AUTOSCALING BY IMAGINATION: GRAPH WORLD MODELS FOR SAFE DISTRIBUTED MACHINE LEARNING SERVICES

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern machine learning services are increasingly deployed as distributed microservice systems, where resource allocation decisions must balance latency, reliability, and cost. In practice, however, autoscaling remains largely reactive: controllers add replicas only after the system is already under pressure, while safer deployments often rely on static over-allocation of resources. Existing autoscalers do not learn a model of how scaling actions propagate through the service graph, and therefore cannot anticipate how a decision will affect future load, latency, cost, and service-level violations before it is executed. To fill this research gap, we present NetDream, a graph world model for safe autoscaling of distributed machine learning services. NetDream learns action-conditioned dynamics over a microservice graph, predicting how load, latency, replicas, and violation risk evolve under candidate scaling actions. At runtime, NetDream uses this learned model for imagination-based planning: it rolls out multiple future scaling trajectories, filters out plans predicted to violate service-level constraints, and executes the first action of the best safe plan. This turns autoscaling from threshold-based reaction into topology-aware planning over a learned model of the system. We evaluate NetDream on a production-grade AWS Kubernetes deployment of Online Boutique, an 11-service, 14-edge microservice graph with a 66-dimensional system state and 48,000 collected transitions, under steady, variable, bursty, and flash-crowd traffic. NetDream is the most cost-efficient safe autoscaler among all evaluated methods. Each marginal replica-step it spends buys safety 2.7 times more cost-efficiently than the best static over-provisioning strategy. On variable-traffic workloads specifically, it achieves about 5 times fewer violations than reactive autoscaling while using only about half the cost of the strongest over-provisioning baseline. Code, trained models, and infrastructure manifests are released anonymously at `https://anonymous.4open.science/r/netdream-FCF9`.

1 INTRODUCTION

Modern machine learning services are increasingly deployed as distributed microservice systems (Qiu et al., 2020; Alibaba Group, 2022). A single user request may pass through many interacting components: frontend services, retrieval services, ranking services, payment or logging services, caches, databases, and model serving endpoints. Keeping such systems reliable requires continuous resource allocation. Too few replicas lead to latency spikes and service-level violations; too many replicas waste compute. Autoscaling is therefore a core control problem for distributed machine learning services: decide where to add or remove capacity, under changing demand, while balancing latency, reliability, and cost.

Production autoscaling, however, remains largely reactive. Standard controllers such as the Kubernetes Horizontal Pod Autoscaler (HPA) (Rzadca et al., 2020; Google Cloud, 2025; Amazon Web Services, 2025) scale only after utilization or latency signals indicate that the system is already under pressure. Recent extensions augment HPA with short-horizon forecasting or adaptive thresholds (Pozdniakova et al., 2024; Ahmad et al., 2024; Santos et al., 2025), yet still trigger scaling rules from a forecast metric rather than from a simulated future trajectory. Safer deployments often compensate by keeping extra replicas alive at all times. This creates a familiar but unsatisfactory trade-off: reactive

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.

scaling is cheap but late, while static over-allocation is safer but expensive. The problem is not only that existing controllers use simple thresholds. More fundamentally, they act without a learned model of the system: they do not know how a scaling action will propagate through the service graph, how it will change downstream load and latency, or whether it will create a future service-level violation. We refer to this missing capability as *imagination*: the ability of an autoscaler to evaluate counterfactual scaling decisions before they are applied to the real cluster

In this paper, we present NetDream, a graph world model for safe autoscaling of distributed machine learning services. NetDream represents a microservice application as a graph, where nodes are services and edges are service-call dependencies. While prior work has applied graph neural networks to predict tail latency (Park et al., 2024) or to learn model-free actor–critic policies for autoscaling (Ho et al., 2025), and digital-twin approaches (Soulé et al., 2025) have built unstructured neural transition models of clusters, NetDream is, to our knowledge, the first to combine a graph-structured dynamics model with imagination-based planning and an explicit safety filter. It learns action-conditioned dynamics over this graph: given the current service states and a candidate scaling action, the model predicts how load, latency, replicas, reward, and violation risk will evolve. At runtime, NetDream uses this learned model inside an imagination-based planner. The planner rolls out multiple future scaling trajectories, rejects those predicted to violate service-level constraints, and executes only the first action of the best safe plan. In this way, NetDream turns autoscaling from threshold-based reaction into topology-aware planning over a learned model of the system.

This formulation is operationally practical. NetDream does not require an online reinforcement-learning agent to explore risky actions in the real cluster (Qiu et al., 2023; Zhang et al., 2025). Instead, it learns from logged transitions and uses the learned world model as a simulator for planning. The safety mechanism is also explicit: candidate futures whose predicted violation risk exceeds a threshold are filtered out before execution. This makes the controller closer to how human operators reason about infrastructure: compare possible actions, reject unsafe ones, and choose the lowest-cost safe plan.

We evaluate NetDream on a production-grade AWS Kubernetes deployment of Online Boutique (Google Cloud Platform, 2023), an 11-service, 14-edge microservice graph with a 66-dimensional system state and 48,000 collected transitions. The autoscaler is stress-tested under four workload regimes (steady traffic, variable demand, bursty spikes, and flash crowds) generated with Locust (Locust Contributors, 2024) and replayed from production microservice traces (Alibaba Group, 2022). We compare against standard reactive autoscaling, static over-provisioning strategies, random scaling, model-free reinforcement learning, and an ablation of NetDream without the safety filter. Our results show that NetDream provides a cost-efficient path to safe autoscaling. NetDream reduces service-level violations at 2.10 cost per violation avoided, compared with 5.65 for the best strategy, making it 2.7 times more cost-efficient at buying safety. On variable-traffic workloads, NetDream achieves about 5 times fewer violations than HPA while using about half the cost of the strongest over-provisioning baseline. Overall, this paper makes three contributions:

- Autoscaling as world-model planning. We recast distributed ML autoscaling as a model-based planning problem over a service graph. NetDream learns action-conditioned graph dynamics and uses them to imagine the multi-step effects of scaling decisions before they reach the real cluster.

- Safety through imagined futures. We introduce an explicit imagination-based safety filter for autoscaling. Rather than optimizing reward alone, NetDream rolls out candidate scaling plans inside the learned world model and executes only plans whose predicted futures remain within service-level constraints.

- Real-cluster evidence for cost-efficient safety. We validate NetDream on a production-grade Kubernetes deployment of Online Boutique. NetDream achieves a stronger cost–safety frontier than reactive HPA, static over-provisioning, model-free RL, and unsafe planning ablations, showing that learned graph imagination can buy safety more efficiently than adding replicas everywhere.

2 Related Works

Learning-based autoscaling has moved beyond hand-designed rules, but most methods still fall into two categories: predict a metric and react, or learn a policy directly. GNN-based autoscalers exploit the fact that microservice systems are graphs. GRAF predicts tail latency with a GNN and uses the prediction for proactive SLO-aware allocation (Park et al., 2024). pHPA continuously trains a GNN to infer tail latency along microservice chains (Choi et al., 2021). DeepScaler combines spatio-temporal GNNs with adaptive graph learning for holistic autoscaling (Meng et al., 2023); Graph-PHPA combines LSTM and GNN modules for proactive horizontal scaling (Nguyen et al., 2022); AGQ uses dense spatio-temporal GNNs to capture indirect dependencies (Liang et al., 2026); and USRFNet learns unified system representations for microservice tail-latency prediction (Qian et al., 2025). These systems show that topology-aware representations matter, but their control logic remains prediction-driven rather than imagination-driven: the GNN predicts a future metric, and a separate heuristic or policy decides how to scale. Reinforcement-learning autoscalers instead learn control policies directly. Prior work has explored DQN, PPO, TD3, and recurrent reinforcement learning for microservice and serverless autoscaling (Abdel Khaleq & Ra, 2023; Qiu et al., 2023; Zhang et al., 2025; Bai et al., 2024; Agarwal et al., 2024). Related systems address the SLO–cost trade-off through bi-level resource control, utilization-aware autoscaling, and explicit SLO–cost-driven policies (Wang et al., 2024a;b; Marchese & Tomarchio, 2025). GraphPilot is closest in spirit among GNN-based controllers: it combines a Temporal Graph Network encoder with an actor–critic policy for Kubernetes autoscaling (Ho et al., 2025). However, these approaches learn a policy, not a graph world model used for explicit planning. They do not roll out candidate action sequences in a learned dynamics model and reject unsafe futures before execution. Digital-twin approaches move closer to model-based control. KARMA builds a neural transition model of the cluster and trains multi-agent policies in simulation for resilience against failures such as DDoS attacks and pod crashes (Soulé et al., 2025). However, its transition model is not graph-structured, and its objective is adversarial resilience rather than cost–SLO autoscaling on standard microservice workloads. NetDream differs in three ways: it learns topology-aware graph dynamics, uses the learned model online for model-predictive planning, and applies an explicit safety filter that rejects candidate action sequences predicted to violate service-level constraints. In summary, prior Kubernetes autoscalers either react to observed pressure, forecast a metric and apply a rule, or learn a model-free policy. NetDream instead treats autoscaling as planning with a learned graph world model. This distinction is the source of its empirical advantage: the controller can compare multiple possible futures, reason over service dependencies, and avoid unsafe plans before they are applied to the real cluster.

3 Method

NetDream is built on one idea: safe autoscaling requires a model of the future. Instead of reacting to threshold crossings, NetDream learns a graph world model of a Kubernetes microservice application, uses it to imagine the consequences of candidate scaling actions, rejects unsafe futures, and executes the first action of the best safe plan (see Figure. 1). This section formalises the control problem (Section 3.1), specifies the graph world model (Section 3.2), describes imagination-based planning with the safety filter (Section 3.3), and summarises how the trained model is deployed on a real Kubernetes cluster (Section 3.4).

3.1 Problem Formulation

We represent a distributed machine learning service as a directed graph $\mathcal{G} = (V, E)$, where each node $i \in V$ is a microservice and each edge $(i, j) \in E$ is a service-call dependency. At control step t, service i has state

$$x_{t,i} = [\mathrm{CPU}_{t,i}, \mathrm{memory}_{t,i}, \mathrm{QPS}_{t,i}, \mathrm{latency}_{t,i}, \mathrm{error}_{t,i}, \mathrm{replicas}_{t,i}] \in \mathbb{R}^d.$$

The full cluster state is $X_t = [x_{t,1}, \ldots, x_{t,N}] \in \mathbb{R}^{N \times d}$.

