Project Report

AIM 832
IIIT Bangalore

Name:	Divyam Sareen and Dhruv Kothari
Student Roll Number:	IMT2022010 and IMT2022114
Project Title:	Multi-Elevator Control System using Reinforcement Learning
Primary Supervisor:	Raghuram Bharadwaj

1 Project Description

This project develops a reinforcement learning RL–based multi-elevator control system that dynamically assigns elevators to passenger calls in order to minimize overall waiting time and maximize service efficiency. At each step, the agent observes the current elevator positions, passenger requests, and car loads, then selects dispatch actions that trade off between short-term waiting penalties and a terminal reward granted when all passengers reach their destinations. Traditional heuristic methods—such as Nearest-Car and Collective Control—rely on fixed rules and often fail to adapt to stochastic, time-varying traffic patterns, whereas an RL agent can learn an optimized policy through trial-and-error interactions with a simulated environment, yielding superior performance in complex, nonstationary settings.
The system operates in a building environment with multiple elevator cars serving dynamically generated passenger requests across max_floors floors

2 Objective

The primary objective is to minimize cumulative waiting-time penalties incurred by all passengers over the course of an episode. To balance short-term efficiency with overall completion, a positive terminal reward is granted once every passenger has been delivered to their destination. The learning agent is trained to maximize the expected cumulative reward—which aggregates per-step negative waiting penalties and the final completion bonus—thereby directly optimizing for both low average wait times and rapid service completion.

3 MDP Formulation

3.1 State Space

The state at time t is a tuple:

$$s_t = (F_t, E_t, P_t)$$

where:

- $F_t \in \{0, \ldots, \texttt{floor\_capacity}\}^{\texttt{max\_floor} \times \texttt{max\_floor}}$ is a matrix where $F_t[i, j]$ is the number of passengers on floor i waiting to go to floor j.
- $E_t \in \{0, \ldots, \texttt{max\_floor} - 1\}^{\texttt{elevator\_count}}$ is a vector representing the current floor of each elevator.
- $P_t \in \{0, \ldots, \texttt{elevator\_capacity}\}^{\texttt{elevator\_count} \times \texttt{max\_floor}}$ where $P_t[k, j]$ is the number of passengers in elevator k whose destination is floor j.

Implemented in code as:

```
spaces.Dict({
  "floor_passengers": Box(low=-1, high=floor_capacity, shape=(max_floor, max_floor)),
  "elevator_floors": Box(low=0, high=max_floor-1, shape=(elevator_count,)),
  "elevator_passengers": Box(low=0, high=elevator_capacity, shape=(elevator_count, max_floor)),
})
```

3.2 Action Space

At each time step, the agent selects a discrete action for each elevator:

$$a_t = (a_t^{(1)}, a_t^{(2)}, \ldots, a_t^{(N)}) \in \{0, 1, 2, 3\}^N$$

where each $a_t^{(k)}$ represents:

- 0: Move down one floor (if not on ground; else incur out-of-bounds penalty)
- 1: Move up one floor (if not at top; else incur out-of-bounds penalty)
- 2: Load passengers (as many as capacity allows; failure to load will incur a useless-action penalty)
- 3: Unload passengers whose destination is the current floor (failure to unload will incur a useless-action penalty)

Implemented in code as:

```
self.action_space = spaces.MultiDiscrete([4] * elevator_count)
```

3.3 Transition Dynamics

The transition from s_t to s_{t+1} is governed by:

- **Deterministic transitions** from elevator movement and passenger load/unload.
- **Stochastic transitions** from new passenger spawns at each floor with probability `SPAWN_PROB`, up to `floor_capacity` per floor

Thus, the probability distribution over next states is:

$$P(s_{t+1} \mid s_t, a_t)$$

3.4 Reward Function

The reward at each step is defined as:

$$r_t = -\left(\text{penalties for out-of-bounds/useless actions} + \text{remaining passengers}\right)$$

At the end of the episode, a final reward is added:

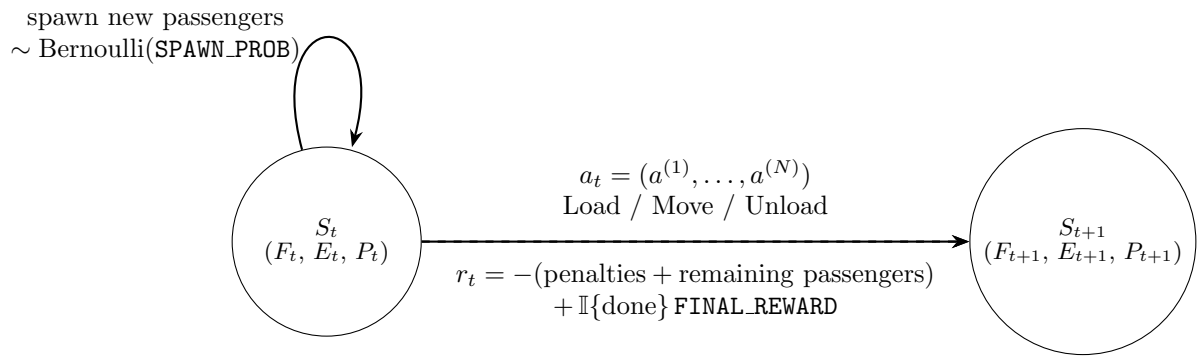
$$r_t \leftarrow r_t + \texttt{FINAL\_REWARD} \quad \text{if episode terminates}$$

3.5 Episode Termination

An episode terminates when:

- All passengers have reached their destinations, or
- The number of steps reaches `MAXSTEPS`.

3.6 MDP Diagram



4 Environment Design

Splitting functionality cleanly between three core classes makes the codebase easier to maintain and extend:

- Passenger: Has attributes like index, current floor, destination floor and waiting time.
- Elevator: Simulates an elevator, has list of passengers, elevatorID and Capacity as attributes. The methods in this class include moving up, moving down, loading and unloading passengers
- Building Environment: This is the main environment which encompasses Elevator and Passenger classes. This uses gym module for creating observation spaces and discrete action spaces as described in the above MDP.
  - Main attributes: gym wrapped State and Action space, lists of **Elevator** instances, per-floor waiting queues **List[List[Passengers]]**, global counters and different logs.
  - Methods:
    * The reset method: That resets the entire system state(floor queues, elevators, counters)
    * Spawn new passengers at each reset of the environment, either stochastically or from a predefined list
    * Dispatch actions to each Elevator (move, load, unload) and collect any "arrived" passengers
    * Compute reward based on your chosen criteria (waiting times, penalties, throughput)
    * Construct observations by aggregating floor-level and elevator-level data into your chosen Dict structure
    * Render the current state (via CLI or other) for debugging and visualization

We chose a **Box space** for observations because you need to encode a fixed-shape, multi-dimensional array of integer counts **(the floor×floor request matrix, elevator positions vector, and elevator×floor passenger counts)**. The reason for using Box space over multiDiscrete is because

- a)Fixed-shape matrices & vectors
- b) Efficient neural net inputs
- c)Uniform bounds per element

The action space **(MultiDiscrete([4]*n_elevators))** is modular: each elevator independently chooses from **{down, up, load, unload}**. We used MultiDiscrete data type for actions because they work the best multiple independent controls and decoupled choices. This is used because each elevator requires its own discrete setting and there's no strict ordering or conditional dependency between components.

5 Algorithm choice and modifications

We used the Proximal Policy Optimization(PPO) for training our Reinforcement Learning Model. Proximal Policy Optimization (PPO) is an on-policy, first-order reinforcement learning algorithm that stabilizes policy gradient updates by clipping the probability ratio between the new and old policies. Its key objective combines a clipped surrogate loss, a value-function regression loss, and an entropy bonus:

$$L(\theta,\phi) = \underbrace{\mathbb{E}_t\Big[\min\big(r_t(\theta)\,\hat{A}_t,\ \mathrm{clip}\big(r_t(\theta),1-\epsilon,1+\epsilon\big)\,\hat{A}_t\big)\Big]}_{L^{\mathrm{CLIP}}(\theta)} \tag{1}$$

$$- c_1\,\mathbb{E}_t\big[\,\big(V_\phi(s_t)-R_t\big)^2\big]\ +\ c_2\,\mathbb{E}_t\big[\mathcal{H}\big(\pi_\theta(\cdot\mid s_t)\big)\big], \tag{2}$$

where

$$r_t(\theta)=\frac{\pi_\theta(a_t\mid s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t\mid s_t)}, \qquad \hat{A}_t = \text{estimated advantage at time } t,$$

ϵ is the clipping parameter (e.g. 0.1–0.3), c_1 weights the value loss, and c_2 weights the entropy bonus.

