# Optimizing Dynamic Replica Placement in in Wide-Area Replicated Systems via Reinforcement Learning

Lukas Bierling[1], Kevin Tierney[2], and Alena Otto[3]

[1] University of Amsterdam
[2] University of Vienna
[3] Technical University Munich

**Abstract.** Replicated state machines (RSMs) are a core abstraction for building fault-tolerant distributed services, but their performance in wide-area deployments is highly sensitive to replica placement. Static placement strategies fail to adapt to dynamic and geographically shifting client demand, while existing dynamic approaches rely on greedy, myopic reconfiguration heuristics with non-negligible runtime overhead. In this work, we study dynamic replica placement as a stochastic location-allocation problem and formalize it within an optimization framework related to dynamic facility location. We propose a reinforcement learning-based solution that learns replica reconfiguration policies offline from simulated system trajectories and enables constant-time decision making at runtime. The approach combines spatio-temporal graph representations with policy optimization to capture global communication effects and long-term trade-offs. Experimental results using a full-system simulation show that the learned policy consistently outperforms both static placement and the existing greedy baseline, with increasing benefits for longer horizons and larger configuration spaces. These findings demonstrate that learning-based optimization is a practical and effective alternative for dynamic replica management in wide-area replicated systems.

**Keywords:** Deep Reinforcement Learning · Optimization · Graph Neural Networks · Distributed Systems

## 1 Introduction

Distributed systems are a central building block of modern computing infrastructures, enabling scalable, reliable, and geographically distributed services. To tolerate failures and ensure correct behavior despite partial outages, many systems rely on replication mechanisms that coordinate multiple nodes to provide a single logical service. Replicated state machines (RSMs) provide an abstraction for such services, providing strong consistency and fault tolerance guarantees to provide a deterministic service across multiple replicas. In wide-area deployments, however, these guarantees come at a substantial performance cost, as client request latency is dominated by network delays and the coordination overhead required for inter-replica agreement.

In practice, replica locations are typically selected statically at system startup. This design implicitly assumes relatively stable client demand. In contrast, real-world workloads are highly dynamic, exhibiting diurnal patterns, regional shifts, and abrupt changes caused by external events. As a result, static replica placement often leads to persistently suboptimal performance. Recent systems work has shown that dynamically relocating replicas at runtime can significantly reduce average request latency by adapting to evolving demand distributions [13].

From an optimization perspective, dynamic replica placement represents a challenging sequential decision problem. Decisions must be taken repeatedly over time, reconfiguration actions incur explicit costs, and system performance depends on the interaction between spatial structure and stochastic, time-varying demand. Existing approaches primarily rely on strategies that greedily reconfigure with respect to the current system state and evaluate candidate configurations online [13].

In this work, we formalize dynamic replica placement in replicated state machines as a Dynamic Stochastic Facility Location Problem (DSFLP) [8]. Unlike classical facility location models, the problem is shaped by replicated execution semantics, where service costs depend on the collective placement of all active replicas rather than on assignments to individual facilities. A central challenge in this setting is the need for fast and predictable decision making in production environments. Replica reconfiguration decisions must be computed frequently and with minimal runtime overhead, which limits the applicability of exact optimization methods and complex online heuristics. This motivates the use of reinforcement learning (RL): by shifting computational effort to an offline training phase, RL enables the learning of policies that approximate the solution of the underlying stochastic optimization problem and can be evaluated at runtime with negligible cost.

We therefore propose a learning-based optimization approach that combines spatio-temporal graph representations with RL to compute adaptive replica relocation decisions. The main contributions of this paper are as follows:

1. We formalize dynamic replica placement in replicated state machines as a DSFLP.
2. We propose a mixed-integer linear programming formulation capturing spatial and temporal aspects of replica reconfiguration.
3. We introduce a reinforcement learning approach that enables constant-time replica reconfiguration decisions at runtime.
4. We provide a fully reproducible simulation framework to support future research at the intersection of learning and dynamic optimization.

## 2 Related Work

### 2.1 Facility Location and Dynamic Location-Allocation

Location-allocation problems are a standard topic in operations research and include well-studied formulations such as the $p$-median problem and the Facility Location Problem (FLP) [23, 1]. These models aim to determine facility placements and demand assignments that minimize transportation or service costs. Dynamic extensions incorporate temporal variation in demand, facility availability, and relocation costs, leading to the Dynamic Deterministic FLP (DDFLP) and the Dynamic Stochastic FLP (DSFLP) [7, 8].

Even static variants of facility location are NP-hard, and the introduction of temporal dynamics or stochastic demand further increases computational complexity [9]. As a result, large-scale or time-critical applications typically rely on heuristics, approximations, or decomposition-based approaches rather than exact optimization [19]. From an optimization perspective, such approaches can be interpreted as local heuristics for a dynamic location-allocation problem under uncertainty, but they do not explicitly model long-term trade-offs or future demand evolution.

### 2.2 Learning Based Approaches For Networked Optimization Problems

Reinforcement learning (RL) provides a general framework for solving sequential decision problems under uncertainty by learning a policy that maps observed system states to actions to optimize long-

term expected performance [18]. Recent work has demonstrated that deep reinforcement learning (DRL) can learn effective heuristics for combinatorial optimization problems by approximating complex decision rules that would otherwise require costly online computation [21]. Rather than solving each optimization instance from scratch, a trained policy directly produces decisions based on the current state, implicitly capturing long-term trade-offs and delayed effects [27, 29, 2].

Many dynamic optimization problems arising in distributed systems, including facility location and replica placement, are naturally defined on graphs. Graph neural networks (GNNs), and in particular graph attention networks (GATs), provide a principled way to learn representations of such graph-structured states by aggregating information across nodes and edges [26]. Combining GNNs with RL enables policies to exploit relational structure and global context while making sequential decisions, and has proven effective for networked optimization problems with complex interactions [17]. This work builds on these ideas by modeling dynamic replica placement in wide-area replicated state machines as a sequential decision problem on a spatio-temporal graph. By combining graph-based state representations with reinforcement learning, we aim to learn reconfiguration policies that capture protocol-level latency dependencies and long-term reconfiguration trade-offs, while enabling efficient runtime decision making.

## 3 Replicated State Machines in Wide-Area Networks

Distributed systems are a core building block of modern computing infrastructures, enabling scalable and reliable services across geographically distributed environments. To tolerate failures and provide strong consistency guarantees, many systems rely on replicated state machines (RSMs), which execute a deterministic service on multiple replicas under a globally agreed-upon order of client requests [14, 5]. RSMs are widely used in practice in settings such as distributed databases, coordination services, and globally replicated storage systems.

*Wide-Area Deployments and Latency* Modern distributed systems increasingly span wide-area networks (WANs) to serve globally distributed clients. In such environments, network latency dominates local computation, and client-perceived performance is primarily determined by communication delays rather than processing time. This effect is amplified in replicated systems. A single client request typically triggers multiple message exchanges: between the client and one or more replicas, and among replicas themselves as part of the agreement protocol. As a result, request latency reflects not only client-replica distances but also the communication structure induced by the replication protocol [13]. A request-response cycle consists of several protocol phases, including client-to-replica communication, inter-replica agreement, and replica-to-client responses, each contributing one or more transmission delays (more details in Appendix B.1).

*Byzantine Faults and Fault Tolerant Replication* An especially important fault model in wide-area deployments is that of Byzantine faults, where replicas may fail arbitrarily, including sending corrupted or inconsistent messages or executing incorrect code. While often considered pessimistic, Byzantine faults are practically relevant. Faults caused by software bugs, hardware errors, or misconfigurations have been observed in real distributed systems [3]. Byzantine-fault-tolerant (BFT) RSM protocols address this threat model by relying on quorum-based agreement. Classical results show that tolerating up to $f$ Byzantine faults requires at least $n \geq 3f + 1$ replicas and that certain protocol phases can only complete once matching responses from a sufficiently large subset of replicas (usually $2f + 1$) have been received [5].