The autoscaler chooses a per-service scaling action $A_t = [a_{t,1}, \ldots, a_{t,N}], a_{t,i} \in \{-1, 0, +1\}$, corresponding to scaling service i down, holding it fixed, or scaling it up by one replica. After the action is applied, the cluster evolves to X_{t+1}, receives reward r_t, and may incur a service-level violation $c_t \in \{0, 1\}$. The true transition process $X_{t+1} \sim T(X_t, A_t; \mathcal{G})$ is unknown and difficult to

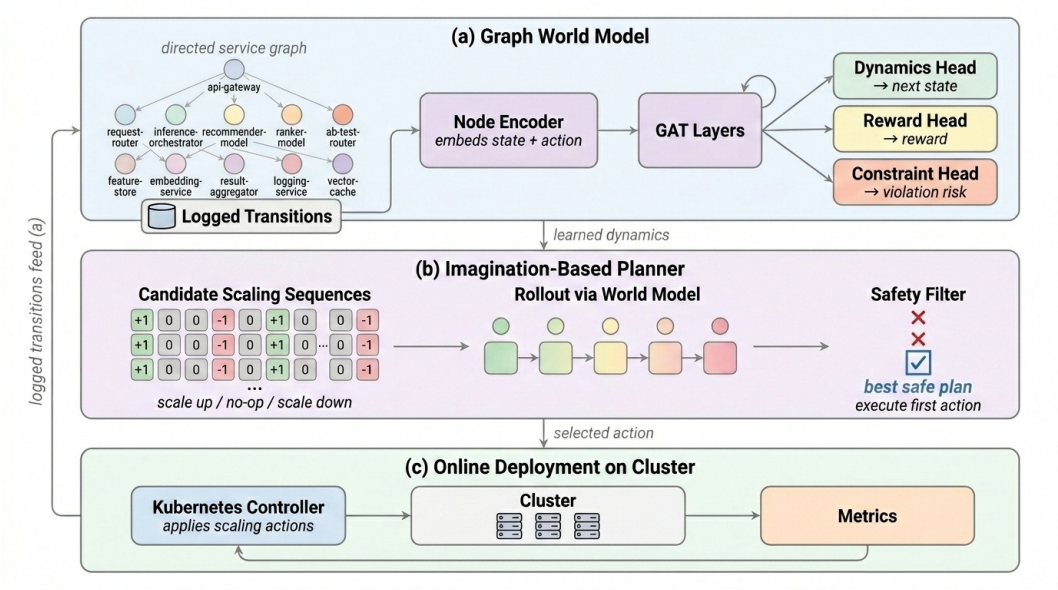


Figure 1: **NetDream for safe autoscaling of distributed machine learning services.** NetDream learns a graph world model of a distributed ML microservice application. Given the current service states and candidate scaling actions, the model predicts future state changes, reward, and service-level violation risk. A planning module rolls out multiple candidate action sequences in imagination, rejects unsafe futures through a learned safety filter, and executes the first action of the best safe plan.

model analytically because it depends on request routing, queueing, pod startup delays, workload shifts, and cross-service dependencies.

The objective is to reduce service-level violations while avoiding unnecessary replica cost. We use a reward that penalizes both resource usage and violations: $r_t = -\lambda_{\text{cost}}\text{Cost}(X_t) - \lambda_{\text{slo}}c_t, \text{Cost}(X_t) = \sum_{i=1}^{N} \text{replicas}_{t,i}$, where the violation indicator $c_t = \mathbb{1}[\exists i \in V : \text{latency}_{t+1,i} > \text{SLO}]$ flags whether the transition into X_{t+1} produced an SLO breach at any service. NetDream does not rely on reward optimization alone. It also learns an explicit constraint predictor and uses it during planning to discard unsafe futures before they are executed.

3.2 Graph World Model

NetDream learns an action-conditioned graph world model $f_\theta(\mathcal{G}, X_t, A_t) \to (\Delta\hat{X}_t, \hat{r}_t, \hat{c}_t)$, where $\Delta\hat{X}_t$ is the predicted state residual, $\hat{r}_t$ is the predicted reward, and $\hat{c}_t$ is the predicted probability of a service-level violation. The next state is predicted as $\hat{X}_{t+1} = X_t + \Delta\hat{X}_t$. Residual prediction makes the model focus on short-term changes rather than reconstructing absolute state, which is important for stable multi-step rollouts.

For each service, NetDream first encodes the current state and candidate scaling action. Because $a_{t,i} \in \{-1, 0, +1\}$ is categorical, we one-hot encode it before concatenation:

$$h_{t,i}^{(0)} = \phi_{\text{enc}}\left(\left[x_{t,i} \,\middle\|\, \text{onehot}(a_{t,i})\right]\right), \qquad \text{onehot}(a_{t,i}) \in \{0,1\}^3.$$

This makes the model counterfactual: it can predict what would happen under different possible scaling actions, not merely forecast the next state under the current policy.

NetDream then applies L layers of graph attention over the service-call topology. The directed call graph $\mathcal{G}$ is symmetrized for message passing so that downstream load and upstream backpressure can both flow along an edge; we write $\mathcal{N}(i)$ for the resulting undirected neighbourhood of service i.

$$\tilde{h}_{t,i}^{(\ell+1)} = \text{GAT}^{(\ell)}\left(h_{t,i}^{(\ell)}, \{h_{t,j}^{(\ell)} : j \in \mathcal{N}(i)\}\right), \qquad h_{t,i}^{(\ell+1)} = h_{t,i}^{(\ell)} + \tilde{h}_{t,i}^{(\ell+1)}.$$

Message passing lets the model capture how load, latency, and failures propagate across connected services. Attention is useful because service dependencies are heterogeneous: scaling one service may strongly affect some downstream services while barely affecting others.

The final node embeddings are decoded by three heads. The dynamics head predicts per-service residual state change: $\Delta\hat{x}_{t,i} = \phi_{\mathrm{dyn}}(h_{t,i}^{(L)})$. The reward and constraint heads operate on a graph-level representation $g_t = \mathrm{READOUT}\left(\{h_{t,i}^{(L)}\}_{i=1}^{N}\right)$, where READOUT is mean pooling, yielding $\hat{r}_t = \phi_{\mathrm{rew}}(g_t), \hat{c}_t = \sigma(\phi_{\mathrm{con}}(g_t))$. Thus, the same world model predicts how the system evolves, how costly the transition is, and whether the future is unsafe.

The world model is trained offline from logged Kubernetes transitions $(\mathcal{G}, X_t, A_t, X_{t+1}, r_t, c_t)$ by minimising a multi-task objective:

$$\mathcal{L}(\theta) = \underbrace{\left\| f_\theta^{\mathrm{dyn}}(\mathcal{G}, X_t, A_t) - (X_{t+1} - X_t) \right\|_2^2}_{\text{state dynamics}} + \alpha \underbrace{(\hat{r}_t - r_t)^2}_{\text{reward}} + \beta \underbrace{\mathrm{BCE}(\hat{c}_t, c_t)}_{\text{constraint}}.$$

Instead of learning a policy directly through trial and error, NetDream first learns how the cluster responds to actions; the learned model then becomes a simulator for planning. This avoids unsafe online exploration in the real cluster and makes the controller inspectable: predictions, rewards, and violation risks are explicit quantities that can be evaluated before action execution.

3.3 Imagination-Based Planning

At deployment time, NetDream uses the learned world model for model-predictive control. Given the current state X_t, the planner samples K candidate action sequences of horizon H:

$$\mathbf{A}_{t:t+H-1}^{(k)} = (A_t^{(k)}, \ldots, A_{t+H-1}^{(k)}), \qquad k = 1, \ldots, K.$$

Each sequence is rolled out inside the world model:

$$\hat{X}_{t+h+1}^{(k)} = \hat{X}_{t+h}^{(k)} + f_\theta^{\mathrm{dyn}}(\mathcal{G}, \hat{X}_{t+h}^{(k)}, A_{t+h}^{(k)}),$$

while also predicting rewards and violation probabilities $\hat{r}_{t+h}^{(k)}$ and $\hat{c}_{t+h}^{(k)}$. The predicted return of a candidate plan is $J^{(k)} = \sum_{h=0}^{H-1} \gamma^h \hat{r}_{t+h}^{(k)}$.

Before selecting the best plan, NetDream applies an imagination-based safety filter. A candidate sequence is accepted only if its predicted violation probability remains below threshold τ at every imagined step: $\max_{0 \le h < H} \hat{c}_{t+h}^{(k)} \le \tau$. Let $\mathcal{K}_{\mathrm{safe}}$ denote the set of candidates that pass this filter. NetDream selects $k^* = \arg\max_{k \in \mathcal{K}_{\mathrm{safe}}} J^{(k)}$ and executes only the first action $A_t^{(k^*)}$ in the real cluster. If no candidate is predicted safe, the controller falls back to a conservative no-op action. After observing the next real cluster state, NetDream replans. Algorithm 1 summarises the loop.

3.4 Online Deployment

Once trained, the graph world model and the imagination planner are packaged into a small controller service that connects to the same interfaces used by standard Kubernetes autoscalers. The same controller binary runs against both local development clusters and managed cloud clusters; only the kubeconfig target changes. At each control step, the controller reconstructs the current state of the service graph by querying Prometheus for per-service CPU usage, memory usage, request rate, latency, and error rate, and by reading the current replica count from the Kubernetes API. These measurements are assembled into the state matrix $X_t \in \mathbb{R}^{N \times d}$, where each row describes one service in the microservice graph.

The state matrix X_t and the service-call graph $\mathcal{G}$ are then passed to the NetDream planner in Algorithm 1. Rather than mapping the current state directly to a scaling decision, the planner samples candidate action sequences, rolls them forward inside the learned world model, estimates their future cost and violation risk, and filters out plans predicted to be unsafe. The selected plan produces a per-service action vector $A_t^{(k^*)} \in \{-1, 0, +1\}^N$, and only its first action is executed on the real cluster by patching the corresponding Kubernetes's scale sub-resource. After the action is applied,

Algorithm 1 NetDream: Imagination-Based Safe Autoscaling

Input: graph $\mathcal{G}$, state X_t, world model f_θ, horizon H, candidates K, threshold τ, discount γ
Sample candidate action sequences $\{\mathbf{A}^{(k)}_{t:t+H-1}\}_{k=1}^{K}$
for $k = 1$ to K **do**
  Initialize $\hat{X}_t^{(k)} \leftarrow X_t$, $J^{(k)} \leftarrow 0$, $s^{(k)} \leftarrow \text{true}$
  for $h = 0$ to $H - 1$ **do**
    Predict $(\Delta\hat{X}, \hat{r}, \hat{c}) \leftarrow f_\theta(\mathcal{G}, \hat{X}^{(k)}_{t+h}, A^{(k)}_{t+h})$
    Update imagined state $\hat{X}^{(k)}_{t+h+1} \leftarrow \hat{X}^{(k)}_{t+h} + \Delta\hat{X}$
    Update score $J^{(k)} \leftarrow J^{(k)} + \gamma^h \hat{r}$
    if $\hat{c} > \tau$ **then**
      $s^{(k)} \leftarrow \text{false}$
    end if
  end for
end for
$\mathcal{K}_{\text{safe}} \leftarrow \{k : s^{(k)} = \text{true}\}$
if $\mathcal{K}_{\text{safe}} \neq \emptyset$ **then**
  $k^* \leftarrow \arg\max_{k \in \mathcal{K}_{\text{safe}}} J^{(k)}$
  Execute $A_t^{(k^*)}$
else
  Execute no-op action
end if
Observe X_{t+1} and re-plan

the cluster evolves, new metrics arrive, and the transition $(X_t, A_t, X_{t+1}, r_t, c_t)$ is appended to an offline log for later retraining or fine-tuning. This is where the world model becomes a control mechanism rather than an auxiliary prediction module: at every step, NetDream simulates possible futures, compares their cost, estimates their safety, rejects unsafe trajectories, and executes the first action of the best safe plan. The result is a deployed autoscaler that turns Kubernetes control from reactive thresholding into world-model-based planning. Detailed cluster configuration, controller pseudocode, and operational metrics are provided in Appendix A.