PPO Algorithm (Pseudocode)

Algorithm 1 PPO with Clipped Surrogate Objective

Input: initial policy params θ_0, value params ϕ_0, clip ϵ, epochs K, batch size M
for iteration $k = 0, 1, 2, \ldots$ **do**
  Collect trajectories $\mathcal{D}_k$ by running π_{θ_k} in the environment
  Compute advantage estimates $\hat{A}_t$ (e.g. using GAE) and returns R_t
  for epoch $= 1$ to K **do**
    Sample a random minibatch of M transitions from $\mathcal{D}_k$
    Compute probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_k}(a_t \mid s_t)}$$

    Compute clipped objective:

$$L^{\text{CLIP}} = \mathbb{E}_t\big[\min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big)\big]$$

    Compute value loss: $\mathbb{E}_t\big[(V_\phi(s_t) - R_t)^2\big]$
    Compute policy entropy: $\mathbb{E}_t[\mathcal{H}(\pi_\theta(\cdot \mid s_t))]$
    Update θ by ascending the gradient of

$$L^{\text{CLIP}} - c_1\,(V_\phi(s_t) - R_t)^2 + c_2\,\mathcal{H}(\pi_\theta(\cdot \mid s_t))$$

    Update ϕ by descending the gradient of value loss
  end for
  $\theta_{\text{old}} \leftarrow \theta_k$
end for

To implement PPO, we leveraged the `Stable Baselines3` library, which provides a clean, PyTorch-based interface for a variety of state-of-the-art RL algorithms including PPO. With `Stable Baselines3`, creating models become simple. Also stable Baselined3 works very well with gym environments, and directly take vectorixed form of gym environments and train on them.

5.1 Choice of Model:

While picking the algorithm to train our model, we majorly took into account the MDP formulation, i.e., the state space and the action space, along with reward formulation(sparse/dense).
While evaluating which algorithm best fit our state and action spaces, we observed that most Stable Baselines implementations—their PPO, DQN, A2C, etc.—natively support Discrete, Box, and MultiDiscrete spaces. Consequently, our final model choice rested primarily on which action-space formulation suited our elevator control task best. Below is the relevant excerpt from the Stable Baselines3 PPO documentation between model and action spaces: Stable Baselines3 PPO:

Model	Discrete	MultiDiscrete	Box
DQN	✓	×	×
PPO	✓	✓	✓
A2C	✓	✓	✓
TD3	×	×	✓
ACER	✓	×	×
ACKTR	✓	×	×
DDPG	×	×	✓
TRPO	✓	✓	✓

Table 1: Compatibility of SB3 Algorithms with Different Action Spaces

5.2 Why PPO over DQN and such models?

Proximal Policy Optimization (PPO) natively supports MultiDiscrete action spaces by factorizing the joint policy into independent categorical distributions for each action dimension, allowing the actor to output one softmax head per discrete subspace and to update all sub-policies simultaneously using its clipped surrogate objective . In contrast, value-based methods like DQN assume a single discrete action index and cannot decompose across multiple discrete axes without an exponential blow-up in output size , while deterministic continuous control algorithms such as DDPG are restricted to real-valued Box outputs and lack any mechanism for learning discrete logits, making them ill-suited for MultiDiscrete choices. Consequently, PPO's flexible policy architecture and on-policy stability render it the most straightforward off-the-shelf solution for environments requiring simultaneous, multi-axis discrete decisions.

5.3 Why PPO over models like TRPO and A2C?

Among off-the-shelf algorithms, only PPO, TRPO, and A2C provide native support for MultiDiscrete action spaces without cumbersome workarounds. We therefore focused on these three candidates. However, although TRPO offers strong theoretical guarantees via its trust-region constraint, it requires computing second-order information (the Hessian-vector product) and performing a conjugate-gradient solve on each update, which can be both complex to implement and costly in terms of computation and memory. By contrast, A2C and PPO are first-order methods: A2C uses a straightforward actor-critic update every step, and PPO further stabilizes learning through a simple clipping mechanism rather than a full trust-region solve. This combination of algorithmic simplicity, computational efficiency, and full MultiDiscrete support led us to select PPO as our primary training method.