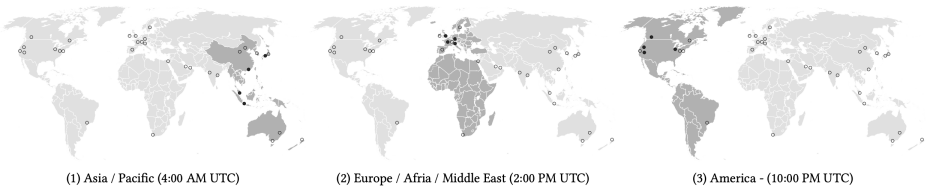



Fig. 1: A system of four replicas (solid circles) is dynamically migrated across datacenters (circles) around the globe at different times of the day (current time of day shaded in dark gray), from [13].

*Configuration-Dependent Latency* Because BFT protocols require agreement among multiple replicas, the latency of a client request is not determined by a single replica in isolation. Instead, a request can only complete once the protocol has collected matching responses from multiple replicas, potentially located at geographically distant sites. Consequently, latency is a function of the entire configuration of active replicas, rather than a local function of client proximity to a single server. Even if a client is close to one replica, the overall request latency may be dominated by slower inter-replica communication paths required for agreement. Consider the example shown in Figure 6, in which demand shifts with the current time of the day. Replicas are shifted corresponding to the time of day and location of the demand among the available datacenters.

*Latency Awareness* A prerequisite for adapting replica placement in wide-area replicated systems is the ability to observe and reason about communication latencies between replicas and clients. In practice, client locations may change dynamically and are not known in advance, while replica locations are typically drawn from a fixed set of data center regions. Systems therefore maintain a continuously evolving view of client locations, replica placements, and measured communication latencies. One practical approach is to continuously measure network transmission delays during normal protocol execution and aggregate them into a global, time-dependent latency representation [13]. This information provides the empirical basis for reasoning about the performance implications of different replica configurations.

## 4 Problem Formulation and Modeling

We consider an RSM deployed over a geographically distributed infrastructure. A fixed set of candidate replica locations $L$ is given, corresponding to data center regions that can host replicas. A dynamically evolving population of clients issues requests from different geographical locations with time-varying request volumes. The system operator can dynamically select which replica locations are active at each point in time. The goal is to minimize long-term system latency while accounting for the operational and synchronization costs incurred by replica reconfiguration. In contrast to standard facility location problems, due to replicated execution and quorum-based coordination, request latency depends on the *entire set of active replicas*.

### 4.1 System State Representation

We represent the state at time $t$ as a graph $G_t = (V_t, E_t)$. The node set $V_t$ consists of two disjoint subsets: a fixed set $L$ of candidate replica locations and a time-dependent set $C_t$ of client locations

active at time $t$. The graph is fully connected. Each edge $(u, v) \in E_t$ is weighted by the observed communication latency between the corresponding endpoints. Latencies between replica locations are assumed to be known and slowly varying, while latencies between clients and replica locations are stochastic and time-dependent. Each client node $c \in C_t$ is associated with a request volume $q_{c,t}$, indicating its contribution to the overall system load.

### 4.2 Problem Formalization

We formalize dynamic replica placement as a stochastic, multi-period decision problem over a finite planning horizon $T = \{1, \ldots, H\}$.

*Random variables and system evolution.* Let $\{S_t\}_{t=1}^{H}$ be a stochastic process defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Each random variable $S_t$ represents the system state at time $t$. A realization $S_t = s$ specifies: (i) the set of active clients $C_s$, (ii) their request volumes $\{q_{c,s}\}_{c \in C_s}$, and (iii) the realized communication latencies between clients and replica locations, as well as between replica locations. The joint distribution of $(S_1, \ldots, S_H)$ captures temporal correlations in demand and network conditions. At each time step $t$, the decision maker observes the current realization $S_t$ and selects a replica configuration for the next period.

*Decision variables.* For each replica location $l \in L$ and time step $t \in T$, let $y_{l,t} \in \{0, 1\}$ be set to 1 iff replica $l$ is active at time $t$, and $z_{l,t} \in \{0, 1\}$ be set to 1 iff replica $l$ is activated at time $t$. The set of active replicas at time $t$ is $L_t^+ = \{l \in L \mid y_{l,t} = 1\}$.

*Latency as a set-dependent random cost.* Let $D(L_t^+, S_t)$ denote the aggregate request latency incurred when the system operates with active replica set $L_t^+$ in system state $S_t$. This function captures client–replica communication delays, inter-replica coordination overhead, and quorum-based completion constraints induced by the replicated state machine protocol. Importantly, $D(\cdot)$ is *not additive* in individual replicas and cannot be decomposed into per-replica or per-client terms. Latency depends on the entire active replica configuration.

We assume the following monotonicity property: $\mathbb{E}[D(L', S_t)] \leq \mathbb{E}[D(L, S_t)]$ for all $L \subseteq L'$. That is, activating additional replicas weakly reduces expected latency by increasing redundancy and shortening critical communication paths. The objective and constraints are as follows.

$$\min \ \mathbb{E}\left[\sum_{t=1}^{H}\left(\sum_{l \in L} k_a y_{l,t} + \sum_{l \in L} k_r z_{l,t} + D(L_t^+, S_t)\right)\right] \tag{1}$$

$$\begin{aligned}
\text{subject to} \quad & z_{l,t} \geq y_{l,t} - y_{l,t-1} && \forall l \in L,\ t \geq 2, && (2)\\
& z_{l,t} \leq y_{l,t} && \forall l \in L,\ t \in T, && (3)\\
& \sum_{l \in L} y_{l,t} \leq K && \forall t \in T, && (4)\\
& y_{l,t}, z_{l,t} \in \{0, 1\} && \forall l \in L,\ t \in T. && (5)
\end{aligned}$$

The objective function (1) minimizes the expected cumulative system cost over the planning horizon. Constraints (2) define activation events by linking $z_{l,t}$ to changes in the activity variable $y_{l,t}$, ensuring that a reconfiguration cost is incurred whenever a replica transitions from inactive to

active. Constraints (3) prevent wrong activations by enforcing that a replica can only be activated if it is active in the resulting configuration. Constraints (4) enforce the maximum number of active replicas at each time step, reflecting fault-tolerance and resource constraints. This formulation models dynamic replica placement as a stochastic, multi-period location–allocation problem with set-dependent service costs and explicit reconfiguration penalties. Even the deterministic single-period variant is NP-hard, as it generalizes classical facility location problems.

## 5 Learning-Based Optimization Approach

The formulation in Section 3 provides a rigorous description of dynamic replica placement but is computationally infeasible for online decision making in production systems. Even approximate or greedy methods require repeated online evaluation of candidate configurations and thus incur non-negligible runtime overhead. We therefore adopt a learning-based optimization approach that shifts computational effort to an offline training phase. A replica reconfiguration policy is learned from simulated system trajectories and can subsequently be evaluated at runtime in constant time, enabling fast and predictable decision making.

### 5.1 Motivation for Reinforcement Learning

We address the dynamic replica placement problem using reinforcement learning by modeling it as a sequential decision problem under uncertainty and learning a reconfiguration policy offline from simulation. Instead of performing repeated online evaluation of candidate configurations, the learned policy directly maps observed system states to replica reconfiguration actions, enabling constant-time decision making at runtime. This shift from online optimization to offline learning is important in production replicated systems, where decisions must be taken frequently and with minimal computational overhead.

A major advantage of DRL in this setting is its ability to jointly learn a decision policy and a compact state representation. The raw system state is a high-dimensional and noisy graph consisting of replica locations, dynamically appearing clients, request volumes, and latency measurements. Rather than relying on handcrafted features or explicit aggregation rules, the policy network learns low-dimensional latent representations that preserve decision-relevant structure. This capability has proven effective for dynamic combinatorial optimization problems such as routing, scheduling, and facility location, where costs depend on complex interactions and delayed effects [17, 29, 2, 10, 28]. In our setting, it enables the policy to optimize long-term latency and reconfiguration trade-offs beyond myopic heuristics.