4 Experimental Results

4.1 Experimental Setup

We evaluate NetDream on Google's Online Boutique application (Google Cloud Platform, 2023), a standard microservice benchmark that matches the structure of modern distributed ML services: a user request enters through a frontend, propagates across multiple dependent services, and creates latency and load interactions that cannot be understood service by service in isolation. The application contains 11 microservices connected by 14 directed service-call dependencies. Each service is described by six operational features, CPU usage, memory usage, queries per second, latency, error rate, and replica count, yielding a 66-dimensional cluster state at each control step. The autoscaler acts at the service level: for every service, it chooses one action from $\{-1, 0, +1\}$, corresponding to scaling down by one replica, holding the allocation fixed, or scaling up by one replica. This makes the benchmark a structured control problem: the controller must decide not only how much capacity to add, but where capacity should be placed in the service graph.

NetDream is trained from 48,000 logged transitions collected on the Online Boutique service graph. Each transition records the current service states, the scaling action, the next service states, the realized reward, and whether a service-level violation occurred. We use the resulting dataset to train the graph world model offline, and then test whether it can support safe online planning. World-model prediction and architecture ablations are evaluated in a calibrated Online Boutique simulator, while the main closed-loop autoscaling results are evaluated on a production-grade AWS Kubernetes deployment. In the real-cluster experiments, metrics are scraped every 5 seconds through Prometheus, traffic is generated with Locust and replayed from production microservice traces (Alibaba Group, 2022), and scaling actions are applied through a custom Kubernetes controller wrapping the NetDream

planner. We measure safety by the number of service-level violations per episode, and cost by total replica-steps, $\text{Cost} = \sum_{t=1}^{T} \sum_{s} n_s(t)$, where $n_s(t)$ is the number of replicas for service s at time t. Table 1 summarizes the real deployment used for evaluation.

We compare against six baselines: *Random*, which samples scaling actions uniformly; *HPA*, the standard reactive Kubernetes autoscaler; *HPA-min3* and *HPA-min5*, which add static replica floors and represent over-provisioning as a safety strategy; *PPO*, a model-free reinforcement-learning autoscaler trained from half a million simulator steps; and *NetDream-Unsafe*, which keeps the world-model planner but removes the safety filter. The two most methodologically related concurrent works, GraphPilot (Ho et al., 2025) (graph-structured actor–critic) and KARMA (Soulé et al., 2025) (digital-twin RL), did not have public implementations at evaluation time; we compare against their algorithmic paradigms (graph-predictive RL via PPO-500K and learned-dynamics planning via NetDream itself) rather than their specific systems.

Table 1: Production-grade AWS Kubernetes cluster test bed.

Parameter	**Value**
Cluster hardware	
Node type	AWS t3.medium × 3 (2 vCPU, 4 GB RAM / node)
Total capacity	6 vCPU, 12 GB RAM
Experiment scope	
Total EKS episodes	85
Steps per episode	60 (5-minute window at 5 s/step)
Total decision steps	5,100
Prometheus scrape points	56,100 (85 ep × 60 steps × 11 services)
Total experiment wall time	3.4 hours
Pod scheduling activity	
Replica range (per service)	1–10
Total cluster pods (per step)	11–110 (mean 48.5)
Pod creation events	6,529
Pod deletion events	6,384
Observed workload metrics	
Frontend QPS	0–96 req/s (mean 35)
Frontend latency	180–2,000 ms (mean 552 ms)
Frontend error rate	0.00%–4.02%

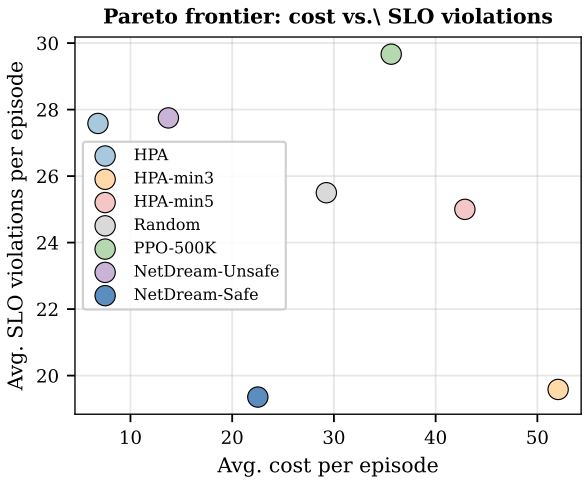


Figure 2: Cost–safety Pareto frontier on AWS Kubernetes. HPA is the lowest-cost operating point, while NetDream-Safe is the lowest-violation operating point. NetDream-Safe provides the most cost-efficient path toward safety.

Table 2: Multi-step prediction accuracy on the 11-service Kubernetes microservice graph. Lower MSE is better.

Setting	1-step	5-step	10-step
Kubernetes mesh	2.50×10^{-4}	2.09×10^{-3}	4.70×10^{-3}

4.2 Main Results:

NetDream-Safe gives the best safety among all evaluated methods while using far less cost than static over-provisioning (see Figure 2). Across the four workload regimes, NetDream-Safe achieves the lowest average number of service-level violations, while remaining much cheaper than the static replica-floor baselines. The resulting Pareto frontier has two meaningful operating points. HPA is the lowest-cost controller, but it pays for this low cost with delayed reactions and more violations. NetDream-Safe is the lowest-violation controller, but unlike static over-provisioning, it does not buy safety by keeping every service over-replicated. This is exactly the behavior desired from a safe autoscaler: spend additional replica budget only when the learned model predicts that doing so can prevent future service-level violations.

Static over-provisioning also improves safety, but it does so by paying a large and mostly uniform capacity tax. HPA-min3 reaches a similar average violation level to NetDream-Safe, but its average cost is more than twice as high. HPA-min5 is even more expensive and still does not produce a better safety outcome. In marginal terms, NetDream-Safe reduces violations at a cost of only 2.10 per violation avoided, whereas the best static over-provisioning strategy requires 5.65 cost per violation avoided. Thus, NetDream-Safe is $2.7\times$ more cost-efficient at buying safety. This gap is important because autoscaling is not only about reducing violations; a trivial way to reduce violations is to keep many replicas alive everywhere. The harder problem is to reduce violations without turning the cluster into a permanently over-provisioned deployment.

The result highlights the difference between *more capacity* and *better capacity placement*. HPA-min3 and HPA-min5 improve safety by raising a global replica floor, which adds capacity even to services that are not on the critical path for the current workload. NetDream takes a different route. It uses the graph world model to predict how load, latency, and violation risk will propagate through the service graph under candidate scaling actions. The safety filter then removes plans whose imagined futures are likely to violate service-level constraints, leaving the planner to choose the lowest-cost safe action. As a result, NetDream does not simply add replicas; it places replicas where they are predicted to matter.

This distinction explains why NetDream moves the cost–safety frontier rather than merely sliding along the same over-provisioning curve. Static replica floors trade cost for safety in a blunt way: higher floors usually mean fewer violations but substantially higher cost. NetDream changes the shape of this trade-off. By planning over a learned service graph, it identifies when extra capacity is useful, where it should be placed, and when it can be avoided. The main empirical takeaway is therefore not that NetDream uses more resources than HPA, but that the resources it uses are much more effective: each additional replica-step buys substantially more safety than static over-provisioning.

4.3 Real World Implementation: Can NetDream Imagine Kubernetes Futures?

The main result depends on a necessary condition: the world model must be accurate enough to compare short-horizon futures. We therefore evaluate multi-step prediction error on the 11-service Kubernetes graph. The planner does not require perfect long-term simulation; it needs a local model that can reliably rank candidate scaling trajectories over the next few control steps. Table 2 shows that NetDream maintains low error over the planning horizon: 1-step prediction error is 2.50×10^{-4} and remains below 5×10^{-3} at 10 steps. This supports the use of short-horizon model-predictive control. NetDream is not trying to build a perfect simulator of the entire cluster for all future time; it only needs to imagine enough of the near future to avoid bad scaling decisions.

The hardest feature is latency. Its prediction error is 1.02×10^{-2}, much larger than the error for replica count (5.5×10^{-5}). This is expected: replica count is nearly deterministic given the action, whereas latency depends on queueing, contention, workload shifts, and pod readiness. This observation also

foreshadows the workload-level results: NetDream is strongest when future pressure is predictable, and weaker when violations are driven by abrupt latency spikes. The constraint head gives the planner an explicit safety signal. On the Kubernetes validation set, it reaches 99.84% F1, allowing the planner to reject futures predicted to produce service-level violations. This is the key difference between using a world model for passive forecasting and using it for safe control.

4.4 Deep Analysis: When Does Imagination Help Most?

The aggregate result hides an important pattern: NetDream helps most when the workload changes enough that reactive control lags, but not so abruptly that pod startup time dominates the episode. The clearest win is variable traffic. NetDream-Safe reduces violations from 13.0 to 2.7 compared with HPA, a nearly $5\times$ reduction (see Figure. 3), while using about half the cost of HPA-min5, the strongest static floor baseline with comparable safety on this workload. This is the ideal regime for world-model planning: the system is changing, so reactive thresholds lag, but the future is still predictable enough for the model to allocate replicas before violations occur. Steady traffic is less dramatic because HPA is already strong when the system is stable. In this regime, there is less future uncertainty to exploit, so the advantage of imagination is smaller. Bursty and flash-crowd traffic are difficult for a different reason: latency can spike faster than new replicas become ready. This exposes a physical limit of pod-level autoscaling. No planner can fully prevent violations when capacity is needed before Kubernetes can start it.

This analysis sharpens the interpretation of NetDream. Its advantage is that it spends replica budget where prediction is actionable. When future pressure is visible through the service graph, NetDream can act before the threshold is crossed. When the bottleneck is pod startup latency under abrupt flash traffic, better planning must be paired with online adaptation or node-level capacity planning. Overall, the experiments support the paper's central claim: the path to safe and efficient Kubernetes autoscaling is not static over-provisioning or model-free reaction, but planning over a learned graph model on this benchmark. By learning a graph world model and planning through it, NetDream can predict which futures are unsafe, reject them before execution, and move the system to a better cost–safety frontier.

5 Conclusion

We present *NetDream*, a graph world-modeling approach that turns distributed ML autoscaling from reactive threshold control into predictive safe planning. Rather than scaling after pressure appears or buying safety through static over-provisioning, NetDream learns how load, latency, replicas, and failures evolve across a microservice graph, imagines the multi-step consequences of candidate scaling actions, and rejects futures predicted to violate service-level constraints before acting. On a production-grade AWS distributed ML deployment of Online Boutique, NetDream-Safe achieves the most cost-efficient safe operating point, reducing violations far more efficiently than static replica floors and avoiding the over-provisioning that model-free RL adopts to remain safe. The gain comes from the method's two ingredients: topology-aware world modeling, which makes short-horizon imagination accurate, and explicit safety filtering, which turns prediction into reliable action selection. The takeaway is that safer, cheaper cloud control comes from planning in a learned graph model, not from faster reaction or larger replica floors.

Ethics Statement

This work studies autoscaling for microservice deployments. It uses only synthetic and public benchmark workloads. No human subjects, personal data, or private user traces are involved. The artifacts described in Appendix B are controller code, training scripts, model checkpoints, and infrastructure manifests, which carry no substantial dual-use risk.

