**
C100_D4_K4	142.56	220.54 (6.82)	131.44 (1.16)	131.86 (1.31)	142.94 (2.83)	131.23 (1.44)	149.75 (0.85)	145.20 (1.01)	**129.06 (1.00)**
C160_D4_K5	522.08	888.39 (23.11)	477.15 (1.92)	480.20 (2.85)	483.46 (2.78)	476.32 (3.63)	489.25 (4.17)	537.58 (3.00)	**465.35 (0.73)**
C240_D6_K5	781.80	1415.42 (18.07)	714.28 (1.94)	715.11 (3.25)	752.62 (9.30)	713.03 (3.76)	739.66 (4.41)	797.21 (6.86)	**709.55 (3.26)**
C249_D2_K14	1065.25	1624.41 (43.73)	1021.35 (3.87)	1024.58 (6.15)	1053.79 (12.68)	1019.63 (6.22)	1175.94 (5.18)	1070.04 (3.36)	**971.73 (2.79)**
C249_D3_K12	795.24	1335.35 (17.36)	759.24 (1.83)	767.42 (3.67)	809.42 (7.33)	757.70 (4.37)	864.54 (4.95)	881.28 (6.69)	**713.32 (3.62)**
C249_D4_K8	782.91	1332.29 (32.94)	751.13 (2.16)	758.60 (5.54)	777.05 (2.06)	749.42 (2.64)	781.59 (8.98)	827.70 (4.39)	**708.18 (3.28)**
C249_D5_K6	837.41	1475.88 (25.63)	781.46 (3.02)	790.85 (4.40)	870.27 (8.87)	779.57 (5.12)	838.10 (8.37)	887.33 (6.48)	**749.72 (2.05)**
C360_D9_K5	1154.32	2193.49 (41.37)	1093.28 (2.87)	1097.65 (9.53)	1182.74 (9.64)	1090.94 (4.14)	1121.49 (3.89)	1222.40 (7.07)	**1028.57 (4.28)**

These gains reflect action granularity. GenSCO regenerates complete solutions, whereas LIFT learns edit-level decisions on the current schedule; since a single customer reassignment propagates through downstream payload, speed, energy, and arrival time, attributing improvement feedback to localized edits yields a more direct refinement signal than solution-level regeneration. The same argument explains the gains over GENIS, whose policy is constrained by predefined operators, and over the remaining baselines, which rely on hand-crafted or LLM-generated heuristics rather than a learned edit policy.

4.3 Ablation Study

To isolate the contributions of the two training stages and of diffusion conditioning, we evaluate four ablated variants: **LIFT (imitation only)** removes PPO fine-tuning; **LIFT (PPO only)** trains the randomly initialized diffusion policy directly with PPO; **LIFT-NoDiff** removes diffusion conditioning alone; **Random-Edit** replaces the learned ADD/DEL scores with independent $U(0, 1)$ samples. All variants keep the candidate graph, decoder, and multi-branch inference search unchanged. Figure 3 shows the refinement trajectories of all variants; per-instance results (Table 7), detailed comparisons, and the PPO training curves (Figure 5) are given in Appendix F.3.

Effect of two-stage training The two stages are complementary. LIFT (PPO only) is clearly worse than full LIFT on almost all configurations and weaker than LIFT (imitation only) in most cases, with the gap widening on the larger out-of-training-scale instances: under a practical PPO budget ($\leq$200k steps and $\leq$24 hours of training), a randomly initialized policy must both discover useful edits and learn how they interact over multiple refinement steps. Conversely, full LIFT outperforms LIFT (imitation only) on 14 of the 15 configurations, with the largest gains on C249 and C360, so expert-edit matching alone is insufficient for long-horizon improvement.

Effect of diffusion conditioning LIFT-NoDiff shares the same expert trajectories, PPO fine-tuning, and search pipeline as full LIFT, whereas Random-Edit removes learned scoring altogether. Full LIFT outperforms both on 14 of the 15 configurations and ties on the smallest, by 3.8% on average and 5.0–5.2% beyond the training scale, so its gains cannot be explained by PPO alone. LIFT-NoDiff has no consistent advantage over Random-Edit: it always produces the same edge ranking for a given schedule, whereas LIFT conditions its scores on resampled noisy edits and timesteps, yielding varied learned distributions for the same schedule.

Effect of search components and expert choice Replacing ADD-guided reinsertion with cost-only insertion, disabling Tabu memory, and disabling shake worsen performance by 3.76%, 3.94%, and 4.22%, respectively, and the greedy branch alone is 7.38% worse than full LIFT whereas each

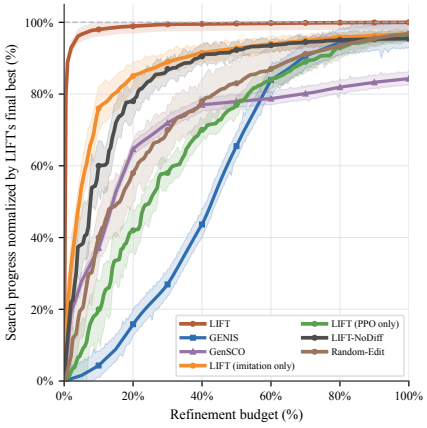

Figure 3: Refinement progress on C50_D4_K2, C75_D5_K3, and C80_D2_K5, normalized by LIFT's final best objective; higher is better.

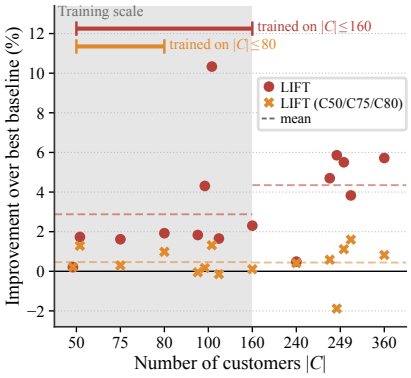

Figure 4: Improvement in total arrival time over the best baseline on each benchmark instance versus instance size $|C|$; higher is better.

stochastic branch alone is only 0.72–0.93% worse (Appendix F.3). Retraining LIFT on ALNS instead of GENIS trajectories (**LIFT-A**) still outperforms both ALNS and GENIS on all 15 configurations, with full LIFT only 1.1% better on average, so LIFT's gains do not depend on a specific expert (Appendix F.4).

4.4 Generalization, Sensitivity, and Robustness

The main LIFT policy is trained on $|C| \in [50, 160]$. A restricted variant, **LIFT (C50/C75/C80)**, trained only on the three smallest classes, still outperforms GENIS on 12 of the 15 configurations, including 5 of the 6 out-of-training-scale instances (Figure 4), so the learned refinement behavior transfers across scales rather than memorizing training sizes. The full 7-class model matches or improves on it on every configuration, and its margin over the best baseline grows from 2.9% within the training scale to 4.3% beyond it (Appendix F.5). LIFT is also robust to its inference-time settings: the refinement budget matters most on the largest instances, shake is necessary while its threshold is not, and halving the candidate-graph neighborhood size or changing the temperature schedule alters total arrival time by less than 1%. Under a nonlinear load-dependent power model, the trained policies remain usable without retraining, with total arrival time increasing by only 4.0% on average (Appendix F.6).

5 Conclusion

In this paper, we introduced LIFT, a refinement-aligned one-step diffusion policy that places diffusion inside the iterative refinement loop for load-dependent UAV scheduling. LIFT proposes schedule edits, decodes them into feasible successor schedules, and fine-tunes the policy with PPO for long-horizon improvement. Its design combines refinement-aligned imitation learning from expert improvement trajectories with a one-step denoiser that provides tractable action likelihoods for policy-gradient updates. Experiments show that LIFT achieves the lowest mean total arrival time across all benchmark configurations and generalizes to larger problem instances without retraining, while ablations confirm the complementary roles of imitation learning, PPO fine-tuning, diffusion conditioning, and the search components.

More broadly, our study points to a richer role for diffusion in combinatorial optimization. Rather than serving only as a generator of complete solutions, diffusion can become a trainable policy class for sequential improvement, where each sampled action contributes to a longer refinement process. This opens a promising future direction for refinement-aligned diffusion policies in scheduling, planning, and resource allocation problems, especially when high-quality solutions emerge through many dependent improvement steps rather than a single generative pass.

ETHICS STATEMENT

This work studies load-dependent UAV scheduling as a combinatorial optimization problem. It does not involve human subjects, personally identifiable information, or sensitive data. All experiments are conducted on synthetic instances generated by us and on publicly available benchmark instances. The intended societal impact is positive, as more efficient UAV schedules can reduce delivery latency and energy use in logistics and emergency response. Practical deployment would still require validation under real-world safety, regulatory, and operational constraints, as discussed in Appendix A. We do not foresee negative ethical implications from this research.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide: (1) the full mathematical formulation of the scheduling problem in Appendix C; (2) complete pseudo-code for initialization, candidate graph construction, decoding, PPO fine-tuning, and multi-branch inference search in Appendix D; (3) the synthetic instance generation procedure, drone and benchmark parameters, and all model, training, and search hyperparameters in Appendix E; and (4) implementation and adaptation details of all baselines in Appendix E.4. The source code and scripts for reproducing all experiments are included in the supplementary material.

AI USE STATEMENT

During manuscript preparation, we used a large language model to assist with language polishing, grammar correction, and rephrasing. We carefully reviewed and edited the LLM-generated text to ensure accuracy and originality. The LLM was not used to generate experimental results, algorithm designs, neural network architectures, or other core technical contributions. We take responsibility for the final content of this work.

REFERENCES

Amazon. Amazon Prime Air's drone takes flight with AWS and Siemens. `https://aws.amazon.com/cn/partners/success/prime-air-siemens/`, 2020. Accessed: 2026-09-02.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, 2021.

John E Bell and Patrick R McMullen. Ant colony optimization techniques for the vehicle routing problem. *Advanced Engineering Informatics*, 18(1):41–48, 2004.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017.

Francesco Betti Sorbelli. UAV-based delivery systems: A systematic review, current trends, and research challenges. *Journal on Autonomous Transportation Systems*, 1(3):1–40, 2024.

Kevin Black, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. Training diffusion models with reinforcement learning. In *International Conference on Learning Representations*, 2024.

Jean-François Cordeau and Mirko Maischberger. A parallel iterated tabu search heuristic for vehicle routing problems. *Computers & Operations Research*, 39(9):2033–2050, 2012.

Kevin Dorling, Jordan Heinrichs, Geoffrey G Messier, and Sebastian Magierowski. Vehicle routing problems for drone delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47 (1):70–85, 2016.

Ying Fan, Olivia Watkins, Yuqing Du, Hao Liu, Moonkyung Ryu, Craig Boutilier, Pieter Abbeel, Mohammad Ghavamzadeh, Kangwook Lee, and Kimin Lee. DPOK: Reinforcement learning for fine-tuning text-to-image diffusion models. In *Advances in Neural Information Processing Systems*, 2023.

Tong Guo, Yi Mei, Wenbo Du, Yisheng Lv, Yumeng Li, and Tao Song. Emergency scheduling of aerial vehicles via graph neural neighborhood search. *IEEE Transactions on Artificial Intelligence*, 6(7):1808–1822, 2025.

Keld Helsgaun. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 12:966–980, 2017.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, 2020.

André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *European Conference on Artificial Intelligence*, 2020.

Haiping Huang, Chengxi Hu, Jie Zhu, Min Wu, and Reza Malekian. Stochastic task scheduling in UAV-based intelligent on-demand meal delivery system. *IEEE Transactions on Intelligent Transportation Systems*, 23(8):13040–13054, 2022.

Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

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

Wouter Kool, Herke Van Hoof, and Max Welling. Stochastic beams and where to find them: The Gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, 2019b.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, 2020.

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

Yang Li, Lvda Chen, Haonan Wang, Runzhong Wang, and Junchi Yan. Generation as search operator for test-time scaling of diffusion-based combinatorial optimization. In *Advances in Neural Information Processing Systems*, 2025.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, 2024.

Yang Liu, Chuan Zhou, Yancheng Chen, Shuai Zhang, Xixun Lin, and Xiaoqing Wang. Hard constraints meet soft generation: Guaranteed feasibility for LLM-based combinatorial optimization. *arXiv preprint arXiv:2602.01090*, 2026.

Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. In *Advances in Neural Information Processing Systems*, 2023.

Shanshan Meng, Xiuping Guo, Dong Li, and Guoquan Liu. The multi-visit drone routing problem for pickup and delivery services. *Transportation Research Part E: Logistics and Transportation Review*, 169:102990, 2023.

Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

Junayed Pasha, Zeinab Elmi, Sumit Purkayastha, Amir M Fathollahi-Fard, Ying-En Ge, Yui-Yip Lau, and Maxim A Dulebenets. The drone scheduling problem: A systematic state-of-the-art review. *IEEE Transactions on Intelligent Transportation Systems*, 23(9):14224–14247, 2022.

Robin L Plackett. The analysis of permutations. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 24(2):193–202, 1975.