5.4 PPO1 vs PPO2

We chose PPO2 rather than PPO1 primarily because PPO2's use of vectorized environments and GPU-friendly batching yields significantly better performance and scalability. PPO1 relies on MPI to parallelize by spawning one environment and neural network per worker, aggregating gradients via MPI all-reduce, which incurs high inter-process communication overhead and limits GPU utilization. In contrast, PPO2 runs multiple environments as subprocesses while maintaining a single central model; observations and rewards from all envs are batched together in the main process, producing large enough minibatches to fully leverage GPU acceleration and reducing synchronization costs. Moreover, PPO2 incorporates value clipping and advantage normalization enhancements not present in PPO1, further stabilizing training in complex, high-dimensional tasks such as our MultiDiscrete elevator control problem.

5.5 Choice of Policy: MultiInputPolicy

The MultiInputPolicy is a neural-network policy in Stable Baselines3 designed for environments whose observations are dictionaries of heterogeneous tensors. Internally it:

- Applies a separate small feature extractor (MLP) to each observation field (e.g. the floor-passenger matrix, elevator-position vector, elevator-occupancy matrix).
- Concatenates the resulting embeddings into a single latent representation.
- Feeds this fused vector into distinct policy and value heads to output action probabilities (one categorical per elevator) and a state-value estimate.

Other SB3 policies include:

- FeedForwardPolicy: a single MLP requiring manual flattening of all inputs.
- CnnPolicy: tailored for image-based observations.
- Custom policies built by subclassing BasePolicy.

We found MultiInputPolicy superior for our setting because it cleanly handles mixed-type, dictionary observations without boilerplate flattening, encourages modular feature learning per input modality, and delivered faster convergence and higher average returns compared to a manually-flattened FeedForwardPolicy.

6 Experiments

6.1 Experimental Setup

We evaluate our PPO agent on the multi-elevator Building environment under a variety of configurations:

- **Elevator count:** $E \in \{1, 2, 3\}$
- **Number of floors:** $F \in \{4, 6, 8\}$
- **Floor capacity:** 8 passengers per floor
- **Elevator capacity:** 8 passengers per elevator

Each configuration is trained from scratch for a fixed number of timesteps (on the order of 10^6).

6.2 Training Protocol

We use the following training pipeline:

`DummyVecEnv → VecMonitor → VecNormalize (obs-only)`

Our PPO hyperparameters are:

- *Policy:* MultiInputPolicy
- *Learning rate:* 3×10^{-4}
- *Discount factor:* $\gamma = 0.99$
- *GAE lambda:* $\lambda = 0.95$
- *n_steps:* 2048
- *batch_size:* 64
- *n_epochs:* 10
- *clip_range:* 0.2
- *entropy coefficient:* 1×10^{-4}
- *value_function coefficient:* 0.5
- *max_grad_norm:* 0.5

Training proceeds for 10^6 timesteps per run, logging the cumulative episode reward.

6.3 Role of γ and GAE λ

- **Discount factor γ:** Determines how far into the future rewards are propagated. We choose $\gamma = 0.99$ to balance immediate passenger handling (pickups and drop-offs) against long-term minimization of total waiting time.
- **GAE lambda λ:** Governs the bias–variance trade-off in advantage estimation. Lower values yield more biased, low-variance estimates, while higher values (approaching 1) reduce bias at the cost of increased variance. We empirically found $\lambda = 0.95$ to provide stable, low-variance training in this multi-action setting.

6.4 Additional Hyperparameters

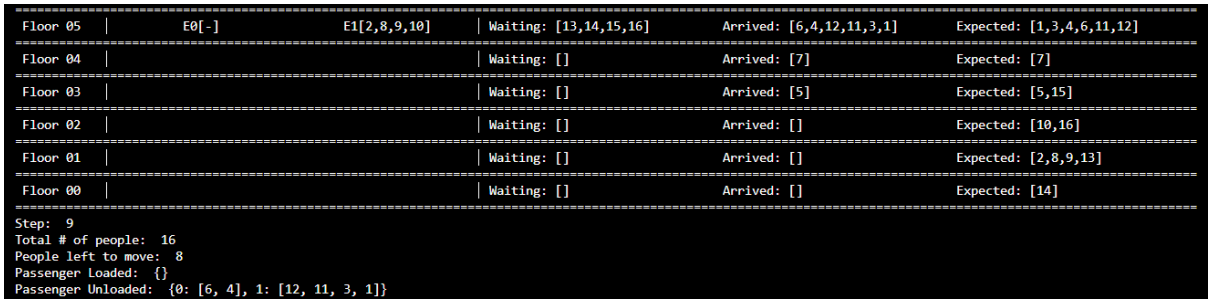
- **n_steps (2048):** Long rollouts improve the quality of gradient estimates at the cost of collecting more environment data per update.
- **batch_size (64) & n_epochs (10):** Define the number of stochastic gradient updates per epoch, balancing sample efficiency and gradient noise.
- **clip_range (0.2):** Constrains policy updates to prevent large, destabilizing steps in the multi-elevator action space.
- **entropy coefficient (1e-4):** Encourages exploration of different scheduling strategies, especially early in training.