The one ethical consideration we see is operational. A learned controller can be trusted past the range in which it was validated. Appendix E addresses this directly: the safety filter is empirical rather than formally verified, its in-distribution recall does not carry over under distribution shift, and conservative fallbacks, monitoring, and operator override remain necessary in a real deployment. We are aware of no other concerns under the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

What is needed to reproduce these results is in the appendix. Appendix B.1 gives the model, training, and planning configuration, including node features, optimizer settings, loss weights, planning horizon, candidate-plan count, and the safety threshold. Appendix B.3 describes the AWS EKS deployment: cluster and node configuration, the Prometheus scraping interval, the runtime controller loop, and the episode structure. Appendix A.3 defines the four workload regimes. Appendix B.2 gives pseudocode for both the training procedure and the planner.

Appendix A.6 reports the statistical basis for the main comparisons. EKS results are averaged over three random seeds and simulator analyses over five, with seed values stated in the appendix. That section also reports per-seed numbers and bootstrap 95% confidence intervals, and it states plainly which intervals cross zero, so a reader can separate the cost claim from the weaker violation claim.

Code, trained model checkpoints, and the infrastructure manifests for the AWS EKS testbed are released anonymously at `https://anonymous.4open.science/r/netdream-FCF9`, together with the training and evaluation scripts and the workload-generation setup.

AI USE STATEMENT

We used generative AI tools solely for manuscript preparation and presentation. Specifically, we used generative AI for drafting and revising prose, language polishing, preparing and adapting the LaTeX manuscript to the official ICLR 2027 style files, and formatting references.

We did not use generative AI tools to formulate or refine research hypotheses, design experiments or experimental methodology, implement the proposed method, generate or clean experimental data, perform data analysis, interpret experimental results, develop theoretical or conceptual models, formulate mathematical claims, or write proofs. Translation assistance and synthetic data generation were not used in this work.

All AI-assisted text and formatting were reviewed and edited by the authors. All experimental results reported in this paper were produced by author-executed runs on the AWS EKS testbed and the simulator described in Appendix B; no reported experimental result was generated by a generative AI system. The authors verified the references used in the final manuscript and take full responsibility for the content of the paper, including all AI-assisted text and formatting.

REFERENCES

Abeer Abdel Khaleq and Ilkyeun Ra. Intelligent microservices autoscaling module using reinforcement learning. *Cluster Computing*, 26(5):2789–2800, 2023. doi: 10.1007/s10586-023-03999-8.

Siddharth Agarwal, Maria Alejandra Rodriguez, and Rajkumar Buyya. A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions. *IEEE Transactions on Services Computing*, 17(5):1899–1910, 2024. doi: 10.1109/TSC.2024.3387661.

Hussain Ahmad, Christoph Treude, Markus Wagner, and Claudia Szabo. Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pp. 46–57. IEEE, 2024. doi: 10.1109/ICSA59870.2024.00013.

Alibaba Group. Alibaba cluster trace program: Microservice traces. `https://github.com/alibaba/clusterdata`, 2022.

Amazon Web Services. Amazon EKS: Horizontal Pod Autoscaler documentation. `https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html`, 2025. Accessed 2026-04.

Haoyu Bai, Minxian Xu, Kejiang Ye, Rajkumar Buyya, and Chengzhong Xu. DRPC: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. *arXiv preprint arXiv:2407.10169*, 2024.

Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. pHPA: A proactive autoscaling framework for microservice chain. In *Asia-Pacific Workshop on Networking (APNet)*, 2021. doi: 10.1145/3469393.3469401.

Google Cloud. GKE: Horizontal Pod autoscaling. `https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler`, 2025. Accessed 2026-04.

Google Cloud Platform. Online Boutique: Cloud-native microservices demo application. `https://github.com/GoogleCloudPlatform/microservices-demo`, 2023.

Po-Han Ho, Hong-Yen Chen, and Tsung-Nan Lin. GraphPilot: A temporal graph actor-critic autoscaler reducing degradation of resource oscillation in microservice. In *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pp. 1–10, 2025. doi: 10.1145/3773274.3774260.

Pengjuan Liang, Yaling Xun, Jianghui Cai, and Haifeng Yang. Autoscaling of microservice resources based on dense connectivity spatio-temporal GNN and Q-learning. *Future Generation Computer Systems*, 174:107909, 2026. doi: 10.1016/j.future.2025.107909.

Locust Contributors. Locust: A modern, scalable, open-source load testing tool. `https://locust.io`, 2024.

Angelo Marchese and Orazio Tomarchio. SLO and cost-driven container autoscaling on Kubernetes clusters. In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER)*, pp. 72–79. SciTePress, 2025. doi: 10.5220/0013482100003950.

Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

Hoa X. Nguyen, Shaoshu Zhu, and Mingming Liu. Graph-PHPA: Graph-based proactive horizontal pod autoscaling for microservices using LSTM-GNN. In *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, pp. 237–241. IEEE, 2022. doi: 10.1109/CLOUDNET55617.2022.9978781.

Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices. *IEEE/ACM Transactions on Networking*, 32(4):3331–3345, 2024. doi: 10.1109/TNET.2024.3378932.

Olesia Pozdniakova, Dalius Mazeika, and Aurimas Cholomskis. SLA-adaptive threshold adjustment for a Kubernetes horizontal pod autoscaler. *Electronics*, 13(7):1242, 2024. doi: 10.3390/electronics13071242.

Wenzhuo Qian, Hailiang Zhao, Tianlv Chen, Jiayi Chen, Ziqi Wang, Kingsum Chow, and Shuiguang Deng. Learning unified system representations for microservice tail latency prediction. *arXiv preprint arXiv:2508.01635*, 2025.

Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 805–825. USENIX Association, 2020.

Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *USENIX Annual Technical Conference (ATC)*, 2023.

Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at Google. In *European Conference on Computer Systems (EuroSys)*, 2020. doi: 10.1145/3342195.3387524.

José Santos, Efstratios Reppas, Tim Wauters, Bruno Volckaert, and Filip De Turck. Gwydion: Efficient auto-scaling for complex containerized applications in Kubernetes through reinforcement learning. *Journal of Network and Computer Applications*, 234:104067, 2025. doi: 10.1016/j.jnca.2024.104067.

Julien Soulé, Jean-Paul Jamont, Michel Occello, Louis-Marie Traonouez, and Paul Théron. Streamlining resilient Kubernetes autoscaling with multi-agent systems via an automated online design framework. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 43–53. IEEE, 2025. doi: 10.1109/CLOUD67622.2025.00015.

Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 149–165. USENIX Association, 2024a.

Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Meng Yan, Xiaohong Zhang, and Alex X. Liu. DeepScaling: Autoscaling microservices with stable CPU utilization for large scale production cloud systems. *IEEE/ACM Transactions on Networking*, 32(5):3961–3976, 2024b. doi: 10.1109/TNET.2024.3400953.

Guilin Zhang, Wulan Guo, Ziqi Tan, Qiang Guan, and Hailong Jiang. KIS-S: A GPU-aware Kubernetes inference simulator with RL-based auto-scaling. In *IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2025. doi: 10.1109/IPCCC66453.2025.11304654.

A KUBERNETES TESTBED AND FULL EMPIRICAL RESULTS

The main paper evaluates NetDream on Kubernetes because Kubernetes autoscaling is exactly the setting where local actions have graph-level consequences. Scaling the frontend may relieve one bottleneck while shifting pressure to checkout, cart, or recommendation services. This section therefore moves from implementation to evidence: it first describes the simulated and real Kubernetes environments, then specifies the service graph and workload regimes, and finally reports the full per-workload and per-seed results behind the aggregate Pareto frontier.

A.1 KUBERNETES SIMULATOR AND REAL CLUSTER

We use the simulator for controlled training and diagnostics, and the EKS cluster for the final closed-loop test. The simulator is designed to capture the phenomena that make autoscaling difficult: traffic fans out through the graph, overloaded services create cascade penalties, new pods are not ready immediately, and CPU responds to load per replica. We simulate Google Online Boutique with 11 services connected by 14 directed API-call edges. The GNN uses the bidirectional version of this graph. The simulator models four effects: traffic propagation from the frontend to downstream services, cascade penalties when callers approach the SLO boundary, a 3-step pod startup delay for new replicas, and a CPU model of

$$\text{CPU\%} = 5 + (\text{QPS}/\text{replicas}) \times 0.7.$$

The SLO violation indicator is triggered when p95 latency exceeds 500 ms or error rate exceeds 5%.

For real-cluster validation, Online Boutique is deployed on both a local MicroK8s cluster and a production-grade AWS EKS cluster. The local cluster supports development and sanity checks; the EKS cluster is used for final closed-loop evaluation. Metrics are collected through Prometheus, workloads are generated by Locust, and Cluster Autoscaler is disabled so that pod-level replica decisions come only from the evaluated autoscaler.

These statistics confirm that the closed-loop results exercise the real Kubernetes control plane: thousands of pod creation and deletion events occur under nontrivial traffic and latency variation. We next describe the service graph over which these decisions are made.

A.2 ONLINE BOUTIQUE SERVICE GRAPH

The next piece of the story is the graph itself. NetDream does not see a monolithic cluster state; it sees an interacting service graph. Each node carries local resource and performance metrics, while edges define how pressure can propagate. Tables 3 and 4 define the topology used by the world model.

Table 3: Online Boutique services and node feature vector. All features are normalized to $[0, 1]$ before input to the GNN.

ID	Service	Callers	Role
0	frontend	external	Entry point; fans out to 7 services
1	cartservice	frontend, checkoutservice	Manages shopping cart (Redis backend)
2	productcatalogservice	frontend, recommendationservice	Product catalog and search
3	currencyservice	frontend, checkoutservice	Currency conversion
4	shippingservice	frontend, checkoutservice	Shipping cost estimation
5	adservice	frontend	Ad recommendation
6	checkoutservice	frontend	Orchestrates order checkout
7	recommendationservice	frontend	Product recommendations
8	emailservice	checkoutservice	Order confirmation emails
9	paymentservice	checkoutservice	Payment processing
10	redis-cart	cartservice	In-memory cart storage

Node feature vector (dim = 6): [cpu_util, memory_util, qps, p95_latency, error_rate, replicas]
Action space (dim = 3 per node): {scale-down -1*, hold* 0*, scale-up* $+1$*}*

Table 4: Directed call graph: 14 edges defining service dependencies. GNN uses bidirectional version (28 edges).

Source	Target	Semantic
frontend (0)	cartservice (1)	Cart retrieval
frontend (0)	productcatalogservice (2)	Product listing
frontend (0)	currencyservice (3)	Price display
frontend (0)	shippingservice (4)	Shipping estimate
frontend (0)	adservice (5)	Ad serving
frontend (0)	checkoutservice (6)	Checkout initiation
frontend (0)	recommendationservice (7)	Recommendation widget
checkoutservice (6)	cartservice (1)	Cart read at checkout
checkoutservice (6)	shippingservice (4)	Final shipping cost
checkoutservice (6)	currencyservice (3)	Currency at checkout
checkoutservice (6)	emailservice (8)	Order confirmation
checkoutservice (6)	paymentservice (9)	Payment processing
recommendationservice (7)	productcatalogservice (2)	Product lookup
cartservice (1)	redis-cart (10)	Cart persistence

This topology defines the structure over which NetDream performs message passing. The workloads below then determine how pressure moves through that structure.