Boualem Rabta, Christian Wankmüller, and Gerald Reiner. A drone fleet model for last-mile distribution in disaster relief operations. *International Journal of Disaster Risk Reduction*, 28: 107–112, 2018.

Allen Ren, Justin Lidard, Lars Ankile, Anthony Simeonov, Pulkit Agrawal, Anirudha Majumdar, Benjamin Burchfiel, Hongkai Dai, and Max Simchowitz. Diffusion policy policy optimization. In *International Conference on Learning Representations*, 2025.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. In *International Joint Conference on Artificial Intelligence*, 2021.

Marcos Melo Silva, Anand Subramanian, Thibaut Vidal, and Luiz Satoru Ochi. A simple and effective metaheuristic for the minimum latency problem. *European Journal of Operational Research*, 221 (3):513–520, 2012.

Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.

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

Maryam Torabbeigi, Gino J Lim, and Seon Jin Kim. Drone delivery scheduling optimization considering payload-induced battery consumption rates. *Journal of Intelligent & Robotic Systems*, 97(3):471–487, 2020.

UPS. UPS expands its medical drone service to University of California San Diego Health. `https://www.ups.com/nz/en/healthcare/news/press-releases/medical-drone-service-california`, 2020. Accessed: 2026-09-02.

Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. DiGress: Discrete denoising diffusion for graph generation. In *International Conference on Learning Representations*, 2023.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, 2015.

Wing. Wing drone delivery. `https://wing.com/`, 2026. Accessed: 2026-09-02.

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. *IEEE Transactions on Neural Networks and Learning Systems*, 33 (9):5057–5069, 2021.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, 2024.

Juan Zhang, James F Campbell, Donald C Sweeney II, and Andrea C Hupman. Energy consumption models for delivery drones: A comparison and assessment. *Transportation Research Part D: Transport and Environment*, 90:102668, 2021.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. An agentic framework with LLMs for solving complex vehicle routing problems. In *International Conference on Learning Representations*, 2026.

Zipline. Drone delivery for food, groceries and medicine. `https://www.flyzipline.com/`, 2026. Accessed: 2026-09-02.

APPENDIX

Table of Contents

A Broad Impacts, Limitations and Future Work **13**

B Related Work **14**

- **B.1 UAV Scheduling and Classical Search** **14**
- **B.2 Learning-Based Construction and Improvement** **14**
- **B.3 Diffusion Models and RL Fine-Tuning** **15**

C Problem Formulation **15**

D Algorithmic Procedures **18**

- **D.1 Regret-Based Initialization** **18**
- **D.2 Candidate Graph Construction** **19**
- **D.3 Sparse Prior Construction for One-Step Diffusion** **19**
- **D.4 Feasibility-Preserving Decoder** **20**
- **D.5 PPO Fine-Tuning Procedure** **21**
- **D.6 Multi-Branch Inference Search** **21**

E Implementation and Reproducibility Details **22**

- **E.1 Synthetic Instance Generation** **22**
- **E.2 Drone and Benchmark Parameters** **24**
- **E.3 Model, Training, and Search Hyperparameters** **24**
- **E.4 Baseline Implementation and Adaptation Details** **24**

F Supplementary Experimental Results **25**

- **F.1 Supplementary Results for the Main Benchmarks** **25**
- **F.2 Runtime Comparison** **26**
- **F.3 Detailed Ablation Results** **27**
- **F.4 Dependence on the Expert Solver** **29**
- **F.5 Generalization Beyond Training Scale** **29**
- **F.6 Parameter Sensitivity and Robustness Analysis** **30**
- **F.7 Failure-Case and Rejected-Action Analysis** **32**
- **F.8 Representative Schedule Visualizations** **34**

A BROAD IMPACTS, LIMITATIONS AND FUTURE WORK

LIFT explores refinement-aligned diffusion policies for load-dependent UAV scheduling, showing that diffusion can be used not only to generate complete solutions, but also to act as an edit-level policy inside an iterative improvement loop. This may benefit autonomous logistics, emergency response, and other constrained routing or scheduling applications where feasible schedules must be improved through many dependent local decisions. Practical deployment would still require additional validation under real-world safety, sensing, regulatory, and operational constraints.

Current limitations include the following. First, although LIFT generalizes to larger benchmark instances without retraining, maintaining the same improvement rate on substantially larger instances

and longer refinement horizons remains an open challenge. Second, our experiments are confined to load-dependent UAV scheduling; robustness on harder distributions and other constrained routing or scheduling variants remains to be tested. Moreover, although the trained policies remain effective without retraining under a nonlinear load-dependent power model (Appendix F.6), other sources of simulator-to-real mismatch, such as wind, battery degradation, and low-level flight-control errors, are not modeled. Third, LIFT relies on expert improvement trajectories for imitation pretraining. Retraining with a different expert shows that its gains do not depend on a specific teacher (Appendix F.4), but removing the need for an expert solver altogether is an important direction for future work. Improving long-horizon search efficiency, robustness under distribution shift, and applicability to broader planning and resource allocation problems are further promising future directions.

B Related Work

B.1 UAV Scheduling and Classical Search

UAV scheduling couples customer assignment, depot dispatch, vehicle usage, service order, payload capacity, battery feasibility, and time propagation. Existing formulations often use mixed-integer routing variables together with auxiliary continuous variables for payload, energy, and arrival time (Pasha et al., 2022; Torabbeigi et al., 2020). Load-dependent variants further make the selected service sequence interact with payload-induced speed and energy consumption (Dorling et al., 2016; Zhang et al., 2021). As a result, changing one customer assignment or route position can affect downstream payload, battery, and arrival-time profiles, making both objective evaluation and feasibility checking state-dependent.

Classical routing and scheduling solvers rely on strong hand-designed search mechanisms, including LKH-3 (Helsgaun, 2017), adaptive large neighborhood search (ALNS) (Ropke & Pisinger, 2006), and variable neighborhood search (VNS) (Mladenović & Hansen, 1997). These methods repeatedly modify a current solution through local moves, recombination, destroy-repair procedures, or neighborhood changes. They are highly effective for per-instance optimization, but their improvement behavior is specified by manually designed operators and acceptance rules. In load-dependent UAV scheduling, such operators must also preserve depot anchoring, customer uniqueness, payload feasibility, battery feasibility, and time consistency after each modification. LIFT keeps the iterative-improvement viewpoint, but learns the schedule edit policy from data and reinforcement feedback.

B.2 Learning-Based Construction and Improvement

Neural combinatorial optimization learns reusable heuristics across problem instances. Constructive methods generate a full solution from the input instance, including Pointer Networks (Vinyals et al., 2015), reinforcement-learning based neural combinatorial optimization (Bello et al., 2017), and attention-based solvers such as AM (Kool et al., 2019a) and POMO (Kwon et al., 2020). Predictive methods output edge scores or heatmaps that are decoded into feasible solutions by search procedures (Joshi et al., 2019). These methods demonstrate transferable neural decision making, but they primarily target one-shot solution construction or prediction rather than stepwise refinement of an available feasible schedule.

Learning-to-improve methods are closer to our setting because they operate on a current solution. Representative work learns to select improvement heuristics (Wu et al., 2021), parameterize neural k-opt exchanges (Ma et al., 2023), perform neural large neighborhood search (Hottung & Tierney, 2020), or guide neighborhood search for constrained aerial scheduling (Guo et al., 2025). These approaches combine learning with iterative search, but many define the learned action through a prescribed neighborhood, operator family, exchange schema, or destroy-repair template. The policy therefore selects or parameterizes a search move, rather than directly modeling the structured edit itself. LIFT instead treats the edit as the action: it samples ordered ADD/DEL edge sets on a schedule-dependent candidate graph and realizes them as feasible successor schedules.

LLM-based optimization methods automate heuristic design by prompting language models to generate, revise, or select heuristic rules and solver components (Zhang et al., 2026; Liu et al., 2026; Ye et al., 2024; Liu et al., 2024). This provides flexibility for complex constrained problems, but the resulting solver is typically non-differentiable and sensitive to prompts, interfaces, and feasibility-

handling details. LIFT instead learns an explicit stochastic policy over schedule edits, so the sampled action has a tractable probability and can be optimized by imitation learning and PPO.

B.3 Diffusion Models and RL Fine-Tuning

Diffusion models have been adapted to combinatorial optimization by treating denoising as structured generation. DIFUSCO (Sun & Yang, 2023) uses graph-based diffusion for discrete combinatorial structures, T2T (Li et al., 2023) connects distribution learning during training with gradient-based search at test time, and DiGress (Vignac et al., 2023) develops discrete denoising diffusion for graph generation. GenSCO (Li et al., 2025) further uses generation as a search operator through disruption and enhancement. These methods show that diffusion is effective for structured optimization, but they mainly operate at the solution level: the model constructs or regenerates candidate solutions. LIFT uses diffusion at the edit level, as a stochastic policy that repeatedly acts on the current feasible schedule.

RL fine-tuning of diffusion models has also been studied outside combinatorial optimization. DDPO (Black et al., 2024) formulates reverse denoising as a multi-step decision process for fine-tuning text-to-image diffusion generators, and DPOK (Fan et al., 2023) applies KL-regularized policy optimization to the same setting. DPPO (Ren et al., 2025) extends this line to diffusion policies in continuous control, fine-tuning multi-step diffusion-based policies for robot learning with PPO. These works show that diffusion generators and diffusion policies can be optimized with downstream rewards via policy-gradient methods. However, PPO-style updates require re-evaluating sampled-action probabilities (Schulman et al., 2017). For standard multi-step diffusion, this requires accounting for the intermediate reverse-chain states when evaluating the final action probability (Ho et al., 2020; Austin et al., 2021; Black et al., 2024). LIFT adopts a tractable one-step interface for combinatorial refinement: a one-step discrete denoiser predicts edit probabilities from the current schedule, a sampled noisy edit, and a noise level, while ordered ADD/DEL subsets provide a Plackett-Luce action likelihood for PPO updates.

C Problem Formulation

We consider a load-dependent UAV scheduling problem. Each drone is assigned to one depot and performs at most one closed route, starting from and returning to the same depot. A feasible schedule specifies the set of active drones, assigns each customer to exactly one drone, and determines the visiting order along each route. The objective is to minimize the sum of customer arrival times, subject to payload and battery constraints induced by a load-dependent energy consumption model. Figure 1 illustrates the schedule structure, where depot-anchored drone routes jointly cover all customers while respecting route-level feasibility.

Sets and indices. Let

$C = \{c_1, \ldots, c_N\}$	set of customers,
$D = \{d_1, \ldots, d_M\}$	set of depots,
$V = C \cup D$	set of all nodes,
K	set of drones (UAVs).

Parameters. Let

q_j	demand (weight) of customer $j \in C$,
N_i	maximum number of drones available at depot $i \in D$,
Q	identical maximum payload capacity for all drones,
B	identical battery capacity (maximum energy) for all drones,
$\bar{w}$	curb (empty) weight of a drone,
L_{ij}	distance from node i to node j, $i, j \in V$,
$v(\cdot)$	flight speed, a non-increasing function of the carried payload,
α	energy coefficient for vertical flight (take-off/landing),

β	energy coefficient for horizontal flight,
H	fixed flight altitude for all drones,
$\mathcal{G}$	sufficiently large constant (Big-M).

Decision variable. The core discrete routing decision is captured by

$$x_{ij}^k \in \{0,1\}, \qquad x_{ij}^k = 1 \;\Leftrightarrow\; \text{drone } k \text{ traverses route } i \to j. \tag{12}$$

We introduce arrival-time variable $t_i^k \geq 0$ for all nodes $i \in V$. Payload and energy feasibility are captured by auxiliary continuous arc variables, e.g., $w_{ij}^k \geq 0$ (payload on (i,j)) and $e_{ij}^k \geq 0$ (remaining energy when departing i toward j). Once x is fixed, the states (t, w, e) are determined by it. The payload and energy relations are linear in these variables, whereas the travel time depends on the carried payload through $v(\cdot)$ and is therefore nonlinear, which is why exact approaches discretize or piecewise-linearize the load-dependent speed. The states are coupled to x through gating constraints such as $w_{ij}^k \leq Q x_{ij}^k$ and $e_{ij}^k \leq B x_{ij}^k$, so that if an arc is not traversed ($x_{ij}^k = 0$), the corresponding auxiliary variables are forced to zero.

Objective function. We minimize the total arrival time across all customers:

$$\min \sum_{k \in K} \sum_{i \in C} t_i^k. \tag{13}$$

This value is governed by the selected route x_{ij}^k through time continuity constraints along traveled flight paths. We minimize the sum of customer arrival time subject to routing, payload, and energy feasibility detailed in the subsequent constraints.

Load-dependent speed and travel time. A heavier drone flies more slowly, so the speed on each route segment depends on the payload carried along it, and the travel time follows from that speed:

$$v_{ij}^k = v\big(w_{ij}^k\big), \qquad t_{ij}^k = \frac{L_{ij}}{v_{ij}^k}. \tag{14}$$

The payload remains constant while a drone traverses an edge, so v_{ij}^k and t_{ij}^k are determined by the carried payload rather than fixed instance parameters. In our experiments, $v(\cdot)$ is linearly interpolated between the empty-load and full-load cruise speeds in Table 3, $v(w) = v_{\text{empty}} - (v_{\text{empty}} - v_{\text{full}})\, w/Q$, and all compared methods use the same load-dependent evaluation.