### 5.2 Markov Decision Process Formulation

Before formalizing the MDP, we highlight two key modeling choices that make the learning problem tractable while preserving the essential structure of the underlying optimization problem. First, the action space is restricted to incremental reconfiguration steps in each decision period, allowing at most one replica location to be activated and at most one to be deactivated. This reflects practical constraints on synchronization overhead in replicated systems and avoids the combinatorial explosion associated with selecting arbitrary replica subsets. Second, temporal dependencies are modeled only for replica locations. Replica placement decisions have persistent effects over time due to state transfer and coordination costs, whereas client locations, request volumes, and network

latencies are treated as exogenous stochastic inputs that evolve independently of the system's control. These design choices significantly reduce the complexity of the decision process while retaining the long-term trade-offs between latency reduction and reconfiguration cost. We model dynamic replica placement as a MDP defined by the tuple $(\mathcal{S}, \mathcal{A}, P, r)$.

**State.** A state $s_t \in \mathcal{S}$ represents the system configuration at time $t$ and consists of the set of active replica locations $L_t^+ \subset L$ and the observed system graph $G_t = (V_t, E_t)$ encoding current client locations, replica locations, request volumes, and measured latencies.

**Action.** An action $a_t \in \mathcal{A}(s_t)$ is defined as a tuple $a_t = (i_t, j_t)$, where $i_t \in L_t^+ \cup \{\varnothing\}$ denotes a replica location to deactivate and $j_t \in (L \setminus L_t^+) \cup \{\varnothing\}$ denotes a replica location to activate. The null element $\varnothing$ allows no-op actions. This formulation subsumes removals, additions, and swaps while enforcing bounded reconfiguration per time step.

**Transition.** The transition kernel $P(s_{t+1} \mid s_t, a_t)$ is induced by two components. First, the action $a_t$ deterministically updates the set of active replicas according to $L_{t+1}^+ = (L_t^+ \setminus \{i_t\}) \cup \{j_t\}$. Second, the system graph evolves stochastically due to exogenous factors, including client arrivals and departures, time-varying request volumes, and stochastic latency realizations. The resulting next state $s_{t+1}$ therefore consists of the updated replica configuration and the newly observed graph $G_{t+1}$.

**Reward.** The immediate reward is defined as the negative system cost incurred at time $t$, consistent with the stochastic optimization problem in Section 4:

$$r(s_t, a_t) = -\Big( D(L_t^+, S_t) \; + \; \sum_{l \in L} k_r \, z_{l,t} \; + \; \sum_{l \in L} k_a \, y_{l,t} \Big),$$

where $L_t^+ \subseteq L$ denotes the set of active replica locations after applying action $a_t$, and $z_{l,t}$ is as previously defined. Here, $D(L_t^+, S_t)$ denotes the aggregate request latency induced by executing the replicated state machine with active replica set $L_t^+$ under system state $S_t$, capturing both client-replica communication delays and inter-replica coordination required by the consensus protocol.

In our experimental setting, operational costs are set to $k_a = 0$ (in accordance with [13]) and are therefore omitted. Maximizing expected cumulative reward is thus equivalent to minimizing expected cumulative system latency under reconfiguration costs, consistent with the objective of the stochastic mixed-integer formulation.

### 5.3 Spatio-Temporal Graph-Based State Representation and Policy Parameterization

At each time step $t$, the system state is represented as a fully connected graph $G_t = (V_t, E_t)$ and $V_t = L \cup C_t$ where $L$ denotes the fixed set of potential replica locations and $C_t$ the dynamically appearing client locations. Each edge $(u, v) \in E_t$ is associated with a scalar weight $d_{u,v,t} \in \mathbb{R}_{\geq 0}$ representing the observed network latency between the corresponding endpoints.

Each node $v \in V_t$ is assigned a feature vector $x_v^t \in \mathbb{R}^{d_0}$, encoding the node type (replica location or client) as a one-hot indicator and the current request volume (clients only; zero for replica locations). Stacking all node features yields the node feature matrix $X_t \in \mathbb{R}^{|V_t| \times d_0}$. A spatial GAT [11, 6, 26] is applied independently at each time step to compute node embeddings

$$H_t = \mathrm{GAT}_\theta(X_t, D_t) \in \mathbb{R}^{|V_t| \times d_s},$$

where $D_t = (d_{u,v,t})$ denotes the latency-weighted adjacency matrix. Through message passing over the fully connected graph, each replica location embedding aggregates information about all clients,

other replicas, and their associated latencies. Temporal dependencies are modeled only for replica locations. For each location $l \in L$, we maintain a sequence of spatial embeddings over a window of length $\tau$, $(h_{l,t-\tau}, \ldots, h_{l,t}) \in \mathbb{R}^{\tau \times d_s}$, which is processed by a shared temporal model, an LSTM [12] to produce a spatio-temporal embedding $\tilde{h}_{l,t} \in \mathbb{R}^{d_t}$. Client embeddings are not temporally aggregated as clients are transient. Collecting all location embeddings yields the state representation used for decision making, $\tilde{H}_t \in \mathbb{R}^{|L| \times d_t}$.

The policy is factorized into a removal and an addition component,

$$\pi_\theta(a_t \mid s_t) = \pi_\theta^{\text{rem}}(i_t \mid s_t)\, \pi_\theta^{\text{add}}(j_t \mid s_t, i_t).$$

Removal logits are computed as

$$u_t = \tilde{H}_t W_{\text{rem}} \in \mathbb{R}^{|L|},$$

followed by a masked softmax over $L_t^+$. Conditioned on the selected removal $i_t$, the embedding $\tilde{h}_{i_t,t}$ is concatenated to each row of $\tilde{H}_t$, and addition logits are computed as

$$v_t = [\tilde{H}_t \,\|\, \tilde{h}_{i_t,t}\mathbf{1}]W_{\text{add}} \in \mathbb{R}^{|L|},$$

followed by a masked softmax over $L \setminus L_t^+$. The state-value estimate is obtained via permutation-invariant pooling over location embeddings,

$$V_\phi(s_t) = g_\phi\Big(\text{pool}(\tilde{H}_t)\Big),$$

where pool$(\cdot)$ denotes mean or attention-based pooling [15] over the spatio-temporal location embeddings $\tilde{H}_t$. The policy and value function are trained using a policy-gradient method based on Proximal Policy Optimization (PPO) [24]. Given trajectories sampled from the simulator, the policy parameters $\theta$ are optimized to maximize expected cumulative reward under the clipped PPO objective, while the value function parameters $\phi$ are trained to approximate the expected return. Details of the training objective are provided in Appendix D.

## 6 Experimental Evaluation

### 6.1 Baselines

We compare the proposed learning-based approach against two baselines: static replica placement and a recent, well established greedy reconfiguration strategy based on Fluidity [13]. The greedy baseline follows a myopic, locally optimal reconfiguration strategy that minimizes observed system latency with respect to the current system state. Fluidity continuously monitors client-replica and inter-replica latencies during normal system operation using additional passive replica locations. These passive locations serve exclusively for latency measurement and exploration; they do not participate in request processing and therefore do not affect the realized system latency. At each decision step, Fluidity evaluates a restricted neighborhood of candidate configurations obtained by exchanging one active replica location with one passive location. For each candidate configuration, the resulting aggregate request latency is estimated based on the most recent measurements, and the configuration with the lowest estimated latency is selected for execution in the next period.

From an optimization perspective, Fluidity implements a local, one-step lookahead heuristic. Decisions are optimized with respect to the current observed state and latency measurements,

without explicitly modeling future demand evolution or longer-term reconfiguration costs. This strategy has been shown to be effective in practice, but requires repeated online evaluation of multiple candidate configurations and does not account for long-term trade-offs. We therefore use it as a strong baseline for the underlying stochastic replica placement problem.

### 6.2 Simulation Setup

All experiments are conducted using the full simulation environment introduced in the Fluidity system [13]. The original Java-based replicated state machine simulator is integrated into a Python-based RL pipeline via a custom OpenAI Gym-compatible wrapper [4], using `JPype` for cross-language interaction [22]. This allows the learning agent to interact with the unmodified Fluidity execution model and latency measurement infrastructure.