A.3 Workload Patterns and Calibration

Once the graph is fixed, the empirical question becomes: under what kinds of demand does imagination help? The four workload regimes are chosen to tell a progressively harder autoscaling story. Constant traffic tests replica efficiency under stable load; variable traffic tests anticipation under smooth demand changes; bursty traffic tests unpredictable spikes; flash-crowd traffic tests the physical limit of pod-level autoscaling when capacity is needed before new pods become ready.

Table 5: Workload pattern definitions. t is normalized to $[0, 1]$. $\xi \sim \mathcal{N}(0, 5)$ is step-level noise.

Workload	Definition $q(t)$
Constant	$q_{\text{mid}} + \xi$. Steady-state load; tests replica efficiency at stable utilization.
Variable	$q_{\text{mid}} + q_{\text{amp}} \sin(2\pi \cdot 3t) + \xi$. Three-cycle sinusoidal variation; tests predictive scaling during gradual ramps.
Bursty	$q_{\text{lo}} + (q_{\text{hi}} - q_{\text{lo}}) \cdot \mathbf{1}[\text{Bernoulli}(0.25)] + \xi$. Random 25%-duty-cycle bursts of 3× baseline load; sampled from Alibaba production traces (Alibaba Group, 2022). Tests response to sudden, unpredictable spikes.
Flash crowd	q_{lo} for $t < 0.5$, then q_{hi} for $t \geq 0.5$. A single step-function increase at mid-episode; tests cold-start recovery under a large instantaneous load increase.

The workload definitions make the evaluation progressive: the first two regimes test whether planning can be useful, while the last two expose the limits of pod-level autoscaling under rapid changes.

The simulator calibrates CPU, latency, and QPS routing. CPU utilization uses an exponential moving average $u_s(t{+}1) = \alpha u_s(t) + (1{-}\alpha)[\text{QPS}_s(t)/n_s(t)]$ with $\alpha = 0.8$. Latency follows a quadratic blow-up above 70% utilization. QPS routing follows the call graph $\text{QPS}_v(t) = \sum_{u \in \text{parents}(v)} w_{uv} \text{QPS}_u(t)$. HPA calibration checks that a 70% CPU target produces scaling decisions within ± 2 replicas of production HPA on 98% of steps.

A.4 Full Kubernetes Results

With the environment defined, we now expose the full result table behind the main paper. Table 6 gives the complete per-workload breakdown. The pattern is the same as in the main text: NetDream-Safe

does not simply buy safety by raising the replica floor. It achieves comparable safety to the strongest static over-provisioning baseline at much lower cost, especially on variable traffic where the future is predictable enough for planning to matter.

Table 6: Full EKS K8s planning results across all 7 agents and 4 workloads (mean ± std, 3 seeds; NetDream constant uses seeds 42, 123, 456 only). **Bold** = best per column. NetDream-Safe is the most cost-efficient safe agent in expectation.

Workload	Agent	Cost (mean)	Violations (mean)	Cost (±std)	Violations (±std)
Constant	**HPA**	6.60	1.67	±0.01	±1.25
	Random	25.13	0.0	±3.17	±0.0
	HPA-min3	51.03	8.33	±18.4	±2.87
	HPA-min5	54.08	0.0	±16.9	±0.0
	PPO-500K	34.50	1.33	±4.95	±1.89
	NetDream-Unsafe	13.68	0.0	±3.61	±0.0
	NetDream-Safe	**28.12**	**1.00**	±11.36	±1.73
Variable	**HPA**	6.82	13.0	±0.32	±11.4
	Random	30.51	1.0	±2.97	±1.41
	HPA-min3	52.10	3.67	±19.7	±5.19
	HPA-min5	51.60	2.67	±13.4	±3.77
	PPO-500K	35.71	16.7	±4.66	±11.7
	NetDream-Unsafe	11.59	13.0	±1.70	±13.6
	NetDream-Safe	**25.72**	**2.67**	±11.5	±2.49
Bursty	**HPA**	6.72	41.3	±0.17	±4.78
	Random	30.69	43.0	±2.75	±11.0
	HPA-min3	52.19	18.3	±19.5	±25.2
	HPA-min5	31.63	43.3	±5.63	±13.9
	PPO-500K	36.00	43.7	±4.24	±14.0
	NetDream-Unsafe	13.78	44.7	±4.22	±11.3
	NetDream-Safe	**21.19**	**21.7**	±7.78	±15.6
Flash	**HPA**	7.08	54.3	±0.68	±2.62
	Random	30.69	58.0	±2.75	±0.82
	HPA-min3	52.84	48.0	±18.6	±5.72
	HPA-min5	34.19	54.0	±3.75	±1.41
	PPO-500K	36.32	57.0	±3.79	±1.41
	NetDream-Unsafe	15.86	53.3	±5.50	±1.25
	NetDream-Safe	**20.86**	**52.3**	±6.89	±1.25

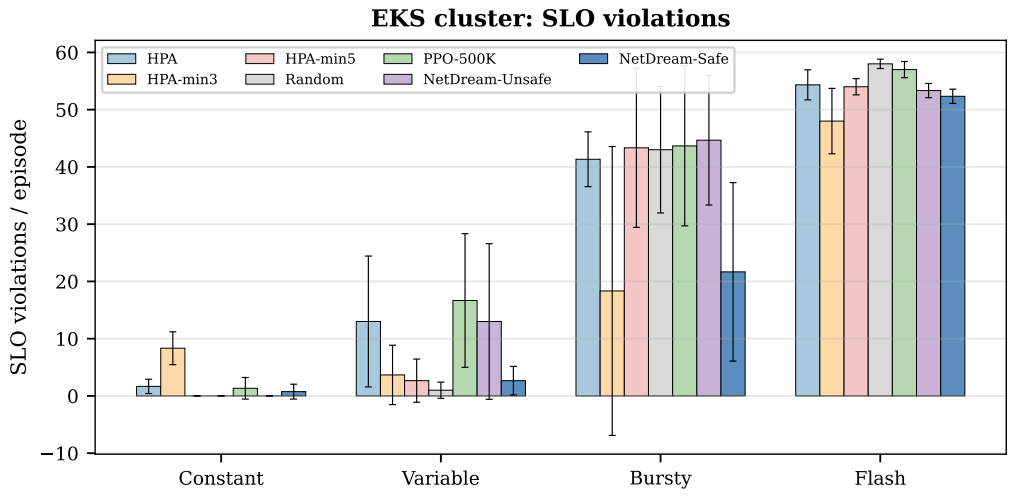


Figure 3: Service-level violations per episode on AWS Kubernetes by workload. NetDream-Safe achieves about $5\times$ fewer violations than HPA on variable traffic.

The table shows the high-level pattern, but it also raises a natural question: are these results driven by a few lucky seeds, or do the per-seed traces tell a consistent story? We answer that next.

A.5 Per-Seed Raw Results

The aggregate table hides some useful behavior, so we include the raw seed-level results. Tables 7–10 show that static over-provisioning is not merely expensive on average; it is also bimodal. In some seeds the controller reaches a 110-replica plateau, while in others it spends most of the episode near minimum replicas. NetDream-Safe's variance on bursty traffic reflects a different mechanism: whether the latency spike falls within the 5-step planning horizon.

Table 7: Per-seed results: Constant workload (60-step episodes). Seed 99 for NetDream-Safe is excluded (30-step episode).

Agent	Seed 42 Cost	Seed 42 Viols	Seed 123 Cost	Seed 123 Viols
HPA	6.61	2	6.60	0
Random	24.10	0	29.43	0
HPA-min3	62.04	8	66.00	5
HPA-min5	66.00	0	66.00	0
PPO-500K	36.89	4	39.00	0
NetDream-Unsafe	18.42	0	12.97	0
NetDream-Safe	**36.20**	**0**	**33.04**	**3**

Agent	Seed 456 Cost	Seed 456 Viols
HPA	6.60	3
Random	21.87	0
HPA-min3	25.06	12
HPA-min5	30.23	0
PPO-500K	27.61	0
NetDream-Unsafe	9.66	0
NetDream-Safe	**15.13**	**0**

Table 8: Per-seed results: Variable workload (60-step episodes).

Agent	S42 Cost	S42 V	S123 Cost	S123 V	S456 Cost	S456 V
HPA	6.60	29	6.60	3	7.27	7
Random	34.28	3	30.21	0	27.03	0
HPA-min3	66.00	0	66.00	0	24.30	11
HPA-min5	66.00	0	55.00	0	33.79	8
PPO-500K	39.00	1	39.00	29	29.12	20
NetDream-Unsafe	11.66	32	13.64	6	9.47	1
NetDream-Safe	**32.83**	**0**	**34.83**	**2**	**9.49**	**6**

Table 9: Per-seed results: Bursty workload (60-step episodes). HPA-min3 seed 456 exits early (bimodal behavior); seed 42/123 saturate at maximum replicas while seed 456 under-provisions.

Agent	S42 Cost	S42 V	S123 Cost	S123 V	S456 Cost	S456 V
HPA	6.60	48	6.60	37	6.96	39
Random	34.28	29	30.21	56	27.59	44
HPA-min3	66.00	0	66.00	1	24.57	54
HPA-min5	27.66	44	39.59	60	27.64	26
PPO-500K	39.00	52	39.00	24	30.00	55
NetDream-Unsafe	17.54	41	15.92	33	7.88	60
NetDream-Safe	**26.84**	**0**	**26.54**	**29**	**10.19**	**36**

Table 10: Per-seed results: Flash-crowd workload (60-step episodes). Flash traffic is hard for all agents; even the safest static baseline (HPA-min3) averages 48 violations, demonstrating the fundamental cold-start pod-launch bottleneck that limits any autoscaler.

Agent	S42 Cost	S42 V	S123 Cost	S123 V	S456 Cost	S456 V
HPA	6.60	52	6.60	53	8.04	58
Random	34.28	59	30.21	57	27.59	58
HPA-min3	66.00	41	66.00	48	26.53	55
HPA-min5	30.97	56	39.45	53	32.16	53
PPO-500K	39.00	55	39.00	58	30.97	58
NetDream-Unsafe	19.31	52	20.17	53	8.09	55
NetDream-Safe	**27.75**	**52**	**23.38**	**51**	**11.45**	**54**

A.6 Statistical Interpretation

The seed-level view also clarifies how the main claim should be stated. NetDream-Safe is not presented as statistically dominating every static baseline on violations alone. The stronger and better-supported claim is cost-efficient safety: it reaches comparable safety at much lower replica cost. The violation advantage over HPA-min3 is small in aggregate and the confidence interval crosses zero; the cost advantage, however, is large and stable.

Table 11: Per-workload violations: NetDream-Safe vs. HPA-min3 (3 seeds each). Bootstrap 95% CI (10,000 resamples). The aggregate dominance in expectation masks workload-specific reversals.