Energy Consumption Definition. The amount of energy used by drone k moving from i to j with payload w_{ij}^k is given by:

$$b_{ij}^k = \alpha H\big(\bar{w}\, x_{ij}^k + w_{ij}^k\big) + \beta L_{ij}\big(\bar{w}\, x_{ij}^k + w_{ij}^k\big). \tag{15}$$

The curb weight is gated by x_{ij}^k so that $b_{ij}^k = 0$ on untraversed arcs, consistent with $w_{ij}^k = e_{ij}^k = 0$ there.

Routing and flow conservation. Each customer must be served exactly once:

$$\sum_{k \in K} \sum_{i \in V \setminus \{j\}} x_{ij}^k = 1, \quad \forall j \in C. \tag{16}$$

Flow conservation holds at each customer for each drone:

$$\sum_{j \in V \setminus \{i\}} x_{ji}^k = \sum_{j \in V \setminus \{i\}} x_{ij}^k, \quad \forall i \in C,\ \forall k \in K. \tag{17}$$

Each drone must start from and return to the same depot, and is allowed to perform at most one closed trip.

$$\sum_{j \in C} x_{ij}^k = \sum_{j \in C} x_{ji}^k, \quad \forall i \in D,\ \forall k \in K. \tag{18}$$

$$\sum_{i \in D} \sum_{j \in C} x_{ij}^k \leq 1, \quad \forall k \in K. \tag{19}$$

Drones never fly directly between depots:

$$x_{ij}^k = 0, \quad \forall i, j \in D,\ i \neq j,\ \forall k \in K. \tag{20}$$

The number of drones departing from each depot is limited by the available fleet size:

$$\sum_{k \in K} \sum_{j \in C} x_{ij}^k \leq N_i, \quad \forall i \in D. \tag{21}$$

Payload constraints. The total demand served by each drone cannot exceed its capacity:

$$\sum_{j \in C} \sum_{i \in V \setminus \{j\}} x_{ij}^k q_j \leq Q, \quad \forall k \in K. \tag{22}$$

The flight path payload variables follow the flow conservation and capacity constraints. For each customer $i \in C$ and drone $k \in K$:

$$\sum_{j \in V \setminus \{i\}} w_{ji}^k - \sum_{j \in V \setminus \{i\}} w_{ij}^k = q_i \sum_{j \in V \setminus \{i\}} x_{ji}^k, \quad \forall i \in C,\ \forall k \in K. \tag{23}$$

And for every flight path (i, j),

$$0 \leq w_{ij}^k \leq Q x_{ij}^k, \quad \forall i, j \in V,\ i \neq j,\ \forall k \in K. \tag{24}$$

A drone returns to its depot empty:

$$w_{ij}^k = 0, \quad \forall i \in C,\ \forall j \in D,\ \forall k \in K. \tag{25}$$

Together, these constraints ensure that the payload dynamically evolves along the flight path: the weight is carried strictly on selected route segments and decreases by exactly q_i after servicing customer i, from the total demand of the route to zero on returning to the depot, while never exceeding the maximum capacity Q throughout the mission.

Battery constraints. Battery variables are bounded by the maximum energy:

$$0 \leq e_{ij}^k \leq B x_{ij}^k, \quad \forall i, j \in V,\ i \neq j,\ \forall k \in K. \tag{26}$$

The energy balance at customers is

$$\sum_{j \in V \setminus \{i\}} e_{ji}^k - \sum_{j \in V \setminus \{i\}} e_{ij}^k = \sum_{j \in V \setminus \{i\}} b_{ji}^k, \quad \forall i \in C,\ \forall k \in K. \tag{27}$$

When a drone departs from a depot, its battery is fully charged, while the battery decreases monotonically after serving each customer.

$$e_{ij}^k = B x_{ij}^k, \quad \forall i \in D,\ \forall j \in V \setminus \{i\},\ \forall k \in K. \tag{28}$$

Every traversed flight path must be energetically feasible:

$$e_{ij}^k \geq b_{ij}^k, \quad \forall i, j \in V,\ i \neq j,\ \forall k \in K. \tag{29}$$

Time continuity. We define arrival time for all nodes. The time at depots is set to zero:

$$t_i^k = 0, \quad \forall i \in D,\ \forall k \in K. \tag{30}$$

Time continuity is enforced along each flight path using a Big-M linearization, with the load-dependent travel time of equation 14:

$$t_j^k \geq t_i^k + t_{ij}^k - \mathcal{G}(1 - x_{ij}^k),\ \forall i \in V,\ \forall j \in C,\ i \neq j,\ \forall k \in K. \tag{31}$$

Here, the inequality allows for potential waiting time at nodes, although the minimization objective effectively tightens these constraints to equality in the optimal solution. When $x_{ij}^k = 1$, the constraint enforces the intended time propagation along arc (i, j). When $x_{ij}^k = 0$, the inequality becomes $t_j^k \geq t_i^k + t_{ij}^k - \mathcal{G}$, which is automatically satisfied for a sufficiently large $\mathcal{G}$ (given $t_j^k \geq 0$), and thus imposes no time relation between i and j unless the arc is selected. Together with $t_i^k \geq 0$, this conditional propagation also rules out customer-only subtours (cycles) since time would have to strictly increase along every selected arc.

Variable domains. Finally, the domains of decision variables are summarized as

$$x_{ij}^k \in \{0,1\}, \quad \forall i,j \in V,\ i \neq j,\ \forall k \in K, \tag{32}$$

$$t_i^k \geq 0,\ \forall i \in V,\ \forall k \in K, \quad w_{ij}^k \geq 0,\ e_{ij}^k \geq 0, \quad \forall i,j \in V,\ i \neq j,\ \forall k \in K. \tag{33}$$

Together, all these restrictions and definitions impose a load-dependent scheduling model. Since the travel time in equation 14 is nonlinear in the carried payload, the resulting formulation is a mixed-integer nonlinear program unless the load-dependent speed is discretized or piecewise-linearized.

D Algorithmic Procedures

D.1 Regret-Based Initialization

Algorithm 1 constructs the starting feasible schedule s_0 used for imitation-trajectory processing, PPO rollouts, and inference. The algorithm maintains a route-list representation of a schedule. Each route is depot-anchored and corresponds to one available drone route slot. At each step, the initializer evaluates feasible tail insertions and selects the customer whose best and second-best insertion costs differ the most. This regret score prioritizes customers whose best feasible placement is clearly better than the alternatives; a customer with a single feasible insertion receives zero regret, and the initializer never failed in our experiments.

Algorithm 1 Regret-Based Initialization

Input: UAV scheduling instance with depots D, customers C, drones K, and feasibility parameters
Output: Starting feasible schedule s_0

Create an empty depot-anchored route slot for each available drone, respecting depot dispatch limits
Initialize the unserved customer set $\mathcal{U} \leftarrow C$
while $\mathcal{U} \neq \emptyset$ **do**
    Initialize an empty list of feasible insertion records
    for each customer $c \in \mathcal{U}$ **do**
        Evaluate all feasible tail insertions of c into the current route slots
        Each feasible insertion must satisfy payload and battery constraints after appending c
        Let $\Delta(c, r)$ be the increase in total arrival time if c is appended to route r
        if customer c has at least one feasible tail insertion **then**
            Let $\Delta_1(c)$ be the smallest feasible insertion cost
            if customer c has at least two feasible tail insertions **then**
                Let $\Delta_2(c)$ be the second-smallest feasible insertion cost
            else
                Set $\Delta_2(c) \leftarrow \Delta_1(c)$
            end if
            Set regret score $\upsilon(c) \leftarrow \Delta_2(c) - \Delta_1(c)$
            Store the best route $r_c^\star \leftarrow \arg\min_r \Delta(c, r)$
        end if
    end for
    if no customer in $\mathcal{U}$ has a feasible tail insertion **then**
        fail
    end if
    Select $c^\star \in \mathcal{U}$ with the largest regret score $\upsilon(c)$
    Break ties by selecting the customer with the smaller best insertion cost $\Delta_1(c)$
    Append $c^\star$ to its best feasible route $r_{c^\star}^\star$
    Update the affected route attributes, including payload, battery, and arrival times
    Remove $c^\star$ from $\mathcal{U}$
end while
Convert the constructed route list into a feasible schedule s_0
return s_0

D.2 Candidate Graph Construction

Algorithm 2 summarizes the construction of the schedule-dependent candidate graph used in Section 3.1. The construction first generates sparse local alternatives around the current schedule and then applies a lightweight validity filter. This filter removes only locally inadmissible arcs and is not a proof of full route feasibility; complete feasibility is enforced later by the feasibility-preserving decoder. Current schedule edges are added after filtering to ensure that all existing route arcs remain available as DEL candidates.

In implementation, E^{knn} is directed and customer-only: each customer connects to its k nearest customers. E^{2hop} adds a bounded neighbors-of-neighbors expansion, again retaining at most k closest two-hop candidates per customer. E^{dep} contains all depot-to-customer and customer-to-depot arcs. We set $k = 64$. The validity filter removes self-loops, depot-depot arcs, demand-infeasible arcs, customer-customer arcs whose endpoint demands exceed capacity, and arcs whose single-edge energy exceeds the battery limit.

Algorithm 2 Schedule-Dependent Candidate Graph Construction

Input: Instance with depots D, customers C, current schedule graph G^{cur}, neighborhood size k
Output: Candidate graph G^{cand}, node features $\{\mathbf{f}_i\}_{i\in V}$, edge features $\{\mathbf{f}_e\}_{e=(i,j)\in E^{\mathrm{cand}}}$

$V \leftarrow D \cup C$
$E^{\mathrm{knn}} \leftarrow \{(i,j) : i,j \in C,\ j \text{ is among the } k \text{ nearest customers of } i\}$
$E^{\mathrm{2hop}} \leftarrow$ bounded two-hop customer-customer arcs, retaining at most k closest candidates per customer
$E^{\mathrm{dep}} \leftarrow \{(d,c),(c,d) : d \in D,\ c \in C\}$
$\widetilde{E} \leftarrow E^{\mathrm{knn}} \cup E^{\mathrm{2hop}} \cup E^{\mathrm{dep}}$
Remove from $\widetilde{E}$ all self-loops and depot-depot arcs
Remove from $\widetilde{E}$ arcs failing local demand/load checks
Remove from $\widetilde{E}$ arcs whose single-edge energy exceeds battery capacity
$E^{\mathrm{cand}} \leftarrow \widetilde{E} \cup E^{\mathrm{cur}}$
for each node $i \in V$ **do**
    $\mathbf{f}_i \leftarrow [\tilde{x}_i, \tilde{y}_i, q_i, \mathbb{I}_{i\in D}]$
end for
for each edge $e = (i,j) \in E^{\mathrm{cand}}$ **do**
    $b_e \leftarrow \mathbb{I}_{e\in E^{\mathrm{cur}}}$
    $d_e \leftarrow$ Euclidean distance between nodes i and j
    $\mathbf{f}_e \leftarrow [b_e, d_e]$
end for
$G^{\mathrm{cand}} \leftarrow (V, E^{\mathrm{cand}})$
return G^{cand}, $\{\mathbf{f}_i\}_{i\in V}$, $\{\mathbf{f}_e\}_{e=(i,j)\in E^{\mathrm{cand}}}$

D.3 Sparse Prior Construction for One-Step Diffusion

For one-step diffusion pretraining, we use a sparse prior rather than a teacher-conditioned forward corruption process. For each training sample, we first draw a discrete timestep $u \sim \mathrm{Unif}\{u_{\min}, \ldots, u_{\max}\}$. The sampled timestep is used as a conditioning variable through its embedding, but we do not simulate a multi-step forward diffusion chain. Instead, we directly construct a sparse noisy edit vector and train the denoiser to recover the clean expert edit from this one-step perturbation. This design avoids teacher-information leakage and keeps the training distribution aligned with the inputs used later during PPO fine-tuning.

For $d \in \{\mathrm{ADD}, \mathrm{DEL}\}$, let E^d denote the corresponding candidate-edge set in the current candidate graph. We construct a binary noisy edit vector $\tilde{a}_u \in \{0,1\}^{|E^{\mathrm{cand}}|}$ by sampling edges uniformly without replacement within each domain. Concretely, for each domain d, we sample $k_d = \min(K_d, |E^d|)$ edges from E^d, where K_d is the edit budget defined in the main policy. The entries of $\tilde{a}_u$ corresponding to the sampled edges are set to 1, and all remaining entries are set to 0. Hence, $\tilde{a}_u$ is sparse by construction, and its support size in each domain is consistent with the action budget used by the decoder.

The resulting vector $\tilde{a}_u$ is then encoded as a one-hot edge indicator and concatenated with the candidate-edge features and the timestep embedding of u. Conditioned on $(s_t, \tilde{a}_u, u)$, the denoiser predicts the clean edge-edit labels. In this way, the one-step diffusion objective can be viewed as learning to denoise a randomly generated, budget-compatible sparse edit pattern into the expert edit that should be executed by the decoder.