Each environment step corresponds to one replica reconfiguration decision interval. The simulator exposes the current system graph, including active replica locations, client locations, request volumes, and measured communication latencies. The selected reconfiguration action is applied before advancing the simulation. All evaluated approaches (static placement, the greedy Fluidity baseline, and the learned policy) operate on identical system dynamics and latency abstractions. On environment reset, the simulator samples a random starting configuration of data centers on which the decision maker has to operate. Refer to Appendix B.1 for a detailed explanation of the simulation. In all experiments, reported system latency corresponds to the realized value of the set-dependent latency function $D(L_t^+, S_t)$, measured by the simulator during replicated execution.

We evaluate three scenarios of increasing temporal horizon and combinatorial complexity. The **Short horizon** with 40 decision steps and 8 candidate replica locations from which four are active following the rule for BFT systems [13]. **Medium horizon** with 400 decision steps and 8 candidate replica locations and **long horizon** with 1500 decision steps and 15 candidate replica locations. Policies are trained and evaluated independently for each setup[4].

### 6.3 Training Behavior

For all scenarios, the policy and value function are trained using PPO as described in Section 5.3 and Appendix D. Figure 2 illustrates the training dynamics of the proposed RL approach across three problem settings of increasing temporal horizon and combinatorial complexity (In Appendix E, the absolute training steps are shown).

In the short-horizon setting with 40 decision steps and 8 candidate locations (Figure 11), the policy converges rapidly. Rewards increase sharply during early training and stabilize after few updates, reflecting the limited temporal credit assignment problem and the small action space. In this regime, effective reactive reconfiguration strategies can be learned quickly.

For the medium-horizon setting with 400 steps and 8 locations, learning progresses more gradually. The reward improves steadily over a longer training period, indicating that the policy must account for delayed effects of reconfiguration decisions and balance short-term latency reductions against future costs. Compared to the short-horizon case, training exhibits increased stability and smoother improvement, consistent with richer temporal dependencies.

The long-horizon setting with 1500 steps and 15 candidate locations presents the most challenging learning problem. Here, reward improvement is slower and initially more variable due to

[4] All of these setups are included as reproducible run configurations in the corresponding repository of this work `https://github.com/Coluding/rsm-final`

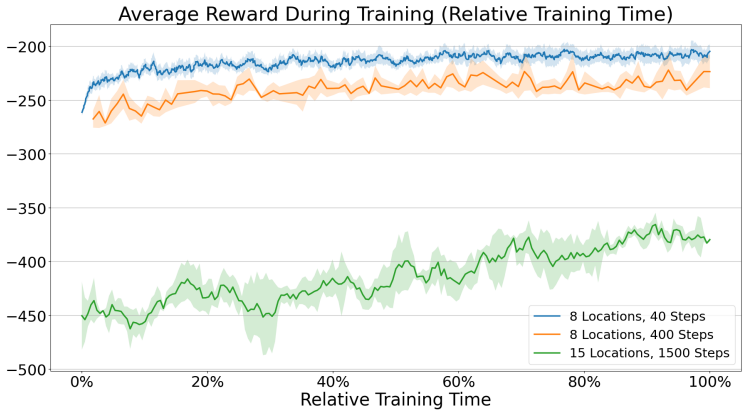



Fig. 2: Training reward (vertical axis) over relative training time (horizontal axis) for the three experimental scenarios. Higher reward corresponds to lower average system latency and reduced reconfiguration costs. Note that each problem setting has a different reward scale, since the latencies differ when we have more replicas in the system, or the episode is longer. Therefore, only the learning trend but not the absolute reward values are comparable across experiments.

the substantially larger action space and the need for long-term planning. Nevertheless, the policy continues to improve throughout training and ultimately reaches a stable high-reward regime. Although fewer training iterations are shown, each update yields larger gains, as long episodes provide more informative trajectories per training step.

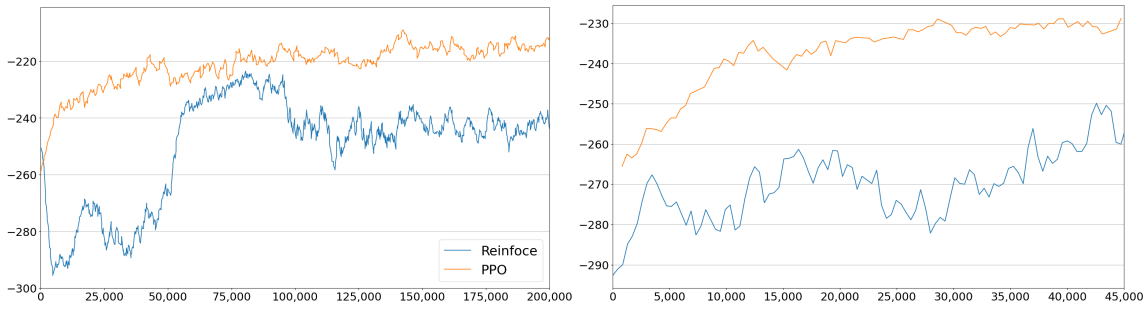
Overall, these results demonstrate that the proposed learning-based approach scales with both temporal horizon and combinatorial complexity. While larger and longer-horizon problems naturally require more training effort, the policy consistently converges to effective reconfiguration strategies. For clarity, training curves are truncated once rewards stabilize. Since longer horizons and larger location sets incur significantly higher simulation cost per episode, the number of training iterations is not directly comparable across scenarios in terms of wall-clock time.

### 6.4 Ablation studies

We perform a series of ablation studies to justify key design choices in our learning-based optimization approach. We analyze the impact of the learning algorithm on training stability and performance. Unless stated otherwise, all ablation experiments use the same simulation setup, reward definition, and state representation. Specifically, we used the 40-step and 8-location setup to allow for an efficient comparison between approaches.

**Learning Algorithm Ablation** Dynamic replica placement poses several challenges for reinforcement learning, including long decision horizons, delayed rewards, stochastic transitions, and a combinatorial action space. To motivate our choice of PPO, we compare it against the standard on-policy alternative, the vanilla REINFORCE algorithm [25].

Across all experiments, REINFORCE exhibits substantially higher variance in its training reward trajectory. In the short-horizon setting with 40 steps and 8 locations (Figure 3a), REINFORCE




(a) 40 steps, 8 locations

(b) 400 steps, 8 locations

Fig. 3: PPO vs. REINFORCE average reward (vertical axis) per episode over training steps (horizontal axis).

shows an initial upward learning trend, but quickly plateaus at a significantly lower average reward than PPO. While some learning is evident, convergence occurs early and remains far from the performance achieved by PPO. This behavior becomes even more evident in the larger problem setting. In the medium-horizon scenario with 400 steps and 8 locations (Figure 3b), REINFORCE shows only a small improvement over time and fails to make sustained progress. The same qualitative pattern was observed in the long-horizon setting (1500 steps, 15 locations), where learning effectively stalled. This behavior is expected: REINFORCE relies on empirical returns for gradient estimation, whose variance grows rapidly with episode length. In the absence of a learned baseline and under long temporal horizons, policy updates become dominated by noisy return signals, leading to unstable and overly aggressive updates that hinder long-term credit assignment. Reward normalization does not lead to measurable improvements, indicating that variance, and not scale, is the primary limiting factor.

**State Representation Ablation** We further ablate the state representation to analyze the impact of temporal modeling choices on performance and computational efficiency. We compare two architectures. In the first, temporal aggregation is applied to the full set of node embeddings (clients and locations) using an LSTM, followed by spatial message passing to derive decision-relevant location embeddings. In the second, spatial message passing is applied first, and temporal aggregation is restricted to the history of replica location embeddings, while client nodes are treated as transient and only processed spatially (as described in Section 5.3). Across all scenarios, we observe no meaningful difference in convergence behavior or final performance: both variants learn policies with comparable average latency and reconfiguration cost. However, the computational requirements differ substantially. Restricting temporal aggregation to replica locations reduces the number of sequences processed by the LSTM from $|L| + |C_t|$ to $|L|$. In our experiments, this corresponds to a reduction from $8 + 512$ to 8 nodes in the and medium-horizon scenarios, and from $15 + 1024$ to 15 nodes in the long-horizon scenario. Since client nodes are highly dynamic and appear and disappear over time, applying temporal aggregation to clients additionally increases sequence variability and memory requirements without improving decision quality.