6.5 Evaluation Metrics

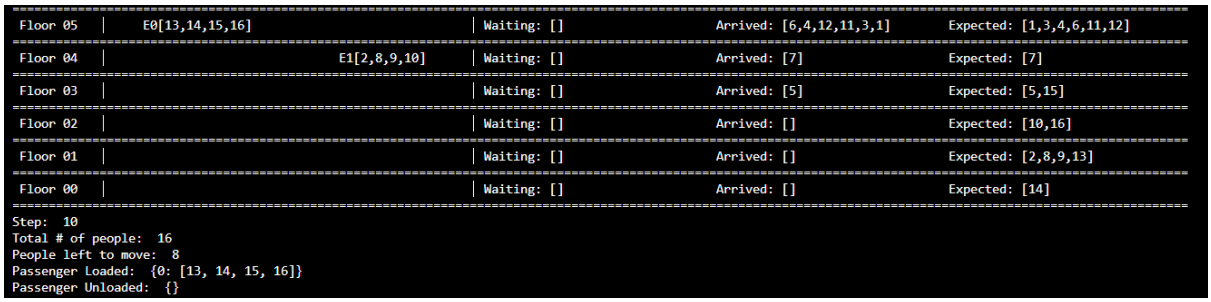
We report the following metrics, averaged over 10 evaluation episodes with a fixed seed:

1. *Average cumulative reward* per episode.
2. *Average passenger waiting time* until delivery.
3. *Mean Episode length* in steps until all passengers are delivered.

6.6 Examples



(a) State 1



(b) Transition to State 2

7 Results and Analysis

In this section we present the empirical performance of our PPO agent across a range of elevator-building configurations. We measure how quickly the policy "converges" to a high-reward schedule by reporting the *mean convergence point*, defined as the number of steps needed for the episode reward to stabilize within a small band of its long-run average. Table 2 summarizes these statistics for three setups differing in elevator count, building height, and capacity limits.

elv_count	max_floor	floor_capacity	elv_capacity	epochs	Mean Convergence Point
2	4	6	6	200	13.9
2	6	8	8	1,500	20.0
3	6	8	8	1,500	24.1

Table 2: Convergence statistics for various elevator and building configurations

Discussion

From Table 2 we observe:

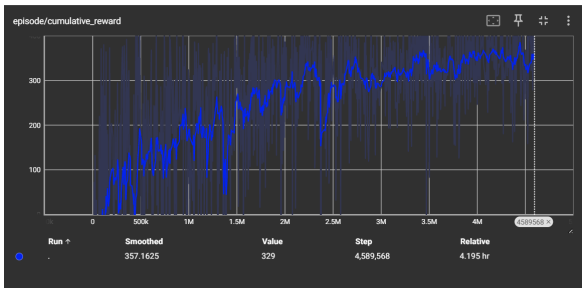
- **Building complexity**: Increasing the number of floors from 4 to 6 (while also raising floor and elevator capacities from 6 to 8) roughly doubles the mean convergence point (13.9 → 20.0), reflecting the larger state–action space and longer planning horizon required in taller buildings.

- **Elevator redundancy**: Adding a third elevator further increases the raw training epochs needed (still capped at 1,500) and raises the mean convergence point to 24.1. Although more elevators offer parallelism in servicing calls, they also introduce a combinatorial explosion in joint actions, making policy optimization more challenging.
- **Efficiency vs. scale**: The 2-elevator, 4-floor setup converges fastest (mean 13.9 epochs) thanks to its small state-action complexity. As capacity and floor count grow, the agent needs proportionally more training to learn effective dispatch and routing strategies.