D.4 Feasibility-Preserving Decoder

The decoder realizes the transition from s_t to s_{t+1}. Given the sampled action $a_t = (a_t^{\mathrm{ADD}}, a_t^{\mathrm{DEL}})$, it does not directly toggle selected edges, since direct edge changes may break route continuity, depot anchoring, customer uniqueness, payload feasibility, or battery feasibility. Instead, the j-th edge of an ordered list of length k_d receives weight $(k_d - j + 1)/k_d$; a customer's DEL score is the sum of the weights of its incident DEL edges, and the ADD support of inserting c between i and j is the sum of the weights of (i, c) and (c, j), with unselected edges counting as zero. Algorithm 3 summarizes the procedure.

Algorithm 3 Feasibility-Preserving Decoder

Input: Current feasible schedule s_t; sampled action $a_t = (a_t^{\mathrm{ADD}}, a_t^{\mathrm{DEL}})$; relocation limit; rerank width M; optional Tabu list
Output: Feasible successor schedule s_{t+1}, and relocated customers for Tabu update

Convert the ordered ADD and DEL lists into rank-weighted edge preferences
Aggregate DEL preferences into customer scores, ignoring depot endpoints and Tabu-blocked customers
Rank customers by aggregated DEL score and keep the top candidates within the relocation limit
Initialize the best decoded schedule as NULL
for each candidate subset of top-ranked customers, of increasing size up to the relocation limit **do**
    Copy s_t and remove the selected customers
    for each removed customer c in order **do**
        Enumerate feasible insertion positions for c in the current routes
        Feasibility checks include route continuity, depot anchoring, payload capacity, and battery constraints
        if no feasible insertion position exists in existing routes **then**
            Try to create a new feasible depot-c-depot route
        end if
        if customer c cannot be placed **then**
            Discard this decoded schedule and continue to the next subset
        end if
        Keep the top M feasible positions by local insertion cost
        Insert c at the retained position with the strongest ADD-list support
        Break ties by smaller local insertion cost
        Recompute route feasibility and cost after insertion
    end for
    Recompute the feasibility and objective value of the decoded schedule
    if the decoded schedule is feasible and has the smallest $J(\cdot)$ so far **then**
        Store it as the best decoded schedule
    end if
end for
if no feasible decoded schedule was found **then**
    return s_t and an empty relocated-customer list
else
    return the best decoded schedule as s_{t+1}, together with its relocated customers
end if

Thus, the decoder uses DEL edits to decide which customers to reconsider and ADD edits to decide where they should be reinserted. Every accepted output is a feasible UAV schedule. If all attempted reassignments fail, the edit is rejected and the state remains $s_{t+1} = s_t$. The relocated-customer list is used only by the Tabu mechanism in Appendix D.6.

The decoder combines both signals because they serve different purposes. Local insertion cost identifies feasible positions with a small immediate cost increase, but it does not capture downstream load-dependent effects. The learned ADD scores indicate which positions are more promising for improving the overall schedule, but they do not guarantee feasibility. The cost-only ablation in Appendix F.3 confirms that both are needed: replacing ADD-based selection with cost-only greedy insertion worsens performance by 3.76% on average.

D.5 PPO Fine-Tuning Procedure

Algorithm 4 summarizes the PPO fine-tuning stage. The actor is initialized from the imitation-pretrained diffusion policy. During rollouts, the policy stores the sampled noisy edit mask and noise level $(\tilde{a}_{u_m}, u_m)$ together with the sampled action a_m. During PPO updates, the same $(\tilde{a}_{u_m}, u_m)$ is replayed so that the conditional log-probability of a_m can be recomputed under the updated actor.

The critic $V_\phi(s)$ is an independent state-value network defined on the same candidate graph G^{cand} as the actor. It encodes node features and edge features with separate lightweight MLPs, pools them into graph-level summaries through attention pooling, and concatenates these summaries with a small set of rollout descriptors capturing action-budget and episode-progress information. The resulting vector is passed through an MLP value head to produce $V_\phi(s) \in \mathbb{R}$. The current schedule is represented implicitly through the candidate graph itself and the edge-membership indicator $b_e = \mathbb{I}_{e \in E^{\mathrm{cur}}}$. The critic is used only during training as a baseline for advantage estimation and is not used at inference time.

D.6 Multi-Branch Inference Search

Algorithm 5 gives the inference-time search wrapper. Each branch repeatedly applies the learned one-step diffusion policy and the feasibility-preserving decoder, keeping the lowest-cost feasible successor among its sampled actions and accepting it only when it improves the current schedule. Different branches use different sampling temperatures, with $\tau_y = 0$ denoting deterministic top-k selection, to encourage diverse refinement trajectories. Each branch maintains its own best schedule, Tabu customer set, and stagnation counter; relocated customers stay Tabu for 10 steps, and the set is cleared on shake. When a branch stagnates, it is restarted from a perturbed version of its best-so-far schedule, and it terminates when n_{shake} consecutive shakes bring no new best schedule. Non-improving successors are accepted during PPO rollouts so that multi-step credit assignment is learned, whereas inference accepts only improvements and relies on branch diversity and shake for exploration.

The Shake move in Algorithm 5 is used only when a branch stops improving. Following the VNS-style destroy-and-repair strategy (Mladenović & Hansen, 1997; Guo et al., 2025; Li et al., 2025), it perturbs the branch-best feasible schedule by randomly applying one of the neighborhood operators listed in Table 2. The perturbed schedule is then passed through the same feasibility checks used by the feasibility-preserving decoder. This restart mechanism improves exploration while preserving the invariant that every branch state remains a feasible UAV schedule.

Table 2: Operators used for solution perturbation. Intraroute operators modify a single route; interroute operators exchange structure across two routes.

Class	Operator
Intraroute	*relocate*: move m consecutive customers to another position ($m \in \{1, 2, 3\}$)
	2-opt: reverse a segment by replacing two edges
	or-opt: reinsert a segment of up to three customers
	insertion: insert an unserved customer into an arc
Interroute	*relocate*: transfer m consecutive customers to another route ($m \in \{1, 2\}$)
	2-opt: reconnect two edges across different routes
	or-opt: move a segment of up to three customers to another route
	cross-exchange: swap two segments between routes
	node-arc exchange: swap a customer with an arc in another route

Algorithm 4 PPO Fine-Tuning of LIFT

Input: Imitation-pretrained actor π_θ; critic V_ϕ; synthetic training instances; rollout horizon; number of PPO update epochs; clipping parameter ϵ; entropy coefficient λ
Output: PPO-fine-tuned actor π_θ

for each PPO iteration **do**
  Initialize an empty rollout buffer
  for each sampled training instance **do**
    Construct the starting feasible schedule s_0 using Algorithm 1
    for rollout step $m = 0, 1, \ldots$ **do**
      Construct the candidate graph G^{cand} from the current schedule s_m
      Sample a noise level $u_m \sim p(u)$
      Sample a noisy edit mask $\tilde{a}_{u_m} \sim m_{u_m}(\cdot \mid G^{\text{cand}})$
      Use π_θ with input $(s_m, \tilde{a}_{u_m}, u_m)$ to compute ADD and DEL edge weights
      Sample an ordered action $a_m = (a_m^{\text{ADD}}, a_m^{\text{DEL}})$ using Gumbel-TopK
      Compute and store the rollout log-probability $\log \pi_\theta(a_m \mid s_m, \tilde{a}_{u_m}, u_m)$
      Compute and store the value estimate $V_\phi(s_m)$
      Decode a_m with Algorithm 3 to obtain a feasible successor schedule s_{m+1}
      Compute reward $r_m = \hat{J}(s_m) - \hat{J}(s_{m+1})$
      Store s_m, a_m, $(\tilde{a}_{u_m}, u_m)$, r_m, the rollout log-probability, and $V_\phi(s_m)$ in the buffer
      Set $s_m \leftarrow s_{m+1}$
    end for
  end for
  Compute return targets and GAE advantages from the collected rewards and value estimates
  Normalize advantages within the rollout buffer
  for several critic update epochs **do**
    Update V_ϕ by minimizing the squared error between $V_\phi(s_m)$ and the return target
  end for
  for several actor update epochs **do**
    for each minibatch from the rollout buffer **do**
      Replay the stored $(\tilde{a}_{u_m}, u_m)$ for each stored state s_m
      Recompute $\log \pi_\theta(a_m \mid s_m, \tilde{a}_{u_m}, u_m)$ for the stored action a_m
      Form the PPO likelihood ratio from the recomputed log-probability and the stored rollout log-probability
      Update θ by maximizing the clipped PPO objective in equation 11
    end for
    Stop the actor update early if the policy change exceeds the preset KL threshold
  end for
end for
return π_θ

E Implementation and Reproducibility Details

E.1 Synthetic Instance Generation

All policy training instances are synthetic. We organize the benchmark-derived templates into 13 classes according to $(|C|, |D|)$ and, for each class, extract the fleet configuration, coordinate range, empirical demand statistics, and depot-customer geometric templates.

Rather than sampling customers uniformly in a bounding box, we generate synthetic instances from real benchmark templates. For each target class, we sample one real template and apply a random similarity transform to its depot layout (scale $\in [0.92, 1.08]$, rotation $\in [-12^\circ, 12^\circ]$, bounded translation). Customers are then resampled around the transformed depots using the template's depot-relative cluster structure, with isotropic Gaussian jitter of standard deviation $\max(0.15\,\sigma_{\text{cluster}},\, 0.5)$ added for diversity. Customer demands and depot-specific routing parameters are inherited from the selected template. A candidate is retained only if total demand fits total fleet capacity, every single-customer demand is within per-drone payload, and every customer admits a feasible depot round-trip under the battery budget and load-dependent energy model. Otherwise it is rejected and resampled.

Algorithm 5 Multi-Branch Inference Search

Input: Trained actor π_θ; initial feasible schedule s_0; number of branches Y; branch temperatures $\{\tau_y\}_{y=1}^{Y}$; sampled actions per step n_c; maximum refinement steps; stagnation threshold; shake patience n_{shake}; decoder parameters
Output: Best feasible schedule found by LIFT

for each branch $y = 1, \ldots, Y$ in parallel **do**
  Set current schedule $s_y^{\text{cur}} \leftarrow s_0$
  Set branch-best schedule $s_y^{\text{best}} \leftarrow s_0$
  Initialize Tabu customer set $\mathcal{T}_y \leftarrow \emptyset$
  Initialize stagnation counter $\text{stag}_y \leftarrow 0$ and unproductive-shake counter $\text{ns}_y \leftarrow 0$
end for
for each refinement step, while at least one branch is active **do**
  for each active branch $y = 1, \ldots, Y$ in parallel **do**
    Construct the candidate graph G^{cand} from s_y^{cur}
    for $i = 1, \ldots, n_c$ **do**
      Sample a noise level $u \sim p(u)$ and a noisy edit mask $\tilde{a}_u \sim m_u(\cdot \mid G^{\text{cand}})$
      Use π_θ with branch temperature τ_y to compute ADD and DEL edge weights
      Sample an ordered action $a = (a^{\text{ADD}}, a^{\text{DEL}})$ using Gumbel-TopK
      Decode a with Algorithm 3, using Tabu set $\mathcal{T}_y$
    end for
    Let s_y^{next} be the lowest-cost feasible decoded schedule (if any) and $\mathcal{M}_y$ its relocated customers
    if $J(s_y^{\text{next}}) < J(s_y^{\text{cur}})$ **then**
      Set $s_y^{\text{cur}} \leftarrow s_y^{\text{next}}$
      Update the Tabu set $\mathcal{T}_y$ using relocated customers $\mathcal{M}_y$
    end if
    if $J(s_y^{\text{cur}}) < J(s_y^{\text{best}})$ **then**
      Set $s_y^{\text{best}} \leftarrow s_y^{\text{cur}}$
      Set $\text{stag}_y \leftarrow 0$ and $\text{ns}_y \leftarrow 0$
    else
      Set $\text{stag}_y \leftarrow \text{stag}_y + 1$
    end if
    if stag_y reaches the stagnation threshold **then**
      Perturb s_y^{best} with a SHAKE move to obtain a feasible schedule $\tilde{s}_y$
      Set $s_y^{\text{cur}} \leftarrow \tilde{s}_y$
      Clear the Tabu set $\mathcal{T}_y$
      Set $\text{stag}_y \leftarrow 0$ and $\text{ns}_y \leftarrow \text{ns}_y + 1$
      if ns_y reaches the shake patience n_{shake} **then**
        Deactivate branch y
      end if
    end if
  end for
end for
return $\arg\min_{y=1,\ldots,Y} J(s_y^{\text{best}})$

The experiments in this paper use 7 synthetic training classes, from $C50_D4$ to $C160_D4$, covering $|C| \in [50, 160]$. We generate 50 synthetic instances per class, yielding 350 synthetic instances in total. Within each class, the instances indexed $\{10, 40\}$ are held out for validation, and the remaining instances are used for training. The held-out synthetic instances are used only for model selection and training diagnostics, not for the final benchmark comparison. All public benchmark instances are kept separate from synthetic training and validation, so the reported test results evaluate transfer from template-based synthetic training to the original benchmark configurations. No benchmark instance or solution is used for training. Because the synthetic instances preserve problem-level structure from the same benchmark family, the nine configurations with $|C| \in [50, 160]$ demonstrate transfer to held-out benchmark instances under matched class statistics, rather than generalization to a new distribution. In contrast, the six larger configurations with $|C| \in [240, 360]$ are never used to

generate training data, and therefore provide a stricter test of cross-scale generalization within the same benchmark family (Section 4.4). For imitation pretraining, GENIS is run once on each of the 350 synthetic training instances, and every accepted improving move is logged as a state transition, yielding 350 improvement trajectories with 564,523 transitions in total (421 to 3,693 per trajectory), of which 20,511 belong to the validation instances.