Table 1: Average evaluation reward across different time horizons. Relative improvement of the learned PPO policy compared to the greedy Fluidity baseline is shown in brackets.

| Scenario | Static Placement | Greedy Fluidity | Learned PPO Policy |
|---|---|---|---|
| Short Horizon | -320.63 | -236.31 | -215.44 ( +8.9% ) |
| Medium Horizon | -318.18 | -270.43 | -238.57 ( +11.5% ) |
| Long Horizon | -407.24 | -363.70 | -338.69 ( +6.9% ) |

### 6.5 Comparison to the State-of-the-Art

We evaluate the trained policies on separate test episodes and compare three strategies: (i) static replica placement, (ii) the greedy Fluidity baseline, and (iii) the learned PPO policy. Figure 4 and Table 4 show the system latency over time for representative evaluation episodes in all three scenarios. Across all settings, static placement performs poorly once client demand shifts away from the initial configuration. The greedy Fluidity baseline adapts to changing demand but exhibits delayed reactions and oscillatory behavior due to its myopic, one-step lookahead nature. In contrast, the learned policy consistently achieves lower average latency and smoother adaptation. This behavior is observed in the short , medium, and long horizon scenarios. These results are consistent with prior findings in Fluidity when comparing against static placement, and further demonstrate that learning-based optimization can outperform the state-of-the-art greedy reconfiguration strategy. All evaluation episodes are executed after training convergence and include demand patterns not encountered during training to test robust generalization of the learned policy.

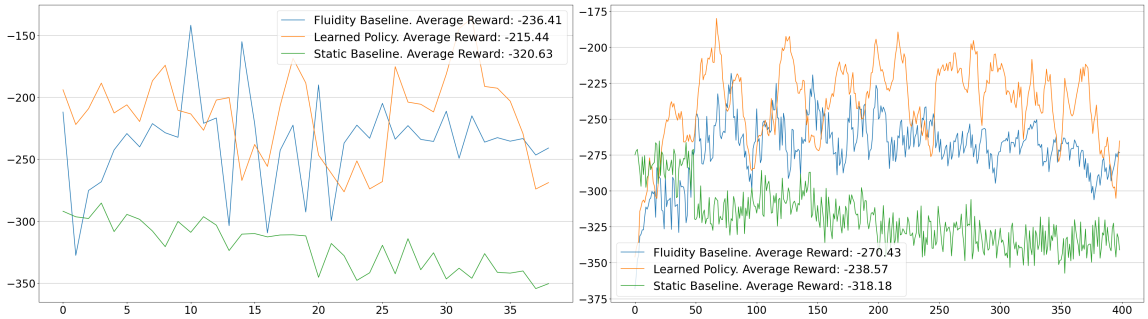
### 6.6 Relocation Action Structure

To better understand the decision making, we generate a heatmap of relocation actions selected by the converged policy, where rows denote removed replica locations and columns denote added locations. The results for the short-horizon experiment can be seen in Figure 5. The strong diagonal indicates that the policy frequently selects no-op actions, reflecting a learned preference for configuration stability once a favorable placement is reached. The heatmap of greedy baseline shows a continous reconfiguration in response to instantaneous measurements, the learned policy performs fewer but more targeted relocations, highlighting its ability to internalize long-term trade-offs between latency reduction and reconfiguration cost. A further analysis on the other problem instances can be found in Appendix G.

## 7 Conclusion

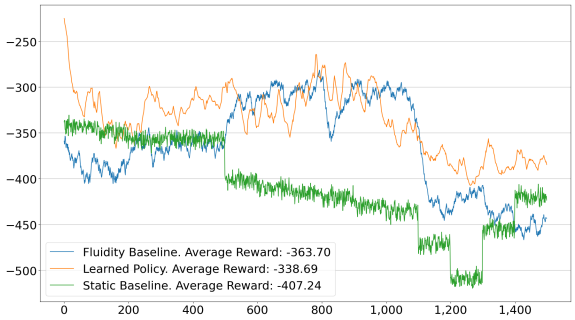
This work studied dynamic replica placement in replicated state machines from an optimization perspective and showed that reinforcement learning provides a scalable alternative to greedy online reconfiguration strategies. By training a policy offline on simulated system trajectories, we obtain constant-time decision making at runtime while explicitly accounting for long-term trade-offs.

Across all experimental scenarios, the learned policy consistently reduces average system latency compared to both static placement and the greedy Fluidity baseline. The performance gap increases with longer horizons and larger candidate sets, highlighting a central advantage of reinforcement learning: effective long-term credit assignment. Unlike myopic heuristics that optimize




(a) 40 steps, 8 locations

(b) 400 steps, 8 locations




(c) 1500 steps, 15 locations

Fig. 4: Step reward (vertical axis) during evaluation (horizontal axis: episode step); larger is better. The evaluation episodes are run after convergence of the respective experiment. For visual clarity, just one comparative run across multiple baselines is used.

only the current system state, the learned policy anticipates demand shifts and avoids unnecessary reconfigurations, resulting in smoother adaptation and lower cumulative latency.

The results further demonstrate that DRL can implicitly learn compact, decision-relevant representations of high-dimensional and noisy system states. Operating on spatio-temporal graph representations allows the policy to capture global client-replica interactions without handcrafted features or explicit scenario enumeration, which is particularly important in wide-area replicated systems where performance depends on collective communication effects.

From a systems perspective, the proposed approach complements existing dynamic replica management techniques. While greedy strategies such as Fluidity remain strong baselines due to their simplicity, our results show that learning-based optimization can surpass these methods once sufficient offline training is available. Since inference incurs minimal runtime overhead, the approach is well-suited for latency-sensitive production environments.

For future work, we propose jointly training policies across multiple system sizes and demand regimes to improve generalization, as well as addressing the simulation bottleneck. In particular, model-based reinforcement learning offers a promising direction to approximate simulator dynamics and accelerate training.


Fig. 5: Relocation action distribution heatmap. Larger values indicate that this action tuple was selected more often. The data of both figures was collected over 10 episodes after policy convergence on the short-horizon (40 steps, 8 locations) setup.

## References

[1] Zeinab Azarmand and Ensiyeh Neishabouri. "Location Allocation Problem". In: *Facility Location: Concepts, Models, Algorithms and Case Studies*. Ed. by Reza Zanjirani Farahani and Masoud Hekmatfar. Heidelberg: Physica-Verlag HD, 2009, pp. 93–109.

[2] Irwan Bello et al. *Neural Combinatorial Optimization with Reinforcement Learning*. 2017. URL: https://openreview.net/forum?id=rJY3vK9eg.

[3] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43.

[4] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG]. URL: https://arxiv.org/abs/1606.01540.

[5] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". en. In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 2002), pp. 398–461.

[6] Gabriele Corso et al. "Graph neural networks". en. In: *Nat. Rev. Methods Primers* 4.1 (Mar. 2024).

[7] David Eisenstat, Claire Mathieu, and Nicolas Schabanel. "Facility Location in Evolving Metrics". In: vol. 8573. Mar. 2014.

[8] Reza Zanjirani Farahani, Maryam Abedian, and Sara Sharahi. "Dynamic Facility Location Problem". In: *Facility Location: Concepts, Models, Algorithms and Case Studies*. Ed. by Reza Zanjirani Farahani and Masoud Hekmatfar. Heidelberg: Physica-Verlag HD, 2009, pp. 347–372.

[9] Elena Fernandez and Mercedes Landete. "Fixed-Charge Facility Location Problems". In: *Location Science*. Ed. by Gilbert Laporte, Stefan Nickel, and Francisco Saldanha da Gama. Springer Books. Springer, Sept. 2015. Chap. 0, pp. 47–77.