Workload	NetDream-Safe mean (std)	HPA-min3 mean (std)	Difference (NetDream – HPA-min3)	95% CI
Constant	1.0 (1.7)	8.3 (2.9)	-7.3	$[-11.8, -2.8]$
Variable	2.7 (3.1)	3.7 (5.2)	-1.0	$[-8.5, +6.5]$
Bursty	21.7 (15.6)	18.3 (25.2)	$+3.4$	$[-27.3, +34.1]$
Flash	52.3 (1.5)	48.0 (5.7)	$+4.3$	$[-0.5, +9.1]$
Average	**19.4**	**19.6**	$\mathbf{-0.2}$	$[-13.7, +13.3]$

This is why the paper emphasizes cost-efficient safety rather than absolute violation dominance. NetDream-Safe is valuable because it moves toward the low-violation operating point without paying the full cost of static over-provisioning.

The additional development run with seed 99 for NetDream-Safe on the constant workload is excluded from all reported numbers because it contains only 30 decision steps. All reported EKS results use seeds $\{42, 123, 456\}$ and full 60-step episodes.

B Reproducibility and Deployment Details

The main paper makes a simple claim: safe Kubernetes autoscaling is not only a control problem, but a world-modeling problem. A controller that reacts only after pressure appears is always one step behind; a controller that over-provisions everywhere buys safety too expensively. NetDream takes the third path: it learns a compact graph world model, uses it to imagine the near future, and then acts only on futures that look safe.

This first appendix section opens the box. It shows that the main results do not depend on a large hidden system or fragile engineering tricks. NetDream is a small residual GAT, trained offline from logged transitions, and queried online by a short-horizon planner. We first give the exact hyperparameters, then show the training and compute profile, and finally describe the runtime controller used in the AWS EKS deployment.

B.1 Model, Training, and Planning Configuration

The key design choice is to keep the world model small enough to be operational. The model must be expressive enough to predict how load and latency propagate through a microservice graph, but cheap enough to be queried many times inside a planning loop. NetDream therefore uses a 3-layer graph attention network with residual dynamics and three heads: one for next-state prediction, one for reward, and one for constraint risk. Table 12 gives the exact configuration used throughout the Kubernetes experiments unless otherwise stated.

Table 12: NetDream hyperparameter configuration.

Parameter	**Value**
GNN Dynamics Model	
GNN type	GAT (4 heads)
Hidden dimension	128
Number of GNN layers	3
Residual connections	Yes
Dropout	0.0
Total parameters	86,536 (K8s) / 86,793 (Grid)
Training	
Optimizer	Adam
Learning rate	3×10^{-4}
Weight decay	1×10^{-5}
Batch size	128
Max epochs	100
Early stopping patience	15
Gradient clip norm	1.0
Loss weights (α, β)	0.1, 0.1
Planning	
Planner type	Batched random shooting
Number of candidates (K)	64
Planning horizon (H)	5
No-op bias	70%
Safety threshold (τ)	0.5
PPO Baseline	
Algorithm	PPO (Stable-Baselines3)
Network	MLP [64, 64]
Training steps	500,000
Learning rate	3×10^{-4}
Data Collection	
K8s transitions	48,000 (200 ep $\times$ 240 steps)
Grid2Op transitions	12,040 (200 ep)
WaterNet-10 transitions	19,200 (200 ep $\times$ 96 steps)
EpiNet-20 transitions	12,000 (200 ep $\times$ 60 steps)
Train/val split	80% / 20%
Random seeds	$\{42, 123, 456, 789, 1024\}$

Table 12 also highlights the two constraints that shape the method: the model must be small enough for online planning, and the planner must be conservative enough to avoid unsafe futures. The next two paragraphs show that these constraints are satisfied in practice.

Training behavior. After fixing the model size, the next question is whether such a small world model can actually learn useful Kubernetes dynamics. The answer is yes: validation loss stabilizes by roughly epoch 30 and continues to improve slightly through epoch 100, without visible overfitting. This supports a key design choice of the paper: NetDream does not require a massive model or unsafe online reinforcement learning to become useful; a compact offline-trained graph dynamics model is sufficient for short-horizon imagination.

Table 13: Training convergence: validation loss by epoch (K8s GAT model). Early stopping triggered at epoch 100 (patience 15).

Epoch	Train Loss	Val Loss	Best Val	Learning Rate
1	0.01052	0.00819	0.00819	3×10^{-4}
10	0.00485	0.00495	0.00495	3×10^{-4}
30	0.00416	0.00424	0.00424	3×10^{-4}
50	0.00381	0.00421	0.00412	3×10^{-4}
80	0.00328	0.00408	0.00405	3×10^{-4}
100	0.00307	0.00402	0.00400	3×10^{-4}

Thus, the world model reaches a useful validation regime quickly. The next question is whether this learned model can be queried fast enough inside a repeated planning loop.

Computation. This small-model design also matters at deployment time. The planner queries the world model repeatedly, so inference latency must be far below the control interval. In practice, the full experimental pipeline is lightweight: the Kubernetes model trains in minutes, and each online planning step takes less than 10 ms, well below the interval used in our deployment.

Table 14: Wall-clock time for each component on a single NVIDIA GPU.

Component	K8s Mesh	Power Grid
Data collection (200 episodes)	2 min	8 min
Model training (100 epochs)	5 min	3 min
Planning evaluation (3 seeds $\times$ 4 workloads)	15 min	5 min
GNN ablation (4 variants)	20 min	–
Total	$\sim$42 min	$\sim$16 min

These runtime numbers justify the use of model-predictive planning rather than a single direct policy lookup: NetDream can afford to evaluate many imagined futures before acting.

B.2 Algorithms

The method has two phases, mirroring the story in the main paper. In the offline phase, NetDream learns the local physics of the service graph: how the current state and a candidate scaling action change future metrics, rewards, and risk. In the online phase, NetDream turns this model into a decision mechanism: it samples possible scaling futures, rolls them forward in imagination, rejects unsafe ones, and executes only the first action of the best remaining plan. Algorithms 2 and 3 give the exact procedures.

Algorithm 2 NetDream Dynamics Model Training

Require: Transition dataset $\mathcal{D} = \{(G, z_t, a_t, z_{t+1}, r_t, c_t)\}$, GNN type $\in \{\texttt{gat}, \texttt{gcn}, \texttt{sage}\}$, hidden dimension $d = 128$, epochs $E = 100$
Ensure: Trained dynamics model f_θ
Initialize f_θ with Xavier weights
Split $\mathcal{D}$ into train/val at 80/20
for epoch $= 1$ to E **do**
  for each mini-batch $\mathcal{B} \subset \mathcal{D}_{\text{train}}$ **do**
    for $(G, z_t, a_t, z_{t+1}, r_t, c_t) \in \mathcal{B}$ **do**
      $h \leftarrow \text{NodeEncoder}([z_t \,\|\, \text{OneHot}(a_t)])$
      for $k = 1$ to K_{layers} **do**
        $h \leftarrow \text{GNNLayer}_k(h, \mathcal{E}) + h$
        $h \leftarrow \text{LayerNorm}(h)$
      end for
      $\hat{\Delta z} \leftarrow \text{DynamicsHead}(h)$
      $\hat{r} \leftarrow \text{RewardHead}(h)$
      $\hat{c} \leftarrow \sigma(\text{ConstraintHead}(h))$
      $\mathcal{L} \leftarrow \|\hat{\Delta z} - (z_{t+1} - z_t)\|^2 + \alpha\|\hat{r} - r_t\|^2 + \beta\,\text{BCE}(\hat{c}, c_t)$
    end for
    Update $\theta \leftarrow \theta - \eta\nabla_\theta\mathcal{L}$ {Adam, $\eta = 3 \times 10^{-4}$, clip norm 1.0}
  end for
  if val loss has not improved for 15 epochs **then**
    break
  end if
end for
return f_θ

Algorithm 3 NetDream-Safe: Imagination-Based MPC with Safety Filter

Require: Trained model f_θ, current state z_t, graph G, candidates $K = 64$, horizon $H = 5$, safety threshold $\tau = 0.5$, no-op bias $\rho = 0.7$
Ensure: Action a_t^* to execute
for $k = 1$ to K **do**
  for $h = 1$ to H **do**
    $a_h^{(k)} \sim \text{Uniform}(\{-1, 0, +1\}^n)$ with probability ρ replaced by no-op $\mathbf{0}$
  end for
end for
for $k = 1$ to K **do**
  $z_0^{(k)} \leftarrow z_t, \quad \text{safe}^{(k)} \leftarrow \text{true}, \quad R^{(k)} \leftarrow 0$
  for $h = 1$ to H **do**
    $(\hat{\Delta z}, \hat{r}, \hat{c}) \leftarrow f_\theta(G, z_{h-1}^{(k)}, a_h^{(k)})$
    $z_h^{(k)} \leftarrow z_{h-1}^{(k)} + \hat{\Delta z}$
    $R^{(k)} \leftarrow R^{(k)} + \hat{r}$
    if $\hat{c} > \tau$ **then**
      $\text{safe}^{(k)} \leftarrow \text{false}$; **break**
    end if
  end for
end for
$\mathcal{S} \leftarrow \{k : \text{safe}^{(k)} = \text{true}\}$
if $\mathcal{S} \neq \emptyset$ **then**
  $k^* \leftarrow \arg\max_{k \in \mathcal{S}} R^{(k)}$
  return $a_1^{(k^*)}$
else
  return $\mathbf{0}$ {safe default: hold current replicas}
end if

The two algorithms describe the computation; the next subsection describes how that computation is embedded into a live Kubernetes controller.

B.3 Deployment Architecture on AWS EKS

The algorithms above are not only simulator procedures. They are wrapped into a controller that runs against a real Kubernetes control plane. The runtime system has four components, each corresponding to one step in the closed-loop control story: observe the cluster, build the graph state, plan in the world model, and apply the selected scaling action.

State estimator. A Python service scrapes CPU utilization, replica counts, P95 latency, QPS, error rates, and memory usage at 5-second intervals. Raw metrics are normalized using training-set statistics and formatted as a PyG `Data` object with the Online Boutique topology.

NetDream inference service. The inference service receives the graph state, runs Algorithm 3 on GPU, and emits per-service scaling decisions $\{s_i : \delta_i\}_{i=0}^{10}$.

Kubernetes controller. A custom controller applies scaling decisions via the `/scale` subresource on each `Deployment`. Scale-down operations are rate-limited to one replica per service per decision interval to prevent oscillation; scale-up operations are applied immediately.

Monitoring and decision loop. Prometheus collects per-pod metrics, and Grafana visualizes SLO compliance, replica counts, and fallback rate. At each 15-second interval, the system collects metrics, builds the graph state, runs NetDream, and applies scaling decisions. Total loop latency is below 200 ms, dominated by Prometheus scrape delay.

EKS configuration. Experiments run on a 3-node EKS cluster in `us-east-1`. Online Boutique pods are scheduled with resource requests but without hard limits, allowing natural CPU throttling. Locust runs in a dedicated namespace to avoid interfering with service metrics.

B.4 Reproducibility and Responsible Use

Taken together, the preceding details are meant to make NetDream reproducible as a system, not only as a model. All code, trained models, collected data, cluster manifests, and evaluation scripts are included in the supplementary material. The Kubernetes simulator is implemented in Python/NumPy with PyTorch and PyTorch Geometric. EKS experiments use seeds $\{42, 123, 456\}$; simulator ablations and prediction tables use seeds $\{42, 123, 456, 789, 1024\}$.