E.2 Drone and Benchmark Parameters

Table 3 lists the drone physical parameters and benchmark normalization constants used in all experiments. They instantiate the load-dependent UAV scheduling environment defined in Appendix C, whose constraints are enforced by the route-level feasibility checker used by LIFT and all baselines.

The physical parameters determine both feasibility and arrival time. Payload and battery capacities constrain feasible routes, while the load-dependent speed and energy model make each route segment depend on the remaining payload. As a result, a local change in customer order or assignment can affect downstream travel time, energy consumption, and feasibility. The coordinate and demand ranges specify the normalization used when converting the benchmark instances into the UAV scheduling setting.

Table 3: Drone physical parameters and benchmark normalization used in all experiments.

Symbol	Description	Value
$\bar{w}$	Curb (empty) weight of a drone	6 kg
Q	Maximum payload capacity	3 kg
B	Maximum battery capacity	600 Wh
α	Energy coefficient for vertical flight (take-off/landing)	4.0 Wh/(kg·km)
β	Energy coefficient for horizontal flight	3.5 Wh/(kg·km)
H	Fixed cruise altitude	50 m
v_{full} / v_{empty}	Cruise speeds of a fully loaded / empty drone	32 / 48 km/h
-	Customer coordinate normalization range	$[0, 20]$ km
q_j	Per-customer demand range	$[0, 2.3]$ kg

E.3 Model, Training, and Search Hyperparameters

Table 4 summarizes the main hyperparameters of LIFT used across all experiments. Unless otherwise stated, the same settings are used for all benchmark configurations and ablation variants. The table groups the parameters into four parts: the graph neural backbone, imitation pretraining, PPO fine-tuning, and the hybrid decoder used during inference.

The backbone parameters define the capacity of the one-step diffusion policy and the size of the schedule-dependent candidate graph. The imitation and PPO settings correspond to the two training stages described in Section 3. The hybrid decoder parameters control inference-time search, including the number of parallel decoding branches, their sampling temperatures, and the stopping criterion. These values were kept fixed when comparing LIFT with its ablated variants, so performance differences reflect the removed training component rather than changes in model or search budget.

E.4 Baseline Implementation and Adaptation Details

This appendix describes the implementation of MLG and the adaptation of the classical search baselines to load-dependent UAV scheduling. All baselines are evaluated under the same budget of 8,000 schedule evaluations as LIFT (Appendix F.2).

MLG. MLG is a greedy constructor for the minimum-latency objective in equation 1. The objective follows the minimum latency problem (Silva et al., 2012), whose single-tour heuristics do not apply to depot-anchored routes with payload and battery constraints; Algorithm 6 therefore specifies the constructor in full. At each step, MLG scores every feasible insertion of each unrouted customer, including opening a new route, by its increase in cumulative arrival time, and commits the customer with the largest regret between its best and second-best option, placing customers with a single feasible option first.

Table 4: Main hyperparameters used by LIFT across all experiments.

Module	Description	Value
Backbone	Hidden dimension h	128
	TransformerConv layers / attention heads	3 / 4
	Customer KNN size used to build E^{knn} and E^{2hop}	64
	Softmax temperatures for edge edit $\tau_{\text{ADD}}/\tau_{\text{DEL}}$	$1.0/1.0$
	Edit budget K_d for $d \in \{\text{ADD}, \text{DEL}\}$	$7/7$
Imitation pretraining	Negative undersampling ratio $\rho_{\text{ADD}} = \rho_{\text{DEL}}$	2.0
	Optimizer, base learning rate	AdamW, 3×10^{-4}
	Learning rate schedule	cosine $\rightarrow 3\times10^{-6}$
	Batch size	512
	Total epochs	100
PPO fine-tuning	Episode length	50
	Actor / critic learning rate	1×10^{-4} / 3×10^{-4}
	PPO clip ϵ / KL threshold δ_{KL}	0.1 / 0.05
	Discount γ / GAE coefficient λ_{GAE}	0.99 / 0.95
	Total epochs / minibatch size	200 / 128
Decoder	Number of parallel branches Y	4
	Branch temperatures $\{\tau_y\}_{y=1}^{Y}$	$\{0.0, 0.5, 1.0, 2.0\}$
	Sampled actions per step n_c	4
	Fixed decoding budget M	7
	Maximum inference steps	500
	Stagnation threshold	16
	Shake patience n_{shake}	15

ALNS, VNS, and LKH-3-style search. Standard implementations of ALNS (Ropke & Pisinger, 2006), VNS (Mladenović & Hansen, 1997), and LKH-3 (Helsgaun, 2017) cannot be used directly because route costs and feasibility change with payload. We therefore recompute payload, battery use, and arrival times after every candidate move, while the search logic otherwise remains unchanged. ALNS uses adaptive destroy-and-repair, VNS switches among the neighborhoods, and the LKH-3-style baseline applies sequential edge exchanges with feasibility repair.

Learning-based baselines. GENIS, GenSCO, and AFL use their official implementations with the original hyperparameters. GENIS adapts its operator selection online on each evaluated instance, GenSCO is trained on the same 350 synthetic instances as LIFT, and AFL queries GPT-4.1 through the OpenAI API as in the original work.

F Supplementary Experimental Results

F.1 Supplementary Results for the Main Benchmarks

This subsection provides supplementary results for the main benchmark evaluation. We first summarize the instance-wise margins of LIFT over the strongest competitor in total arrival time (Figure 4), and then report makespan as a diagnostic of tail-completion behavior.

Figure 4 shows, for every benchmark configuration, the relative improvement of LIFT over the best competing method, computed as $100\% \times (J_{\text{best competitor}} - J_{\text{LIFT}})/J_{\text{best competitor}}$. LIFT improves over the best competitor on every configuration, by 3.47% on average and up to 10.33%, and all differences are statistically significant under the Wilcoxon rank-sum test with $\alpha = 0.05$ (Table 1). GENIS is the closest competitor on 14 of the 15 configurations and AFL on the remaining one. On the six configurations beyond the training scale, the improvement over the best competitor averages 4.35%, compared with 2.88% within it.

Since LIFT optimizes the sum of customer arrival times, a natural concern is whether the average-case latency gains come at the cost of a disproportionately late-finishing drone. To examine this, Table 5 reports makespan, the completion time of the latest-finishing drone, as a diagnostic of tail-completion behavior. Makespan is not part of the training objective and is reported only as a diagnostic. LIFT

Algorithm 6 Minimum-Latency Greedy (MLG) Construction

Input: Customer set C; depot set D; fleet and feasibility constraints.
Output: Feasible route set $\mathcal{R}$, or INFEASIBLE.

for each customer $c \in C$ **do**
    evaluate all feasible single-customer routes $d \to c \to d$
    record the depot that serves c best and the cost of its best single-customer route
end for
initialize $\mathcal{R}$ with one initial route per depot when possible
$\mathcal{U} \leftarrow C \setminus \{\text{customers already in } \mathcal{R}\}$
while $\mathcal{U} \neq \emptyset$ **do**
    for each $c \in \mathcal{U}$ **do**
        evaluate all feasible insertions into the current routes
        also consider opening a new route if fleet capacity remains
        if no feasible action exists for c **then return** INFEASIBLE
        end if
        if c has only one feasible action **then**
            mark c as forced
        else
            compute the regret of c as the gap between its best and second-best feasible actions
        end if
    end for
    give highest priority to forced customers; otherwise prefer customers with larger regret, using the single-customer estimates to break ties
    execute the best current action for the selected customer
    remove that customer from $\mathcal{U}$
end while
return $\mathcal{R}$

Table 5: Main results on the 15 public benchmark instances in terms of makespan. Lower values are better. The best and second-best mean values are shown in bold and underlined, respectively.

Instance	MLG	ACO	ALNS	VNS	LKH-3	GENIS	AFL	GenSCO	LIFT
C50_D4_K4	384.42	447.11 (29.01)	381.17 (7.14)	381.11 (0.00)	398.80 (27.78)	380.74 (1.03)	388.59 (16.37)	414.58 (47.78)	**378.97 (0.00)**
C50_D4_K2	459.32	551.07 (24.43)	422.27 (23.66)	423.55 (11.26)	449.08 (56.53)	421.03 (23.63)	438.43 (36.44)	467.81 (45.75)	**413.72 (21.29)**
C75_D5_K3	445.26	489.79 (36.56)	423.66 (25.12)	420.86 (22.07)	444.21 (23.93)	420.74 (15.31)	429.86 (16.45)	476.56 (59.28)	**413.96 (3.20)**
C80_D2_K5	1218.65	1382.39 (129.91)	1127.35 (45.08)	1134.27 (36.12)	1131.08 (29.77)	1130.29 (31.11)	1129.66 (111.30)	1278.20 (173.69)	**1107.90 (46.72)**
C100_D2_K8	579.31	747.55 (138.05)	518.75 (30.60)	517.53 (21.20)	521.21 (45.78)	516.57 (20.71)	607.75 (38.87)	554.81 (52.98)	**507.09 (17.74)**
C100_D2_K5	695.96	824.81 (97.11)	654.39 (48.74)	656.04 (40.77)	658.86 (72.88)	660.20 (56.19)	828.91 (50.29)	768.73 (56.70)	**631.77 (52.54)**
C100_D3_K6	419.78	693.65 (115.15)	400.42 (9.61)	401.55 (13.67)	412.97 (30.21)	403.56 (17.11)	414.14 (59.29)	420.95 (42.73)	**361.86 (6.48)**
C100_D4_K4	470.20	727.38 (60.67)	429.72 (25.51)	432.45 (25.82)	472.63 (48.51)	432.82 (16.64)	493.93 (31.59)	478.90 (28.58)	**425.66 (17.31)**
C160_D4_K5	995.27	1798.79 (191.37)	904.88 (107.22)	924.21 (124.69)	926.56 (163.20)	908.03 (24.26)	932.69 (82.97)	1024.81 (158.48)	**887.12 (59.99)**
C240_D6_K5	1055.66	1856.59 (121.55)	**956.13 (66.20)**	963.07 (110.07)	1025.85 (119.65)	962.80 (87.74)	998.77 (58.87)	1076.46 (55.15)	958.10 (91.37)
C249_D2_K14	1729.27	1968.36 (78.93)	1644.15 (14.55)	1648.26 (20.80)	1701.17 (62.02)	1655.21 (24.09)	1908.95 (122.10)	1737.04 (12.40)	**1577.46 (38.50)**
C249_D3_K12	1312.26	1715.25 (158.10)	1241.09 (41.94)	1254.59 (92.52)	1347.43 (125.89)	1250.32 (99.83)	1426.61 (151.89)	1454.25 (140.74)	**1177.09 (84.06)**
C249_D4_K8	1183.81	1723.92 (230.28)	1125.04 (19.82)	1136.34 (48.69)	1164.23 (126.95)	1133.17 (40.58)	1181.80 (66.80)	1251.54 (261.06)	**1070.80 (61.03)**
C249_D5_K6	1174.13	2005.68 (0.00)	1085.18 (24.30)	1098.34 (55.52)	1230.72 (194.14)	1093.04 (59.70)	1175.10 (234.39)	1244.13 (4.89)	**1051.19 (95.05)**
C360_D9_K5	1153.81	1923.47 (96.43)	1082.52 (24.32)	1107.45 (122.60)	1192.50 (118.34)	1090.46 (96.69)	1120.99 (61.88)	1221.86 (221.24)	**1028.12 (57.26)**

Note. Values are measured in seconds and reported as mean (standard deviation) over 10 runs; MLG is deterministic and is therefore reported without standard deviation. Instance labels `C#_D#_K#` denote the numbers of customers, depots, and drones per depot, respectively.

achieves the best mean makespan on 14 of the 15 configurations and the second-best on the remaining one, where ALNS is 0.2% lower. Thus, the gains in total arrival time do not come at the expense of substantially worse tail completion.