[10] Wenxuan Guo et al. "Unified and Generalizable Reinforcement Learning for Facility Location Problems on Graphs". In: *Proceedings of the ACM on Web Conference 2025*. WWW '25. Sydney NSW, Australia: Association for Computing Machinery, 2025, pp. 1182–1195.

[11] William L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (2020), pp. 62–64.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[13] Johannes Köstler et al. "Fluidity: Location-Awareness in Replicated State Machines". In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. SAC '23. Tallinn, Estonia: Association for Computing Machinery, 2023, pp. 192–201.

[14] Leslie Lamport. "The part-time parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169.

[15] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-Attention Graph Pooling". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 3734–3743.

[16] Adrian Ly et al. "Elastic step DQN: A novel multi-step algorithm to alleviate overestimation in Deep Q-Networks". In: *Neurocomputing* 576 (2024), p. 127170.

[17] Zoubir Mammeri. "Reinforcement Learning Based Routing in Networks: Review and Classification of Approaches". In: *IEEE Access* 7 (2019), pp. 55916–55950.

[18] Nina Mazyavkina et al. "Reinforcement learning for combinatorial optimization: A survey". In: *Computers & Operations Research* 134 (2021), p. 105400.

[19] Nenad Mladenovic et al. "The p-median problem: A survey of metaheuristic approaches". In: *European Journal of Operational Research* 179.3 (2007), pp. 927–939. ISSN: 0377-2217.

[20] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[21] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. "Deep Reinforcement Learning: An Overview". In: *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*. Ed. by Yaxin Bi, Supriya Kapoor, and Rahul Bhatia. Cham: Springer International Publishing, 2018, pp. 426–440.

[22] Karl E. Nelson and Martin K. Scherer. *JPype*. June 2020. URL: https://www.osti.gov/biblio/code-46107.

[23] Charles S Revelle, Horst A Eiselt, and Mark S Daskin. "A bibliography for some fundamental problem categories in discrete location science". In: *European journal of operational research* 184.3 (2008), pp. 817–848.

[24] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018, pp. 58–60. URL: http://incompleteideas.net/book/the-book-2nd.html.

[26] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[27] Ching Pui Wan, Tung Li, and Jason Min Wang. *RLOR: A Flexible Framework of Deep Reinforcement Learning for Operation Research*. 2023. arXiv: 2303.13117 [math.OC].

[28] Chenguang Wang et al. "Solving uncapacitated P-Median problem with reinforcement learning assisted by graph attention networks". en. In: *Appl. Intell.* 53.2 (Jan. 2023), pp. 2010–2025.

[29] Qi Wang and Chunlei Tang. "Deep reinforcement learning for transportation network combinatorial optimization: A survey". In: *Knowledge-Based Systems* 233 (2021), p. 107526.

# A Latency Computation in Replicated State Machines

This appendix provides additional technical details on how client request latency is computed in replicated state machines, following the methodology used in Fluidity [13]. The purpose of this appendix is to support the latency abstraction used in the main text by making explicit how individual communication delays contribute to the overall request latency.

## A.1 Dynamic Client Evolution

Figure 6 shows an example of the dynamic, global, client movement in a distributed network system.

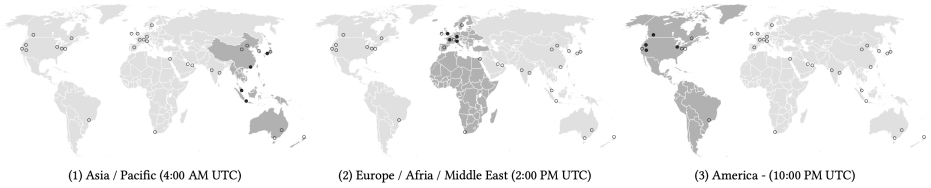



Fig. 6: A system of four replicas (filled circles) is dynamically migrated across datacenters (circles) around the globe. For the indicated time, the respective images show (shaded in darker gray) the regions with the highest client activity, which results in the ideal placement of the replicas for that time [13]

## A.2 Fluidity Location Graph Development

Figure 7 shows how the location graph develops over time as more information arrives.

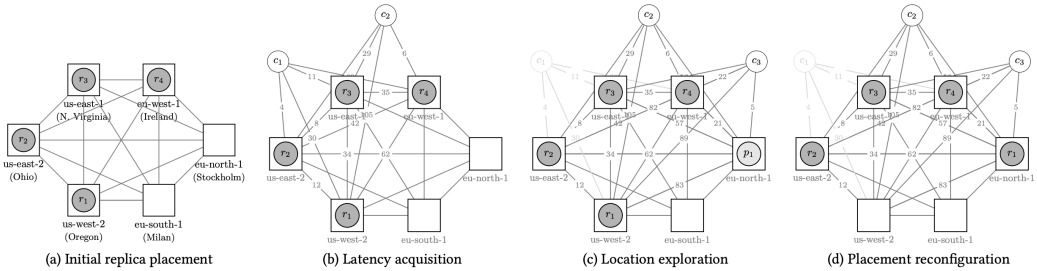



Fig. 7: Fluidity Location Graph. The location graph evolves over time as more information is gathered. It keeps track of historic latencies and current client locations and requests.

### A.3 Client Request Execution in RSMs

In a replicated state machine, a single client request is not processed by a single server in isolation. Instead, it is disseminated to all active replicas, which coordinate via a consensus protocol to agree on a global execution order. This coordination involves several protocol phases and message exchanges between replicas.

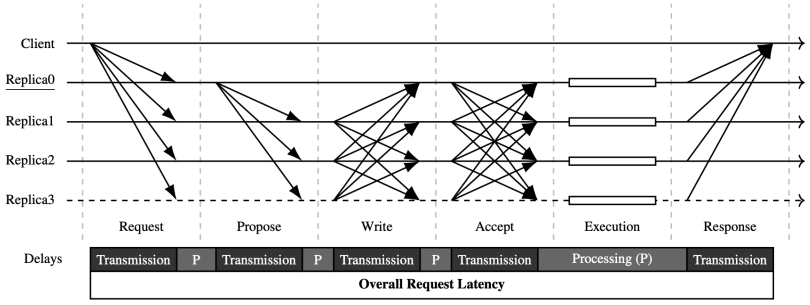



Fig. 8: Demonstration of overall request latency for a client. Each client sends the request to the leading replica, which sends it to all other replicas and executes the consensus protocol.

Figure 8 illustrates the message flow of a single client request in a replicated state machine, as presented in Fluidity. In addition to the client request and response messages, the execution involves multiple agreement steps, each requiring one or more inter-replica message exchanges. Each protocol step contributes one or more communication delays. These delays include client-to-replica transmission, replica-to-replica communication during agreement, and replica-to-client response transmission. As a result, the end-to-end request latency experienced by a client is the aggregation of multiple message transmission delays rather than a single round-trip time.

### A.4 Aggregation of Latency Components

The following discussion describes the latency incurred along a *single execution path* of a client request through the replicated state machine protocol. It serves to illustrate how individual communication delays contribute to request completion time, rather than to define an additive latency model over replica locations.

Let a client request traverse a sequence of protocol phases indexed by $k = 1, \ldots, K$. Each phase incurs one or more network transmission delays, denoted by $\delta_k$. The overall request latency $L$ along this execution path can be expressed as

$$L = \sum_{k=1}^{K} \delta_k + \sum_{k=1}^{K} p_k,$$

where $p_k$ denotes local processing delays at replicas.

In wide-area deployments, network transmission delays dominate processing delays. Following Fluidity and related work, processing delays are therefore neglected and the overall request latency

is approximated as

$$L \approx \sum_{k=1}^{K} \delta_k.$$

This approximation captures the dominant performance effects while significantly simplifying measurement and modeling at the protocol level.

In the main text, we abstract over individual execution paths by modeling system latency as a set-dependent function $D(L_t^+, S_t)$. This function represents the effective end-to-end latency induced by quorum-based execution with active replica set $L_t^+$ under system state $S_t$, implicitly accounting for the fastest quorum, protocol phases, and stochastic network effects. The present discussion motivates this abstraction by showing how low-level communication delays contribute to request latency along any given execution path.