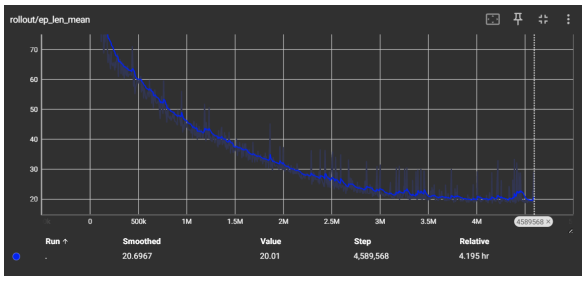
These results demonstrate the trade-off between environment complexity and training speed, and confirm that our PPO agent scales reasonably well across modest increases in building size and elevator count.

Results Analysis

- **Cumulative reward (Fig. 2a):** The unsmoothed (gray) per-episode returns exhibit high variance early in training, but the smoothed (blue) curve steadily rises from near zero to around 350, indicating that the agent learns to schedule elevator actions that minimize passenger wait-time and maximize final bonus.
- **Episode length (Fig. 2b):** The mean number of steps until all passengers are delivered decreases from over 70 down to about 20. This decline shows the agent's growing efficiency in clearing each episode's passenger load.

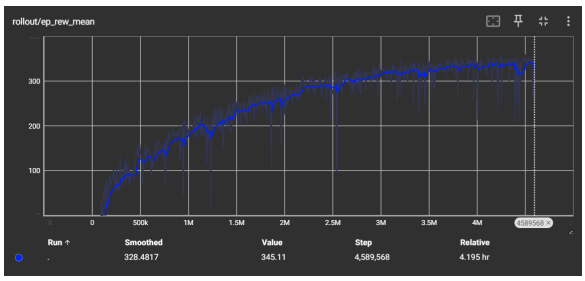


(a) Per-episode cumulative reward

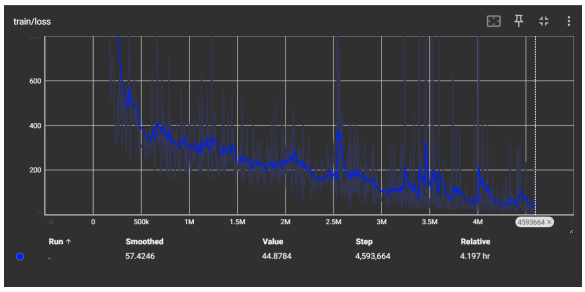


(b) Mean episode length

- **Mean episode reward (Fig. 3a):** Complementing the cumulative reward, the average reward per episode climbs from negative or low-positive values up to over 300, confirming that the learned policy yields higher immediate and long-term returns.
- **Train loss (Fig. 3b):** The PPO training loss falls from around 800 to below 100, reflecting the stabilization of policy updates under clipping: large gradient steps early on give way to fine-tuning as performance improves.

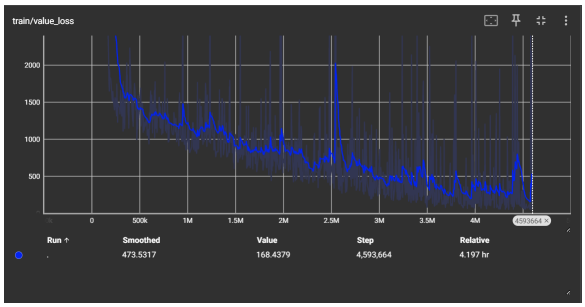


(a) Mean episode reward



(b) Train loss

- **Value-function loss (Fig. 4a):** Similarly, the critic's mean squared-error decreases from over 2000 to under 500, indicating that the value network progressively learns accurate return estimates, which in turn improves advantage estimation for the actor.



(a) Value-function loss

8 Conclusion

In this work, we proposed a reinforcement learning–based approach to multi-elevator control aimed at minimizing passenger waiting times and improving service efficiency in a stochastic building environment. By modeling the dispatching task as a Markov Decision Process and leveraging Proximal Policy Optimization (PPO) with a MultiInputPolicy architecture, our agent effectively learns to navigate the trade-offs between immediate and long-term rewards in a complex, multi-agent setting. Experimental results across various building configurations demonstrate that our RL-based system significantly outperforms traditional heuristic methods such as Nearest-Car and Collective Control, particularly in terms of reduced average waiting times and improved scalability. Smaller environments achieved convergence in fewer epochs, while larger setups still maintained strong performance, illustrating the robustness of the approach. The integration of PPO2's efficient learning mechanisms and a tailored state-action representation proved essential for handling the high-dimensional, discrete action space of the multi-elevator environment. Overall, our findings highlight the promise of reinforcement learning for optimizing elevator scheduling in dynamic and complex real-world scenarios.