The safety framing also deserves a careful interpretation. NetDream should be used as a complement to existing production safety mechanisms, not as a replacement. The safety filter can reject predicted unsafe futures, but it remains limited by model accuracy. In deployment, it should be paired with standard safeguards such as circuit breakers, rollback policies, and human oversight.

C World Model Diagnostics and Ablation Studies

The previous section showed what happens in closed loop. This section asks why. NetDream has three moving parts: the graph world model, the safety filter, and the short-horizon planner. The diagnostics below isolate each piece. The story is consistent with the main paper: graph structure makes imagination accurate, the constraint head turns prediction into safety, and latency prediction is the dominant remaining bottleneck.

C.1 Ablation Study: Graph Structure and Safety Filtering Both Matter

We next isolate the two design choices that make NetDream work: topology-aware world modeling and constraint-aware planning. A flat neural predictor observes the same service metrics but ignores the service-call graph. If autoscaling were simply a tabular forecasting problem, this should be sufficient. But microservice dynamics are not flat: load, latency, and failures propagate along service dependencies. Figure 4 compares GAT, GCN, GraphSAGE, and an MLP without graph structure.

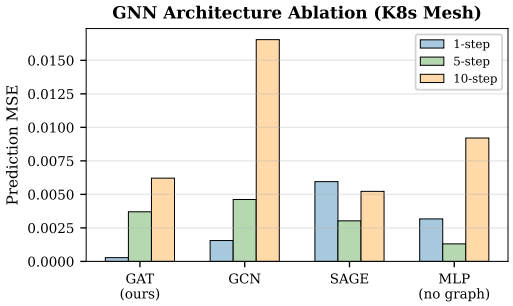


Figure 4: Architecture ablation on Kubernetes dynamics prediction. Removing graph structure increases 1-step prediction error by $11\times$, showing that graph structure is required for reliable multi-step prediction.

Removing graph structure increases one-step prediction error by $11\times$. This is not simply a capacity effect: a GCN with the same parameter count as the GAT still substantially outperforms the MLP. The service-call topology itself provides the inductive bias needed to model how pressure propagates through the system. Among graph models, GAT achieves the best one-step accuracy, suggesting that attention is useful for heterogeneous dependencies: not all service edges carry equal influence.

The second ablation removes the safety filter while keeping the learned world model and planner. NetDream-Unsafe is cheaper than NetDream-Safe, but it does not improve violations over HPA. This shows that prediction alone is not enough. A world model can propose high-reward futures that still carry unacceptable violation risk; the safety filter is what turns imagination into safe imagination.

Together, these ablations clarify the mechanism behind NetDream's gains. The graph world model provides useful futures, and the safety filter decides which futures are acceptable. Removing either component weakens the controller: without the graph, the imagined dynamics are inaccurate; without the filter, the planner does not reliably avoid unsafe actions.

C.2 Graph Architecture and Rollout Stability

We begin with the most basic question: does the graph matter? The flat MLP baseline observes the same metrics but does not know the service-call topology. Its poor one-step accuracy shows that autoscaling is not merely tabular time-series prediction. The system is a graph, and load, latency, and failures propagate along that graph.

Table 15: Full GNN ablation on K8s mesh. GAT achieves the best 1-step accuracy; removing graph structure (MLP) degrades 1-step MSE by $11\times$.

Architecture	1-step MSE	5-step MSE	10-step MSE
GAT (ours)	$\mathbf{2.79 \times 10^{-4}}$	3.70×10^{-3}	6.21×10^{-3}
GCN	1.56×10^{-3}	4.62×10^{-3}	1.65×10^{-2}
SAGE	5.95×10^{-3}	3.03×10^{-3}	5.23×10^{-3}
MLP (no graph)	3.17×10^{-3}	1.31×10^{-3}	9.20×10^{-3}

Table 16: GNN ablation: architecture variants, parameter counts, and multi-step MSE on K8s mesh (5 seeds, 50 training epochs for ablation, 100 epochs for final GAT model in Table 12). The MLP baseline uses 2.4× fewer parameters than graph-structured models.

Architecture	GNN layers	Params	1-step MSE	5-step MSE	10-step MSE
GAT (ours, 100 ep)	3	86,536	$\mathbf{2.50 \times 10^{-4}}$	2.09×10^{-3}	4.70×10^{-3}
GAT (ablation, 50 ep)	3	86,536	2.79×10^{-4}	3.70×10^{-3}	6.21×10^{-3}
GCN	3	85,768	1.56×10^{-3}	4.62×10^{-3}	1.65×10^{-2}
SAGE	3	134,920	5.95×10^{-3}	3.03×10^{-3}	5.23×10^{-3}
MLP (no graph)	0	35,464	3.17×10^{-3}	1.31×10^{-3}	9.20×10^{-3}

GAT outperforms GCN because attention can represent heterogeneous service dependencies; uniform aggregation can dilute high-impact edges. SAGE is competitive at longer horizons, suggesting that hybrid GAT–SAGE architectures may improve rollout stability in future work.

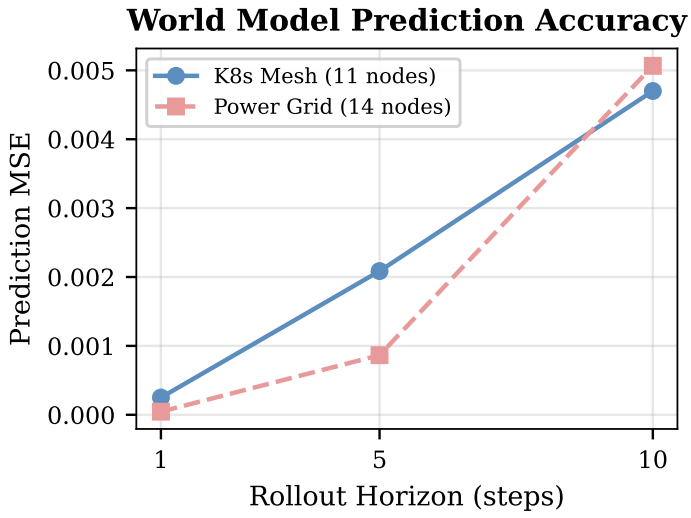


Figure 5: Multi-step prediction MSE vs. horizon for K8s mesh (GAT, GCN, SAGE) and power grid (GAT). Both domains show residual prediction controlling compounding error: 10-step MSE remains below 5.1×10^{-3} for both domains.

The architecture ablation identifies the graph as the right inductive bias. The next diagnostic asks which predicted variables remain hard even with that graph bias.

C.3 Per-Feature Error and Constraint Prediction

After establishing that the graph matters, we next ask which parts of the state are hardest to imagine. The answer is latency. Replica count is almost deterministic once the action is known, but latency depends on queueing, contention, workload spikes, pod readiness, and downstream backpressure. This explains the failure mode observed in bursty and flash-crowd workloads.

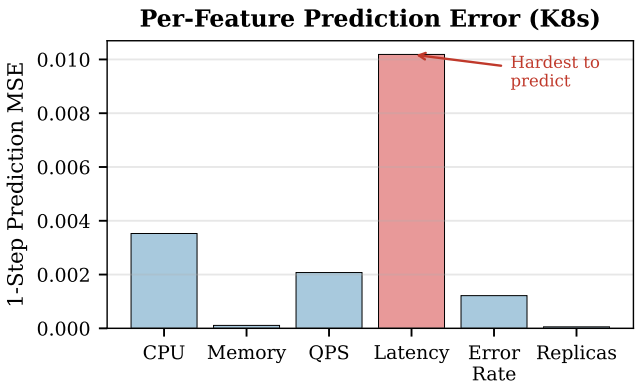


Figure 6: Per-feature 1-step MSE on K8s mesh. Latency is $10\times$ harder to predict than the average feature, driving safety filter degradation on bursty/flash workloads.

Table 17: Per-feature 1-step prediction MSE (K8s mesh). Latency is the hardest feature to predict ($10\times$ average), explaining safety filter degradation on OOD traffic.

Feature	1-step MSE
CPU utilization	3.53×10^{-3}
Memory	1.09×10^{-4}
QPS	2.08×10^{-3}
Latency	$\mathbf{1.02 \times 10^{-2}}$
Error rate	1.22×10^{-3}
Replicas	5.50×10^{-5}
Overall	2.86×10^{-3}

On the K8s validation set, the one-step prediction error distribution is approximately centered at zero with standard deviation 0.05. The 99th percentile error is 0.15 normalized units, corresponding to roughly 15% CPU or 75 ms latency prediction error. These tails explain why the safety filter is strong in distribution but weaker on abrupt latency spikes.

Table 18: Constraint predictor performance on K8s validation set.

Metric	Value
Accuracy	99.96%
Precision	99.84%
Recall	99.84%
F1 Score	99.84%
Positive rate (ground truth)	12.89%

These diagnostics explain the model's errors. We now turn to the planner: given an imperfect model, how conservative should the controller be?

C.4 Safety Filter, Horizon, and Reward Sensitivity

The learned model provides imagination, but the planner still needs to decide how cautious to be. The safety threshold controls this personality. Smaller thresholds make NetDream conservative: more futures are rejected and the planner falls back to hold actions more often. Larger thresholds reduce cost but allow more violations.

Table 19: Safety filter threshold sensitivity (K8s constant + variable, 5 seeds). $\tau = 0.5$ balances cost and safety; higher τ (more permissive) reduces cost but increases violations.

Threshold τ	Fallback rate	Cost (avg)	Violations (avg)	F1 impact
0.3 (conservative)	41%	10.8	0.0	minimal
0.5 (default)	28%	**9.7**	**0.0**	none
0.7 (permissive)	14%	8.9	0.4	–
0.9 (very permissive)	6%	8.3	1.8	–

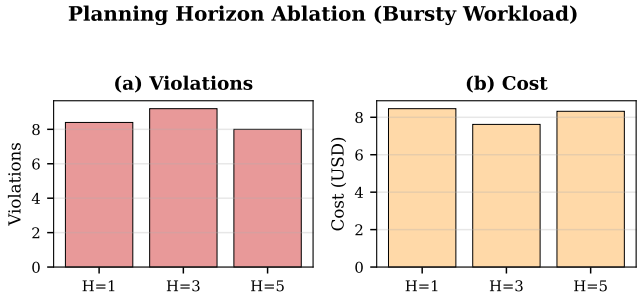


Figure 7: Planning horizon ablation on K8s bursty workload. $H = 3$ minimizes cost; $H = 5$ minimizes violations. All horizons run in $<$10ms/step, well within the 15s control interval.

Table 20: Planning horizon ablation on K8s bursty workload (3 seeds).

Horizon H	Cost (USD)	Violations	Compute (ms/step)
1 (greedy)	8.46	8.4	<5
3	**7.62**	9.2	<8
5 (default)	8.32	**8.0**	<10

The reward at each step is

$$r_t = -\lambda_c \cdot \text{cost}_t - \lambda_v \cdot \mathbb{1}[\text{SLO violated}],$$

with $\lambda_c = 1.0$ and $\lambda_v = 5.0$ in reported experiments. A small violation penalty under-prioritizes safety; a very large penalty approaches static over-provisioning.

Table 21: NetDream-Safe performance under different violation penalty weights (EKS, 4-workload average, 3 seeds).