### A.5 Location Graph Representation

To reason about latency systematically, Fluidity represents replicas and clients as nodes in a location graph. Nodes correspond to geographical locations, and edges are weighted by measured network transmission delays between the corresponding endpoints. Replica locations are drawn from a fixed and predefined set of data center regions, while client nodes appear and disappear dynamically as clients join and leave the system.

At system startup, the graph contains only replica locations and the initial replica placement. Over time, client nodes are added, latencies are measured, and additional passive replicas may be deployed to explore alternative placements. When a placement with lower aggregated request latency is identified, replicas may be migrated accordingly.

### A.6 Latency Measurement and Trust Assumptions

Latencies are acquired during normal protocol execution by piggybacking measurement probes onto existing protocol messages. This allows continuous latency observation without introducing excessive measurement traffic.

Under a Byzantine fault model, latency measurements reported by other nodes cannot be trusted blindly. Fluidity therefore relies on symmetry assumptions of network links and redundant measurements to sanitize observed latencies. For example, for each pair of nodes, measurements in both directions are compared, and pessimistic estimates are used to prevent malicious nodes from appearing artificially faster.

A detailed discussion of the measurement, sanitation, and fault-tolerance mechanisms is provided in the work of Koestler et al. [13]. These mechanisms ensure that latency estimates are sufficiently accurate and robust for decision making, while preserving the safety and liveness guarantees of the replicated state machine.

## B Simulation

### B.1 Demand and Latency Model

Client demand evolves as a non-stationary stochastic process over a fixed set of geographical regions. At each time step $t$, a total request volume of $Q = 100{,}000$ requests is generated and distributed

among active clients. Request volumes are sampled from a Dirichlet distribution

$$(q_{1,t}, \ldots, q_{|C_t|,t}) \sim \text{Dirichlet}(\alpha_t),$$

where the concentration parameters $\alpha_t$ depend on whether a client is located in a demand peak region at time $t$. This induces spatially concentrated demand with shifting peaks over time, modeling diurnal and regional workload dynamics.

Client request latency is computed using the replicated state machine abstraction described in Section 2 and Appendix A. For a given time step $t$ and active replica configuration $L_t^+$, the simulator executes the replicated protocol and aggregates client–replica and inter-replica communication delays into an effective end-to-end latency $L_{c,t}$ for each client $c \in C_t$.

The realized system latency at time $t$ is then computed internally by the simulator as the request-volume-weighted mean

$$\bar{L}_t = \frac{1}{Q} \sum_{c \in C_t} q_{c,t} L_{c,t}.$$

This quantity represents a single realization of the set-dependent latency function $D(L_t^+, S_t)$ used in Sections 4 and 5 and serves as the primary evaluation metric in all experiments.

In addition to latency, replica reconfiguration incurs a fixed cost $k_r = 30$ per activated replica. This choice introduces a meaningful trade-off between reducing latency through reconfiguration and maintaining configuration stability and is consistent with the cost model and empirical observations reported in the Fluidity work [13]. For a detailed understanding, see the implementation under `https://github.com/codebold/emusphere/tree/main/simulator-xmr`.

### B.2 Simulation-Environment Coupling

From an implementation perspective, we build an executable JAR file that runs the simulation with a runtime configuration file, with which we can set the experiment specifics (episode steps, number of locations, etc.). Using `JPype` for cross-language interaction [22], we instantiate Java-based classes and call their methods through a Python wrapper. This allows us to integrate the Java simulation in an OpenAI-Gym compatible wrapper with a graph observation space. This setup makes the experiments fully reproducible as long as the executable JAR and run configurations are provided. The simulation implementation can be found here: `https://github.com/codebold/emusphere/tree/main/simulation-fluidity`

## C Graph Neural Network Architecture

Our GNN is based on spatial message passing between nodes and temporal aggregation between the decision node embedding from previous states.

## D Training Objective

This appendix provides the explicit formulation of the Proximal Policy Optimization (PPO) objective used to train the policy and value function.

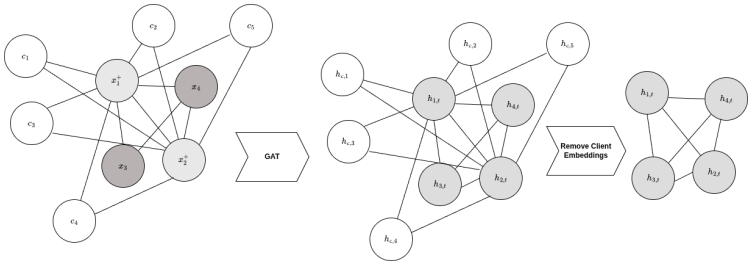



Fig. 9: GAT Forward Pass. Client nodes $c$ and data center nodes $x$ where $x^+$ denotes an active data center, are passed through the GAT. After the message passing between all nodes, the client embeddings are removed for further processing.

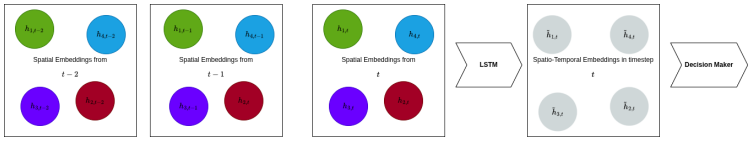



Fig. 10: Temporal Embedding aggregation. After the GAT forward pass, the spatial data center embeddings $h$ are aggregated with their respective spatial embedding from the last timesteps to understand temporal dependencies. This aggregation results in the final spatio-temporal embeddings $\tilde{h}$ which are the basis for the final decision maker.

Let $\pi_\theta(a_t \mid s_t)$ denote the current policy and $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ the policy used to generate the training trajectories. The PPO objective is defined as

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

where

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

is the likelihood ratio, $\epsilon > 0$ is the clipping parameter, and $\hat{A}_t$ denotes an estimate of the advantage function.

The advantage is computed using the learned value function $V_\phi(s_t)$ and the observed rewards,

$$\hat{A}_t = \sum_{k=0}^{K-1} \gamma^k r(s_{t+k}, a_{t+k}) - V_\phi(s_t),$$

where $\gamma \in (0,1)$ is the discount factor. The value function is trained by minimizing the squared error

$$\mathcal{L}_{\text{V}}(\phi) = \mathbb{E}_t\big[(V_\phi(s_t) - \hat{V}_t)^2\big],$$

with $\hat{V}_t$ denoting the empirical return.

The full objective additionally includes an entropy regularization term to encourage exploration. After training, the policy $\pi_\theta$ is deployed to compute replica reconfiguration decisions via a single forward pass.

### D.1 Hyperparameter Selection

The PPO hyperparameters were chosen based on standard recommendations in the literature and refined through empirical tuning on the simulation environment. Table 4 summarizes the values used across all experiments.

Table 2: PPO hyperparameters used for training.

| Hyperparameter | Value |
|---|---|
| Discount factor $\gamma$ | 0.99 |
| Clipping parameter $\epsilon$ | 0.2 |
| Policy learning rate | $3 \times 10^{-4}$ |
| Value function learning rate | $1 \times 10^{-3}$ |
| Entropy coefficient | 0.01 |
| Number of PPO epochs | 50 |
| GAE parameter $\lambda$ | 0.95 |