F.2 RUNTIME COMPARISON

For a fair comparison, all methods use the same budget of at most 8,000 schedule evaluations, matching LIFT's default inference budget, which corresponds to $Y = 4$ branches $\times$ 500 refinement steps $\times$ $n_c = 4$ decoder calls per step. For the baselines, one evaluation is likewise one search step that returns a complete feasible schedule: a destroy-and-repair iteration for ALNS, a neighborhood

move for VNS, an edge-exchange move for the LKH-3-style search, one constructed solution per ant for ACO, one operator application for GENIS, one iteration of the generated search procedure for AFL, and one generated solution for GenSCO; MLG is a deterministic constructor and is run once. Every method's step is counted once regardless of the candidates it enumerates internally, such as the insertion positions scanned by an ALNS repair, the neighbors examined by a VNS move, the exchanges evaluated by an LKH-3-style move, or the subsets tried by the LIFT decoder. Under this common budget, LIFT achieves the lowest mean total arrival time on all 15 benchmark configurations. Table 6 additionally reports *time-to-best*, defined as the wall-clock time at which each run first finds its final best schedule, averaged over the same 10 seeds. The final row reports LIFT's average gain over each baseline.

No baseline achieves both a shorter time-to-best and better final quality than LIFT. LIFT reaches its best schedule in 1,556.4 seconds on average. The four faster baselines, MLG, AFL, ACO, and ALNS, produce worse schedules on all 15 configurations, while GENIS, VNS, GenSCO, and LKH-3 take 8.3, 5.3, 3.8, and 2.4 times longer than LIFT, respectively, yet still achieve worse final results. On the largest configuration, C360_D9_K5, GENIS takes about 14 hours and remains worse than the solution LIFT finds in about 1.6 hours. This is consistent with Figure 3, which shows that LIFT reaches near-final quality early in the refinement process. Thus, LIFT's advantage is not caused by a larger search budget or longer runtime, but by finding better schedules more efficiently.

Table 6: Time-to-best on the 15 public benchmark instances under the matched budget of 8,000 schedule evaluations. Values report wall-clock seconds as mean (standard deviation) over 10 seeds; MLG is deterministic and is therefore reported without standard deviation. The final row reports the average gain of LIFT over each baseline, computed as the relative reduction in total arrival time.

Instance	MLG	ACO	ALNS	VNS	LKH-3	GENIS	AFL	GenSCO	LIFT
C50_D4_K4	0.5	236.7 (73.4)	21.1 (5.8)	171.8 (104.6)	64.8 (24.8)	111.3 (82.8)	3.5 (3.9)	242.2 (76.6)	39.0 (12.5)
C50_D4_K2	0.5	323.2 (68.2)	17.2 (6.5)	222.8 (190.7)	77.2 (19.1)	270.6 (168.2)	3.2 (2.6)	426.8 (92.9)	158.6 (41.8)
C75_D5_K3	1.7	500.1 (147.8)	59.9 (12.8)	639.4 (278.1)	23.6 (4.8)	651.0 (197.7)	5.8 (2.6)	853.7 (275.8)	163.0 (49.2)
C80_D2_K5	2.3	516.0 (156.7)	55.6 (17.3)	751.6 (324.2)	306.6 (113.7)	896.6 (392.4)	4.2 (2.5)	866.2 (316.2)	103.1 (61.9)
C100_D2_K8	4.1	1090.3 (178.8)	190.3 (34.9)	1555.1 (573.7)	5470.7 (1755.9)	2139.7 (991.8)	39.9 (20.4)	1897.1 (446.9)	537.0 (165.6)
C100_D2_K5	4.9	607.2 (246.7)	244.7 (44.8)	2839.2 (302.3)	325.1 (69.4)	3921.2 (1028.4)	40.5 (25.6)	2250.7 (403.9)	399.8 (160.0)
C100_D3_K6	4.0	717.8 (301.4)	177.7 (25.5)	1463.5 (346.1)	918.6 (251.1)	1631.6 (840.3)	40.8 (34.9)	1816.6 (389.5)	441.4 (204.8)
C100_D4_K4	4.2	954.2 (229.0)	197.9 (19.3)	2019.4 (356.3)	325.2 (70.0)	1958.6 (758.7)	26.1 (9.6)	1276.7 (254.5)	516.1 (263.2)
C160_D4_K5	17.9	1203.1 (714.5)	286.5 (61.7)	6159.9 (588.3)	236.2 (74.5)	8837.7 (2484.0)	180.9 (90.1)	5428.9 (695.1)	396.3 (102.4)
C240_D6_K5	56.5	2027.5 (716.0)	978.0 (175.4)	12731.3 (2915.5)	283.0 (67.7)	21971.0 (4023.1)	767.4 (183.0)	12451.9 (916.6)	2065.7 (788.6)
C249_D2_K14	71.7	855.2 (343.3)	2301.4 (113.1)	18186.3 (1385.4)	24857.1 (3786.1)	28106.7 (5723.5)	912.0 (108.5)	14598.4 (735.3)	4126.2 (300.6)
C249_D3_K12	67.1	240.0 (65.4)	1890.1 (113.1)	14502.6 (1897.7)	17077.8 (1849.6)	24488.2 (4337.9)	986.2 (186.7)	7766.7 (507.9)	3319.3 (392.4)
C249_D4_K8	70.2	232.6 (65.8)	1888.8 (183.4)	15773.1 (2291.0)	3312.2 (464.6)	25689.4 (3995.1)	1048.8 (141.1)	7666.1 (866.3)	1329.2 (663.8)
C249_D5_K6	70.8	165.2 (55.8)	1906.7 (326.4)	15947.2 (3304.2)	1763.5 (257.5)	22584.7 (4715.9)	1024.6 (162.6)	3375.3 (753.6)	3825.3 (455.3)
C360_D9_K5	187.8	606.0 (125.6)	2812.1 (303.1)	31145.4 (4865.8)	242.8 (62.2)	50761.8 (6869.2)	2269.2 (204.6)	28922.5 (1497.6)	5926.5 (329.5)
Mean	37.6	685.0	868.5	8273.9	3685.6	12934.7	490.2	5989.3	1556.4
Avg. LIFT gain (%)	9.5	41.9	3.6	4.0	7.6	3.5	10.2	13.1	—

F.3 Detailed Ablation Results

This appendix complements the ablation study in Section 4.3 with full variant definitions, per-comparison statistics, and the search-component ablations. Per-instance results for all variants are reported in Table 7.

LIFT (PPO only) vs. full LIFT. Full LIFT outperforms LIFT (PPO only) on 14 of the 15 configurations in total arrival time, tying only on the smallest instance C50_D4_K4. Averaged over all configurations, full LIFT improves over LIFT (PPO only) by 4.7%, and the gap widens on the larger out-of-training-scale configurations, where the improvement averages 6.9% and reaches 13.0% on C360_D9_K5. Under the training budget considered, PPO from random initialization learns a functional but consistently weaker refinement policy than the full two-stage pipeline. Figure 5 shows the PPO training progress on C75_D5_K3: starting from pretrained policies, LIFT and LIFT-NoDiff improve quickly and plateau, whereas LIFT (PPO only) starts far lower and is still improving when the budget ends.

LIFT (imitation only) vs. full LIFT. Full LIFT improves over LIFT (imitation only) on 14 of the 15 configurations, again tying only on C50_D4_K4. The gap is moderate on smaller instances,

Table 7: Ablation results on the 15 public benchmark instances in total customer arrival time (10^2 seconds, mean (standard deviation) over 10 seeds; lower is better). The best and second-best mean values are shown in bold and underlined, respectively.

Instance / Method	LIFT (imitation only)	LIFT (PPO only)	LIFT-NoDiff	Random-Edit	LIFT
C50_D4_K4	**47.76 (0.00)**	**47.76 (0.00)**	**47.76 (0.00)**	**47.76 (0.00)**	**47.76 (0.00)**
C50_D4_K2	71.57 (0.87)	72.86 (1.26)	73.61 (0.38)	72.42 (1.34)	**70.90 (0.53)**
C75_D5_K3	77.88 (0.52)	77.17 (0.44)	77.26 (0.40)	76.96 (0.24)	**76.03 (0.46)**
C80_D2_K5	243.35 (0.26)	243.48 (0.37)	243.36 (0.21)	243.28 (0.29)	**239.56 (0.29)**
C100_D2_K8	150.28 (0.70)	151.52 (1.09)	149.60 (0.57)	150.57 (1.07)	**147.89 (0.89)**
C100_D2_K5	183.58 (1.72)	184.02 (1.72)	182.23 (1.49)	183.93 (1.69)	**175.03 (1.44)**
C100_D3_K6	116.43 (0.59)	117.02 (0.65)	116.56 (0.72)	116.66 (0.58)	**105.34 (0.59)**
C100_D4_K4	131.98 (0.68)	132.62 (0.68)	132.25 (0.60)	133.09 (1.21)	**129.06 (1.00)**
C160_D4_K5	475.75 (0.94)	477.80 (1.43)	475.46 (1.79)	476.19 (1.63)	**465.35 (0.73)**
C240_D6_K5	715.12 (3.30)	721.88 (3.38)	713.24 (1.20)	713.87 (2.60)	**709.55 (3.26)**
C249_D2_K14	1026.41 (7.93)	1040.39 (4.25)	1033.08 (3.81)	1027.59 (5.70)	**971.73 (2.79)**
C249_D3_K12	771.28 (4.73)	772.78 (3.77)	773.19 (1.30)	768.99 (4.19)	**713.32 (3.62)**
C249_D4_K8	755.57 (3.17)	754.50 (4.21)	755.28 (2.33)	756.02 (4.69)	**708.18 (3.28)**
C249_D5_K6	797.60 (3.98)	801.64 (6.44)	796.43 (4.06)	798.84 (5.43)	**749.72 (2.05)**
C360_D9_K5	1129.80 (10.38)	1182.79 (5.04)	1080.99 (4.41)	1076.92 (4.64)	**1028.57 (4.28)**

such as 0.9% on C50_D4_K2, and grows on the out-of-training-scale instances, reaching 5.3% on C249_D2_K14, 7.5% on C249_D3_K12, and 9.0% on C360_D9_K5. Imitation pretraining thus provides a strong edit prior, but expert matching alone does not fully optimize repeated decoded refinements; PPO fine-tuning supplies the missing long-horizon alignment.

LIFT (imitation only) vs. LIFT (PPO only). Excluding the tied smallest instance, LIFT (imitation only) outperforms LIFT (PPO only) on 12 of the remaining 14 configurations, showing that supervised learning from expert improvement trajectories provides a stronger standalone edit prior than direct PPO training from random initialization under the same training budget.

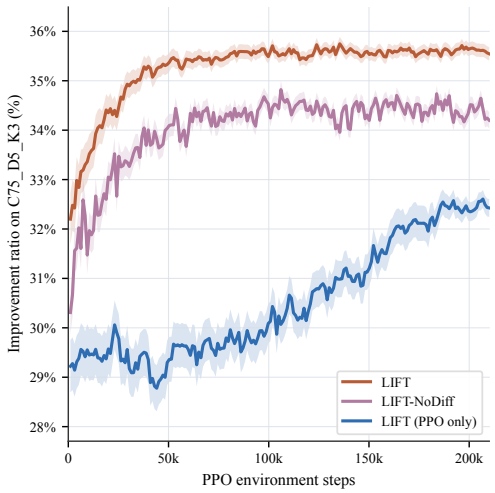

Figure 5: PPO training progress on C75_D5_K3. LIFT and LIFT-NoDiff fine-tune pretrained policies, while LIFT (PPO only) trains from random initialization.

Diffusion conditioning: LIFT-NoDiff and Random-Edit. To separate the contributions of the search procedure, learned edge scoring, and diffusion conditioning, we evaluate two matched controls. **LIFT-NoDiff** is a parameter-matched non-diffusion variant. It retains the same candidate graph, TransformerConv network, ADD/DEL scoring heads, imitation loss, PPO fine-tuning, Plackett-Luce/Gumbel-TopK sampling, critic, decoder, and Tabu/shake search. Diffusion conditioning alone is removed by fixing the noisy-edit input to zero and using the same timestep for every sample, while action sampling remains stochastic through Gumbel-TopK; imitation pretraining thereby becomes standard supervised edge prediction. **Random-Edit** removes the learned policy by replacing its ADD/DEL scores with independent $U(0,1)$ samples for every candidate edge at every refinement step. It uses the same initial schedules, random seeds, decoder, search procedure, and refinement budget as LIFT.

Random-Edit remains competitive with GENIS on some instances, showing that the decoder and multi-branch search already form a useful optimization procedure. Random-Edit also matches or exceeds the single-stage variants on six configurations, so neither stage alone reliably improves on this procedure, whereas the full two-stage policy does so consistently. LIFT-NoDiff has no consistent advantage over Random-Edit, performing better on eight configurations, worse on six, and tying on one. On average, LIFT improves over both ablations by about 3.8%, with gains of up to 9.7%, and its advantage increases from about 2.9% within the training scale to 5.0% over Random-Edit and

5.2% over LIFT-NoDiff on the out-of-training-scale configurations. The one-step diffusion design is complementary: diffusion conditioning improves the edit policy, and the one-step formulation makes the conditional action log-probability in equation 8 directly re-evaluable for PPO.