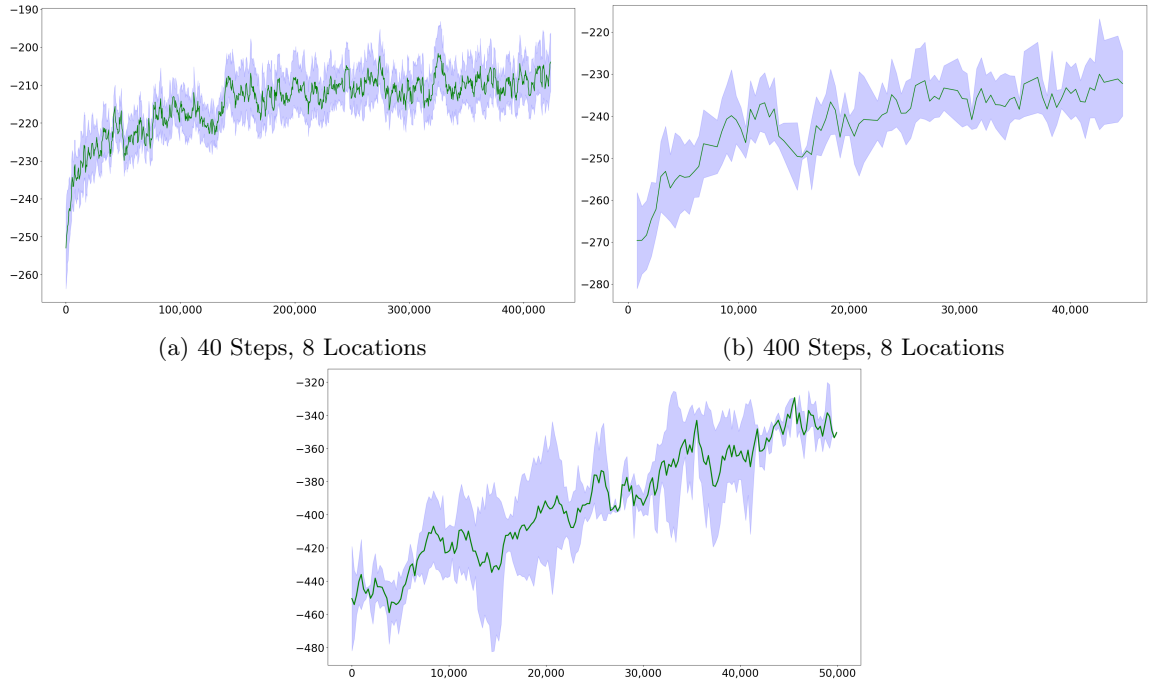
These values were kept fixed across all experimental settings to ensure comparability. We found PPO to be robust to moderate variations of these parameters, and no environment-specific tuning was required beyond adjusting the rollout length to match the temporal scale of demand dynamics. Training frequency was set two training steps per episode while the epoch and batch size depends on the problem size:

Table 3: PPO batch size used for training.

| Problem Size | Batch Size |
|---|---|
| Small Horizon (40 Steps) | 32 |
| Medium Horizon (400 Steps) | 256 |
| Long Horizon | 512 |

Table 4: PPO number of epochs per training step

| Problem Size | Batch Size |
|---|---|
| Small Horizon (40 Steps) | 50 |
| Medium Horizon (400 Steps) | 100 |
| Long Horizon | 200 |




(a) 40 Steps, 8 Locations

(b) 400 Steps, 8 Locations

(c) 1500 Steps, 15 Locations

Fig. 11: Training reward (vertical axis) over training steps (horizontal axis) for the three experimental scenarios. Shaded blue area is the standard deviation. Green line is the mean over three seeds. Higher reward corresponds to lower average system latency and reduced reconfiguration costs. Note that each problem setting has a different reward scale, since the latencies differ when we have more replicas in the system, or the episode is longer. Therefore, only the learning trend but not the absolute reward values are comparable between experiments.

# E Training Dynamics PPO

# F Ablations

## F.1 PPO vs. DQN

We compare PPO against the off-policy Deep Q-Network (DQN) [20]. In contrast to PPO, DQN fails to converge in all evaluated scenarios. Figure 12 shows the temporal-difference (TD) loss during training, which initially decreases but then steadily increases and remains highly unstable, indicating a breakdown of value learning. Replica reconfiguration decisions correspond to structured, combinatorial actions, and rewards are delayed over long horizons. In this setting, small approximation errors in Q-values can accumulate over time and lead to large policy errors. The upward trend of the TD loss reflects a growing one-step Bellman error [25, 20] and is characteristic of Q-value overestimation in value-based methods [16]. As a result, the learned value function does not converge to a consistent estimate and fails to provide a meaningful learning signal. While several

extensions exist to mitigate Q-value overestimation [16], we did not pursue them further given the clear instability observed and the superior performance and robustness of PPO in this problem setting.

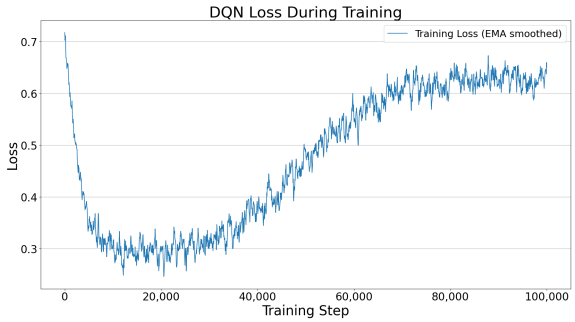



Fig. 12: DQN Loss (One-step TD error) during training of the 40 steps 8 location setup.

### F.2 State Representation Ablation

As shown in Figure 13, no real difference can be observed between the full and reduced state represenation while the reduced state representation shows clear computational benefits.


Fig. 13: Average Training Reward Comparison Between Full State Space (LSTM over all client and location nodes followed by a GAT spatial forward pass) and Reduced State Space (GAT spatial forward pass on all nodes and LSTM only on locations).

## G Action Distribution Heatmap

A similar "diagonal-peak" pattern can be seen in the larger problem setting albeit a bit less nuanced. This is expected since the strong visual effects on the diagonal diminish with more actions in the heatmaps through longer episodes and more locations.

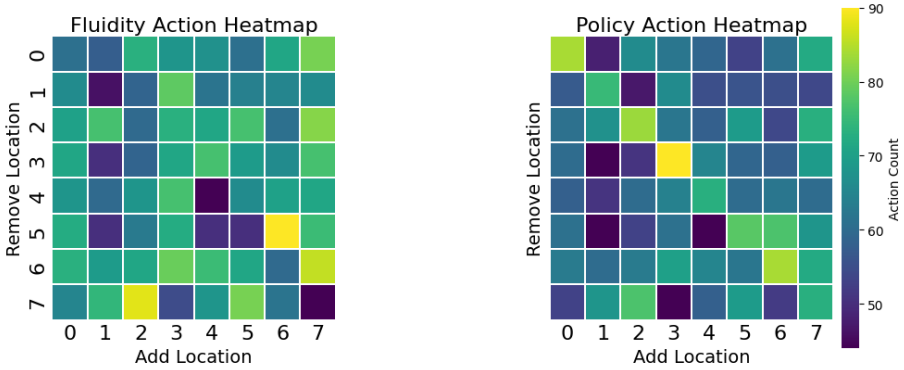



Fig. 14: Relocation action distribution heatmap. Larger values indicate that this action tuple was selected more often. The data of both figures was collected over 10 episodes after policy convergence on the medium-horizon (400 steps, 8 locations) setup.

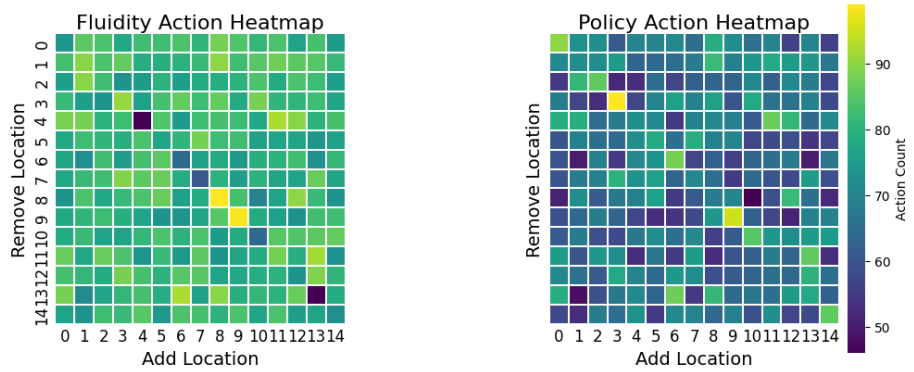



Fig. 15: Relocation action distribution heatmap. Larger values indicate that this action tuple was selected more often. The data of both figures was collected over 10 episodes after policy convergence on the long-horizon (1500 steps, 15 locations) setup.