Search-component ablations. We further measure the marginal contribution of each decoder and search component on all 15 benchmark configurations over 10 seeds, with all other settings unchanged. **Cost-only** removes ADD-edge guidance while keeping the sampled DEL edges unchanged: each removed customer is greedily reinserted at the feasible position with the smallest immediate cost increase, providing a stronger no-ADD baseline than random insertion. **Single-branch** evaluates each branch in Algorithm 5 independently with its own temperature. **No-Tabu** and **No-Shake** disable the corresponding search mechanisms. Table 8 reports the average gain of full LIFT over each ablation. Because full LIFT returns the best result across four branches, the single-branch gaps capture the combined benefit of additional branches and temperatures.

Table 8: Search-component ablations, reporting the average gain of full LIFT over each ablation across the 15 public benchmark instances.

Ablation	LIFT gain (%)
Cost-only insertion (no ADD guidance)	3.76
Single branch ($\tau = 0.0$)	7.38
Single branch ($\tau = 0.5$)	0.72
Single branch ($\tau = 1.0$)	0.93
Single branch ($\tau = 2.0$)	0.88
No-Tabu	3.94
No-Shake	4.22

F.4 Dependence on the Expert Solver

LIFT uses expert improvement trajectories from GENIS only for imitation pretraining, and GENIS is absent at inference. Keeping GENIS as a baseline therefore directly tests whether LIFT merely reproduces its teacher or improves beyond it. To further test whether LIFT depends on GENIS as the expert, we retrain LIFT from scratch using improvement trajectories from ALNS (Ropke & Pisinger, 2006), keeping all other settings unchanged. We denote this variant **LIFT-A**. Table 9 compares LIFT-A and LIFT with their respective expert solvers on the 15 benchmark configurations.

LIFT outperforms LIFT-A by 1.1% on average, with gains of 0.5% within the training scale and 2.1% beyond it, showing that imitating GENIS trajectories yields an additional performance benefit. Crucially, LIFT-A still outperforms both ALNS and GENIS on all 15 configurations despite never using GENIS trajectories. Thus, LIFT's gains do not depend on GENIS-specific imitation: expert trajectories provide the starting point, while LIFT consistently learns to improve beyond its teacher.

F.5 Generalization Beyond Training Scale

Table 10 compares two variants of LIFT that differ only in training data composition: (a) the main 7-class model trained on configurations with $|C| \in [50, 160]$, and (b) a restricted variant trained only on the three smallest classes, C50, C75, and C80. Both share the same architecture, optimization schedule, candidate-graph construction, and inference pipeline, so this comparison isolates the role of training-scale coverage. Results are reported for total arrival time as mean (standard deviation) over 10 random seeds.

Even the restricted variant already outperforms GENIS on 12 of the 15 benchmark configurations in total arrival time, including 5 of the 6 out-of-training-scale configurations. This indicates that LIFT learns local graph-action patterns that generalize across instance sizes without requiring exposure to large training instances. The full 7-class model is nevertheless consistently stronger: it matches or improves on the restricted variant on every configuration in total arrival time, with larger gains on average beyond the training scale. This shows that broader scale coverage further sharpens the policy on instance sizes that the restricted variant never covered.

Table 9: Effect of the expert choice on the 15 public benchmark instances, comparing LIFT-A (trained on ALNS trajectories) with LIFT (trained on GENIS trajectories). Values report total customer arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds. Lower values are better. The best and second-best mean values are shown in bold and underlined, respectively.

Instance	ALNS	GENIS	LIFT-A (ALNS expert)	LIFT (GENIS expert)
C50_D4_K4	47.92 (0.18)	47.86 (0.10)	**47.76 (0.00)**	**47.76 (0.00)**
C50_D4_K2	72.24 (0.85)	72.15 (1.58)	71.04 (0.12)	**70.90 (0.53)**
C75_D5_K3	77.35 (0.47)	77.28 (0.79)	76.26 (0.20)	**76.03 (0.46)**
C80_D2_K5	244.87 (0.94)	244.40 (0.66)	240.52 (0.37)	**239.56 (0.29)**
C100_D2_K8	150.86 (2.21)	150.65 (1.62)	148.63 (0.12)	**147.89 (0.89)**
C100_D2_K5	183.05 (0.82)	182.91 (1.80)	175.91 (2.08)	**175.03 (1.44)**
C100_D3_K6	117.62 (0.58)	117.48 (0.73)	106.39 (0.41)	**105.34 (0.59)**
C100_D4_K4	131.44 (1.16)	131.23 (1.44)	129.71 (0.32)	**129.06 (1.00)**
C160_D4_K5	477.15 (1.92)	476.32 (3.63)	470.00 (0.66)	**465.35 (0.73)**
C240_D6_K5	714.28 (1.94)	713.03 (3.76)	711.68 (4.97)	**709.55 (3.26)**
C249_D2_K14	1021.35 (3.87)	1019.63 (6.22)	993.10 (5.89)	**971.73 (2.79)**
C249_D3_K12	759.24 (1.83)	757.70 (4.37)	733.30 (2.63)	**713.32 (3.62)**
C249_D4_K8	751.13 (2.16)	749.42 (2.64)	728.00 (1.01)	**708.18 (3.28)**
C249_D5_K6	781.46 (3.02)	779.57 (5.12)	764.70 (3.60)	**749.72 (2.05)**
C360_D9_K5	1093.28 (2.87)	1090.94 (4.14)	1059.40 (3.98)	**1028.57 (4.28)**

Table 10: Cross-scale generalization results on the 15 public benchmark instances, comparing the main 7-class model with a restricted variant trained only on C50, C75, and C80. Values report total customer arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds; lower values are better. The best and second-best mean values are shown in bold and underlined, respectively.

Instance / Method	LIFT (trained on C50/C75/C80 only)	GENIS	LIFT
C50_D4_K4	**47.76 (0.00)**	47.86 (0.10)	**47.76 (0.00)**
C50_D4_K2	71.22 (0.65)	72.15 (1.58)	**70.90 (0.53)**
C75_D5_K3	77.05 (0.48)	77.28 (0.79)	**76.03 (0.46)**
C80_D2_K5	241.86 (0.38)	244.40 (0.66)	**239.56 (0.29)**
C100_D2_K8	150.72 (1.46)	150.65 (1.62)	**147.89 (0.89)**
C100_D2_K5	182.61 (1.36)	182.91 (1.80)	**175.03 (1.44)**
C100_D3_K6	115.93 (0.85)	117.48 (0.73)	**105.34 (0.59)**
C100_D4_K4	131.42 (1.57)	131.23 (1.44)	**129.06 (1.00)**
C160_D4_K5	475.81 (1.90)	476.32 (3.63)	**465.35 (0.73)**
C240_D6_K5	710.09 (4.85)	713.03 (3.76)	**709.55 (3.26)**
C249_D2_K14	1013.67 (4.01)	1019.63 (6.22)	**971.73 (2.79)**
C249_D3_K12	771.96 (3.91)	757.70 (4.37)	**713.32 (3.62)**
C249_D4_K8	741.07 (3.69)	749.42 (2.64)	**708.18 (3.28)**
C249_D5_K6	767.09 (3.02)	779.57 (5.12)	**749.72 (2.05)**
C360_D9_K5	1081.95 (4.84)	1090.94 (4.14)	**1028.57 (4.28)**

F.6 Parameter Sensitivity and Robustness Analysis

This appendix analyzes the sensitivity of LIFT to its main inference-time settings: the refinement budget, the shake threshold (the stagnation threshold in Algorithm 5), the candidate-graph sparsity, the decoder insertion budget M, and the branch-wise temperature schedule. Each setting is varied independently while all other settings remain at their defaults in Table 4, and results are reported over 10 random seeds.

Refinement budget. Table 11 varies the refinement budget from 100 to 1,000 steps on the six largest out-of-training-scale configurations, with the shake threshold fixed at 16. Total arrival time decreases consistently on all six configurations as the refinement budget increases. The 500-step default improves over 100 and 200 steps by 8.07% and 6.44% on average, while extending the budget

to 1,000 steps yields a further 0.91% gain. Longer refinement horizons therefore remain beneficial, but the gains diminish beyond 500 steps, and the default budget provides a practical balance between solution quality and inference cost.

Table 11: Sensitivity to the refinement budget on the six largest out-of-training-scale configurations. Values report total arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds; lower is better. The final column reports the average gain of the default 500-step setting over each budget; a negative value indicates that the setting outperforms the default.

Steps	C240_D6_K5	C249_D2_K14	C249_D3_K12	C249_D4_K8	C249_D5_K6	C360_D9_K5	Default gain (%)
100	725.62 (3.70)	1073.68 (11.85)	785.03 (6.81)	778.39 (10.52)	817.45 (7.86)	1145.97 (9.05)	8.07
200	720.68 (4.36)	1049.52 (6.27)	780.11 (7.84)	765.77 (7.25)	809.32 (5.05)	1096.80 (6.51)	6.44
500	709.55 (3.26)	971.73 (2.79)	713.32 (3.62)	708.18 (3.28)	749.72 (2.05)	1028.57 (4.28)	0.00
1000	705.29 (2.65)	961.04 (3.34)	707.61 (2.36)	701.81 (4.56)	744.47 (4.85)	1015.20 (3.99)	−0.91

Shake threshold. Table 12 varies the shake threshold with the refinement budget fixed at 500 steps. Performance varies only modestly across shake thresholds, with values of 4, 8, and 32 performing at most 3.02% worse than the default value of 16 on average. The default threshold performs best on all six configurations. Disabling shake causes a larger 5.26% degradation, confirming that the shake operation is useful even though its precise threshold is less critical. Overall, a sufficient refinement budget becomes increasingly important as problem size grows, while the default shake threshold provides consistently strong performance.

Table 12: Sensitivity to the shake threshold on the six largest out-of-training-scale configurations. Values report total arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds; lower is better. The final column reports the average gain of the default threshold of 16 over each setting.

Threshold	C240_D6_K5	C249_D2_K14	C249_D3_K12	C249_D4_K8	C249_D5_K6	C360_D9_K5	Default gain (%)
4	732.26 (7.33)	998.94 (11.04)	737.57 (8.80)	727.30 (7.71)	772.96 (7.28)	1064.57 (8.21)	3.02
8	719.48 (4.45)	981.45 (6.70)	724.73 (6.27)	716.68 (6.97)	759.47 (6.78)	1046.06 (10.60)	1.35
16	709.55 (3.26)	971.73 (2.79)	713.32 (3.62)	708.18 (3.28)	749.72 (2.05)	1028.57 (4.28)	0.00
32	721.61 (4.60)	992.14 (5.01)	726.16 (5.34)	723.76 (4.72)	761.72 (3.63)	1053.26 (4.44)	1.93
No shake	746.45 (6.37)	1028.09 (12.21)	751.84 (6.07)	749.96 (7.47)	787.21 (7.31)	1090.28 (5.76)	5.26

Candidate-graph sparsity and components. The neighborhood size k in Algorithm 2 bounds both the kNN and the two-hop candidate edges. Table 13 evaluates the trained LIFT policies without retraining, with k halved from 64 to 32. Halving k degrades total arrival time by only 0.90% on average and at most 2.48%, and the $k = 32$ variant still outperforms every baseline on every configuration. We further remove individual edge sets in equation 3 while retaining the current-schedule edges. Removing the two-hop edges degrades performance by 0.54% on average, as they add longer-range alternatives that the dense kNN set largely covers. Removing the depot-customer edges E^{dep} degrades performance most, by 1.97%: they are the only ADD candidates for route-endpoint insertions and cross-route reassignment, and cannot be substituted by the other edge sets. LIFT is thus robust to candidate-graph sparsity, each edge set in equation 3 contributes, and the full candidate graph remains best.

Decoder insertion budget and temperature schedule. We further vary the decoder insertion budget M and the branch-wise temperature schedule on three representative configurations (C50_D4_K2, C75_D5_K3, and C80_D2_K5), using the same random seed across settings so that runs are directly comparable. Varying M from 4 to 14 leaves the final best schedule unchanged on all three configurations: the retained feasible insertion positions with the lowest insertion cost already contain the useful choices, so enlarging the retained set does not change the decoded schedules. For the temperature schedule, a more greedy schedule $[0, 0, 0.5, 1]$ and a more exploratory schedule $[0, 1, 2, 4]$ change total arrival time by at most 2.3% on a single configuration and by less than 0.7% on average. This is consistent with the single-branch results in Appendix F.3, where combining branches with different temperatures reduces the dependence on any particular temperature choice. Overall, LIFT requires a balanced exploration level rather than careful temperature calibration.

Table 13: Sensitivity to the candidate-graph neighborhood size on the 15 public benchmark instances, evaluating the trained LIFT policies without retraining under $k = 32$. Values report total arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds; lower is better. The final column reports the gain of the default $k = 64$ setting.

Instance	LIFT ($k = 32$)	LIFT ($k = 64$, default)	Default gain (%)
C50_D4_K4	47.76 (0.00)	47.76 (0.00)	0.00
C50_D4_K2	71.22 (0.84)	70.90 (0.53)	0.45
C75_D5_K3	76.27 (0.50)	76.03 (0.46)	0.31
C80_D2_K5	240.42 (0.33)	239.56 (0.29)	0.36
C100_D2_K8	148.47 (0.57)	147.89 (0.89)	0.39
C100_D2_K5	177.03 (1.52)	175.03 (1.44)	1.13
C100_D3_K6	108.02 (0.41)	105.34 (0.59)	2.48
C100_D4_K4	129.75 (1.05)	129.06 (1.00)	0.53
C160_D4_K5	467.99 (1.75)	465.35 (0.73)	0.56
C240_D6_K5	710.86 (4.55)	709.55 (3.26)	0.18
C249_D2_K14	985.01 (6.62)	971.73 (2.79)	1.35
C249_D3_K12	726.28 (4.00)	713.32 (3.62)	1.78
C249_D4_K8	719.05 (7.39)	708.18 (3.28)	1.51
C249_D5_K6	760.42 (5.85)	749.72 (2.05)	1.41
C360_D9_K5	1039.43 (5.21)	1028.57 (4.28)	1.04

Robustness under a nonlinear energy model. The energy model in equation 15 follows a widely used linear-in-payload approximation in UAV scheduling and drone delivery (Guo et al., 2025; Meng et al., 2023; Zhang et al., 2021; Rabta et al., 2018; Huang et al., 2022). In particular, prior work derives multirotor power as proportional to total mass raised to $3/2$ and validates a linear approximation using real quadcopter measurements (Dorling et al., 2016). To test sensitivity to this approximation, we additionally freeze the trained LIFT policies and replace the linear cruise-energy model with the underlying nonlinear load-dependent power model:

$$b_{ij}^{k,\mathrm{NL}} = \alpha H\left(\bar{w}x_{ij}^{k} + w_{ij}^{k}\right) + \beta L_{ij}\left(\bar{w}x_{ij}^{k} + w_{ij}^{k}\right)\sqrt{\frac{\bar{w}x_{ij}^{k} + w_{ij}^{k}}{\bar{w}}}. \tag{34}$$

The two models agree at zero payload, but the nonlinear model increases faster as payload grows, and a fully loaded route can consume up to 22.5% more energy. Because the nonlinear model imposes tighter energy constraints, its feasible set is smaller and the scheduling problem is inherently harder. We evaluate the trained LIFT policies on all 15 benchmark configurations without retraining, adopting the nonlinear energy model throughout inference while keeping all other settings unchanged. Feasibility under the nonlinear model is enforced by the decoder, so the relevant question is the cost of the tighter constraints: as reported in Table 14, total arrival time increases by only 4.0% on average and by at most 10.77%. This stress test evaluates sensitivity to nonlinear load dependence, but does not capture all sources of simulator-to-real mismatch, such as wind, battery degradation, and low-level flight-control errors; real-world validation is left to future work.

F.7 Failure-Case and Rejected-Action Analysis

This appendix analyzes when the sampled edits fail to produce useful feasible improvements, covering all 150 evaluation runs of full LIFT on the 15 benchmark configurations, comprising 1.12 million sampled actions. At each refinement step, every active branch decodes its sampled actions and retains the lowest-cost feasible successor, which is accepted only if it improves the branch's current schedule (Algorithm 5). A step can therefore fail in two ways: no sampled action admits a feasible successor, or the retained successor is rejected as non-improving, leaving the schedule unchanged. Table 15 reports per-instance statistics, and Table 16 classifies the retained successors by refinement stage, splitting each 500-step budget into equal thirds.

The policy-decoder pair produces a feasible successor for 99.98% of sampled actions, and steps where no sampled action admits a feasible successor occur on only three configurations, never exceeding 0.32%. Rejected successors are almost never harmful: only 15 of the 280,027 retained successors (0.005%) would have worsened the current schedule, and the remainder leave its cost unchanged.

Table 14: Robustness of the trained LIFT policies under the nonlinear load-dependent energy model, evaluated on the 15 public benchmark instances without retraining. Values report total arrival time in 10^2 seconds as mean (standard deviation) over 10 seeds. The final column reports the increase in total arrival time under the nonlinear model.

Instance	LIFT	LIFT (nonlinear energy)	Increase (%)
C50_D4_K4	47.76 (0.00)	47.76 (0.00)	0.00
C50_D4_K2	70.90 (0.53)	71.87 (1.13)	1.37
C75_D5_K3	76.03 (0.46)	77.03 (0.46)	1.32
C80_D2_K5	239.56 (0.29)	243.55 (0.28)	1.67
C100_D2_K8	147.89 (0.89)	150.44 (0.81)	1.72
C100_D2_K5	175.03 (1.44)	183.86 (1.63)	5.04
C100_D3_K6	105.34 (0.59)	116.68 (0.62)	10.77
C100_D4_K4	129.06 (1.00)	132.08 (1.81)	2.34
C160_D4_K5	465.35 (0.73)	475.13 (0.93)	2.10
C240_D6_K5	709.55 (3.26)	716.18 (4.68)	0.93
C249_D2_K14	971.73 (2.79)	1035.37 (6.46)	6.55
C249_D3_K12	713.32 (3.62)	769.39 (4.55)	7.86
C249_D4_K8	708.18 (3.28)	756.88 (4.50)	6.88
C249_D5_K6	749.72 (2.05)	799.92 (6.89)	6.70
C360_D9_K5	1028.57 (4.28)	1075.29 (8.62)	4.54

Table 15: Failure-case statistics over all 150 evaluation runs of full LIFT (15 benchmark configurations, 10 seeds). Accepted edits are retained successors that improve the branch's current schedule, with percentages measured per branch step; the mean gain is the resulting reduction in total arrival time per accepted edit; the final column reports the average step at which the final best schedule is found under the 500-step budget.

Instance	Accepted edits (%)	No feasible successor (%)	Mean gain per accepted edit (%)	Best-found step
C50_D4_K4	14.4	0.02	2.08	46
C50_D4_K2	15.5	0.32	2.47	306
C75_D5_K3	18.3	0.01	1.70	228
C80_D2_K5	17.1	0.00	1.96	166
C100_D2_K8	21.5	0.00	1.24	353
C100_D2_K5	24.0	0.00	1.36	347
C100_D3_K6	22.2	0.00	1.29	315
C100_D4_K4	23.0	0.00	1.18	342
C160_D4_K5	23.8	0.00	1.21	230
C240_D6_K5	30.4	0.00	0.74	363
C249_D2_K14	38.3	0.00	0.39	473
C249_D3_K12	37.0	0.00	0.51	452
C249_D4_K8	37.0	0.00	0.60	367
C249_D5_K6	34.9	0.00	0.59	447
C360_D9_K5	42.7	0.00	0.64	462

Table 16: Outcomes of the retained successors by refinement stage, pooled over all 150 evaluation runs; branches that stop early contribute only their observed steps. An accepted successor either sets a new branch-best schedule or recovers the current schedule after a shake without yet reaching it; rejected successors leave the schedule unchanged.

Refinement stage	Successors	New best (%)	Recovery (%)	Rejected (%)
Early (steps 1–167)	100,135	33.0	5.6	61.3
Middle (steps 168–334)	98,636	5.7	14.5	79.8
Late (steps 335–500)	81,256	3.3	18.4	78.3

These unchanged successors are reconstructions of the current schedule: when no sampled edit yields a cheaper feasible placement, the removed customers are reinserted at their original positions, so the retained successor coincides with the current schedule. Temporary worsening therefore enters the search only through shake, which perturbs the branch-best schedule by design: about 88% of shake moves worsen it before subsequent accepted edits recover the loss. The learned edits drive improvement; shake alone supplies the exploration.

Improvement concentrates early and changes character over time. New-best successors fall from 33.0% of steps in the first third of the budget to 5.7% and then 3.3%, so most progress on the branch-best schedule is made early, consistent with the refinement trajectories in Figure 3. Recovery instead grows from 5.6% to 18.4%: late in the search, improvement operates mainly through shake-and-recovery cycles, which is why disabling shake costs 4.22% (Appendix F.3).

The two scale groups fail differently. Within the training scale, 14–24% of steps are accepted, each accepted edit reduces total arrival time by 1.18–2.47% on average, and the final best is found after 46–353 steps. Beyond the training scale, edits are accepted more often (30–43% of steps) but gain less each time (0.39–0.74%), and the final best arrives only at steps 363–473. Large instances thus do not run out of useful edits; they need many small ones, which is why they benefit most from longer refinement budgets (Appendix F.6).

F.8 Representative Schedule Visualizations

Figure 6 visualizes representative schedules produced by LIFT on 15 benchmark configurations. Each subfigure is labeled by its instance setting. In each schedule, large circular nodes denote depots and small nodes denote customers. Directed edges with arrows indicate the visiting order. A closed tour that starts from a depot, visits a subset of customers, and returns to the same depot corresponds to the route of one drone. Thus, the number of closed tours around a depot reflects the number of drones dispatched from that depot in the shown schedule. Colors indicate depot-associated fleet groups, grouping together the routes served from the same depot. The examples cover diverse customer scales and depot layouts, illustrating how LIFT produces spatially coherent multi-drone schedules across heterogeneous benchmark configurations.

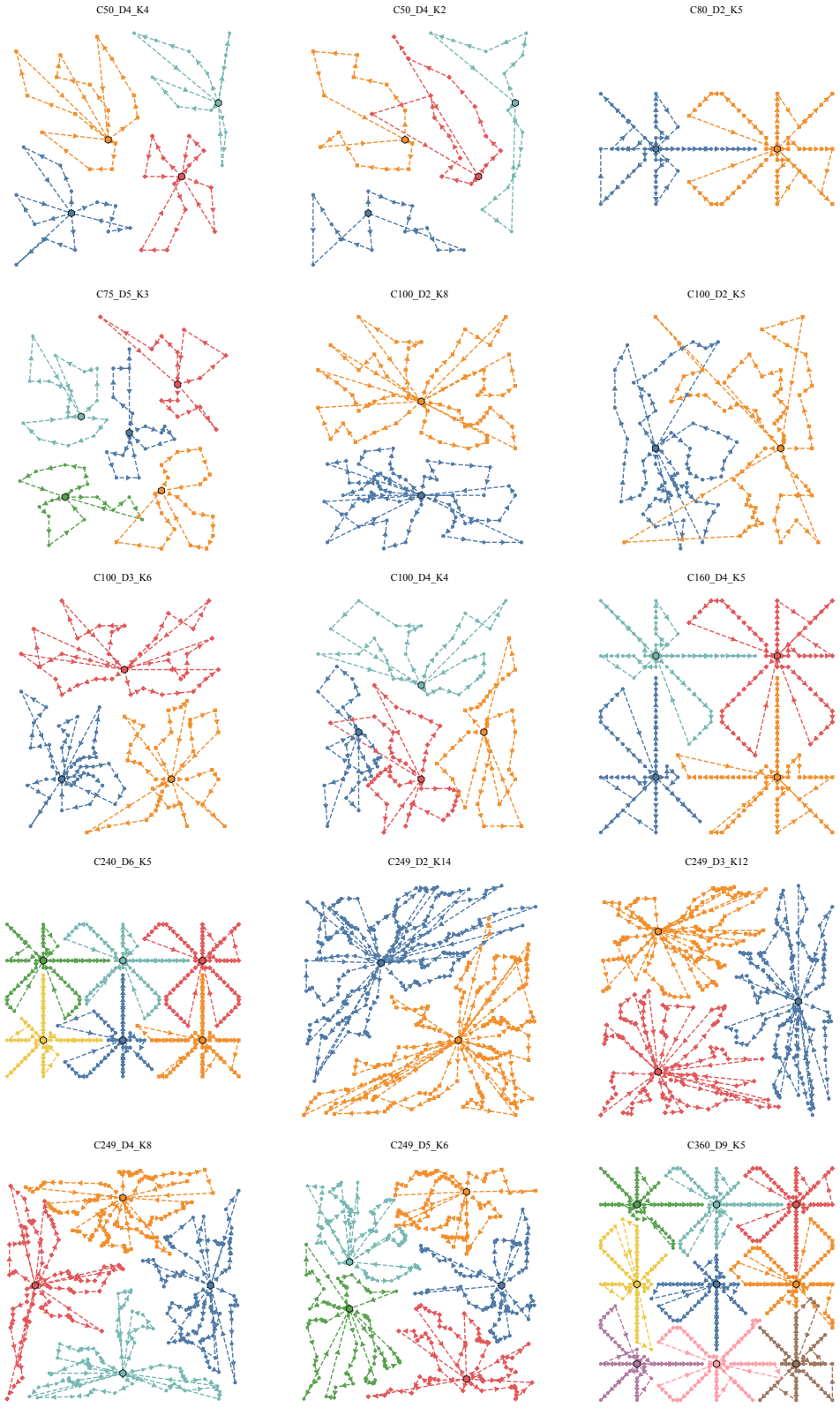

Figure 6: Representative LIFT schedules on the benchmark configurations.