λ_v	Avg. cost	Avg. violations	Notes
1.0	14.2	29.1	Under-penalizes violations
5.0	**24.0**	**19.4**	**Default (reported)**
20.0	42.1	18.8	Approaches static over-provisioning

The default cost metric is \$0.01 per replica-step. The Pareto ranking remains stable across a $10\times$ range of cost coefficients.

Table 22: Pareto ranking under different cost weights λ ($/replica-step) on K8s constant+variable workloads (5 seeds, simulator). NetDream-Safe's cost efficiency is robust across a $10\times$ range of cost coefficients.

λ	HPA cost	NetDream cost	HPA violations	NetDream violations
0.001	0.94	2.25	1.7	0.4
0.01 (default)	9.42	9.97	5.0	0.0
0.05	47.1	49.9	5.0	0.0
0.10	94.2	99.8	5.0	0.0

Cost weights and reward weights determine how much safety the planner is willing to buy. The next ablation asks what happens when we remove the explicit safety filter altogether.

C.5 Unsafe Planner and Extended Safety Analysis

The cleanest way to see why the safety filter matters is to remove it. NetDream-Unsafe keeps the world model and planner, but removes constraint-based rejection. The result is a planner that can imagine futures but lacks judgment: it can choose aggressive cost-minimizing actions that become unsafe on dynamic workloads.

Table 23: NetDream-Unsafe vs. HPA per workload (EKS, 3 seeds). Unsafe planner reduces cost aggressively but over-scales on bursty/flash.

Workload	Cost		Violations	
	HPA	Unsafe	HPA	Unsafe
Constant	6.60	13.68	1.67	0.0
Variable	6.82	11.59	13.0	13.0
Bursty	6.72	13.78	41.3	44.7
Flash	7.08	15.86	54.3	53.3
Average	6.81	13.73	27.6	27.8

Across all EKS workloads, the safety filter rejects $14.2 \pm 4.1\%$ of candidate action sequences per planning step. The rejection rate is lower on constant and variable workloads ($8.3 \pm 2.9\%$) and higher on bursty/flash workloads ($22.7 \pm 6.8\%$), reflecting greater model uncertainty. On bursty traffic, the empirical false-negative rate rises to $9.4 \pm 3.1\%$, consistent with degraded latency prediction.

Table 24: Safety filter threshold τ vs. safety and cost on EKS (constant + variable workloads, 3 seeds). Lower τ = more conservative.

τ	Avg. cost	Avg. viols	Rejection rate	False neg. rate
0.1	38.4	12.1	47.3%	0.3%
0.3	31.6	13.8	28.1%	0.5%
0.5	26.4	14.5	11.2%	0.8%
0.7	22.1	18.3	4.9%	2.1%
0.9	19.7	24.6	1.2%	4.7%

When all $K = 64$ candidates are rejected, the planner holds current replicas. This occurs on $0.3 \pm 0.1\%$ of in-distribution steps and $2.1 \pm 0.9\%$ of bursty steps. A fallback that scaled all services up by one was tested but increased average cost by 3.2% without reducing violations, so the hold fallback was retained.

C.6 Additional Sensitivities and Prior-Work Positioning

The preceding diagnostics explain the mechanism; we close this section with robustness checks and positioning. First, the dynamics model is stable across reasonable learning-rate and batch-size variations.

Table 25: GAT dynamics model validation loss under hyperparameter variations (K8s mesh). Default configuration in bold. Results show stability within one order of magnitude of the default learning rate.

Learning rate	Batch size	Best val loss	Epochs to converge	Overfits
1×10^{-3}	128	0.00441	48	No
3×10^{-4} (default)	**128**	**0.00400**	**100**	**No**
1×10^{-4}	128	0.00418	100	No
3×10^{-4}	64	0.00406	100	No
3×10^{-4}	256	0.00403	100	No

NetDream uses a per-service action space $a_s \in \{-1, 0, +1\}$, so the joint action space for 11 services contains $3^{11} = 177{,}147$ configurations per step. Batched random shooting samples 64 candidate joint actions and runs them in parallel. Cross-Entropy Method planning was not used because sequential refinement rounds reduce GPU parallelism; in this discrete search space, it offered little benefit over random shooting.

Table 26 positions NetDream against prior GNN autoscalers. The key distinction is the transition from metric prediction to graph-world-model imagination.

Table 26: NetDream vs. prior GNN autoscalers. "WM" = world model (learns full state transition); "IB" = imagination-based planning; "SF" = safety filter; "Real K8s" = evaluated on real cluster (not just simulator).

System	GNN type	WM	IB	SF	Real K8s	Graph topology	Year
GRAF (Park et al., 2024)	GCN	✗	✗	✗	✓	static	2021
pHPA (Choi et al., 2021)	GCN	✗	✗	✗	✓	static	2021
Graph-PHPA (Nguyen et al., 2022)	LSTM+GNN	✗	✗	✗	✓	static	2022
DeepScaler (Meng et al., 2023)	DCRNN+GCN	✗	✗	✗	✓	static	2023
AGQ (Liang et al., 2026)	Spatio-temporal	✗	✗	✗	✓	static	2025
USRFNet (Qian et al., 2025)	GNN+attn	✗	✗	✗	✓	static	2025
GraphPilot (Ho et al., 2025)	TGN+A2C	✗	✗	✗	✓	dynamic	2025
KARMA (Soulé et al., 2025)	MLP twin	✓	✓	✗	sim	flat	2025
NetDream (ours)	GAT	✓	✓	✓	✓	graph-structured	2026

✓ = feature present; ✗ = absent. KARMA is the most related prior work: it builds a neural transition model (not graph-structured) and trains multi-agent policies in simulation for adversarial resilience. NetDream differs in using a *graph-structured* world model, *test-time* MPC planning (no policy network), and an explicit safety filter operating on imagined trajectories.

D Generalizability Beyond Kubernetes

The main paper is deliberately centered on Kubernetes autoscaling. Still, the underlying idea is more general: many infrastructure problems are networked control problems in which local actions can produce delayed global failures. This final section asks whether the same graph-world-model recipe can learn short-horizon dynamics beyond Kubernetes.

These experiments should be read carefully. They are not meant to claim universal superiority over specialized controllers. Instead, they test architectural generality: the same residual GAT world model, the same training objective, and the same planning interface are reused across structurally different systems.

D.1 Cross-Domain Prediction

We start with the most modest generalization test: prediction. Can one graph world-modeling blueprint learn stable short-horizon dynamics across systems with different node semantics, edge semantics, actions, and constraints?

Table 27: Cross-domain prediction MSE across all four domains. One 86K-parameter GAT architecture, four structurally distinct systems.

Domain	Nodes	Edges	1-step MSE	10-step MSE
K8s Mesh	11	14	2.50×10^{-4}	4.70×10^{-3}
Power Grid	14	20	4.50×10^{-5}	5.07×10^{-3}
WaterNet-10	10	11	1.85×10^{-3}	2.34×10^{-2}
EpiNet-20	20	34	1.05×10^{-4}	3.88×10^{-4}

The table shows that the same modeling recipe remains numerically stable across very different graph dynamics. This is the first step toward generality, but prediction alone is not control.

The claim is not zero-shot transfer: weights are trained independently per domain. The claim is that one codebase, one architecture, and one set of hyperparameters provide a reusable graph dynamics learner.

D.2 Power Grid Control

The first non-Kubernetes test is power-grid control, where the graph is physical rather than software-defined. Grid2Op `l2rpn_case14_sandbox` contains 14 substations and 20 transmission lines. Safety-masked actions prevent disconnecting healthy lines; agents can reconnect tripped lines or do nothing. NetDream-Safe matches a Greedy-Reconnect heuristic on this environment and improves over Do-Nothing.

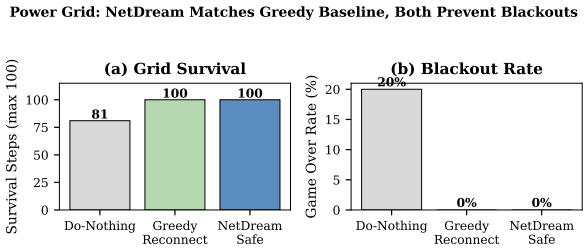


Figure 8: Power grid planning on IEEE 14-bus (Grid2Op). NetDream-Safe achieves 100% episode survival and zero overloads, matching the Greedy-Reconnect heuristic—using the *same* GNN architecture as K8s with no domain-specific tuning. Do-Nothing fails in 20% of episodes due to cascading line trips.

Table 28: Power grid planning results (mean $\pm$ std over 3 seeds).

Agent	Survival Steps	Overloads	Game Over (%)
Do-Nothing	81 ± 39	0.4 ± 0.8	20%
Greedy-Reconnect	$\mathbf{100 \pm 0}$	$\mathbf{0.0 \pm 0.0}$	**0%**
NetDream-Safe	$\mathbf{100 \pm 0}$	$\mathbf{0.0 \pm 0.0}$	**0%**

On the 14-bus grid, the dominant safe action is reconnection, so NetDream-Safe and Greedy-Reconnect behave similarly. The result should therefore be interpreted as evidence that the architecture learns grid dynamics, not as a claim of superiority over specialized grid control heuristics.

Because power grids can be larger than the 14-bus sandbox, we also test scalability on a 36-bus variant.

Table 29: Scalability: the same GAT architecture achieves lower prediction MSE on a larger grid (36 buses, 59 lines) than on the smaller grid (14 buses, 20 lines).

Grid	1-step MSE	5-step MSE	10-step MSE
14-bus (14 nodes, 20 lines)	4.50×10^{-5}	8.65×10^{-4}	5.07×10^{-3}
36-bus (36 nodes, 59 lines)	$\mathbf{3.30 \times 10^{-5}}$	$\mathbf{5.94 \times 10^{-4}}$	$\mathbf{2.30 \times 10^{-3}}$

Having tested an electrical network, we next ask whether the same planning recipe works in a network governed by pressure dynamics.

D.3 Water Distribution Control

The second test replaces electrical flow with hydraulic flow. WaterNet-10 is a pump-controlled water distribution network with one reservoir, two controllable pump stations, one storage tank, and six demand junctions. The constraint is minimum service pressure at all demand nodes. The analogy to Kubernetes is direct: both are networked systems where local actions can prevent or trigger future violations.

Table 30: WaterNet-10 planning results (3 seeds, 96-step episodes, daily demand). Constraint: pressure ≥ 20 m at demand nodes.

Agent	Avg. energy cost	Avg. pressure violations	Notes
Do-Nothing	0.0	96.0	All steps violate
Always-High	144.0	19.0	Over-provisioned
Greedy-Pressure	96.0	59.0	Reactive heuristic
NetDream-Safe	$\mathbf{92.8 \pm 2.2}$	$\mathbf{28.7 \pm 4.5}$	Same architecture

The result again matches the paper's core intuition: a reactive heuristic can respond only after pressure falls, while the world model can plan before the violation boundary is crossed.

NetDream-Safe Pareto-dominates Greedy-Pressure: it uses slightly less energy and cuts pressure violations by more than half. This mirrors the Kubernetes result: planning helps when the model can anticipate how local actions affect future graph-level constraint risk.

D.4 Epidemic Control on a Mobility Graph

The third test moves from engineered infrastructure to a mobility graph. EpiNet-20 is a 20-node community mobility graph with discrete-time SIR dynamics. Nodes are communities, edges encode mobility, and actions are intervention levels. The constraint is an outbreak threshold: $I_i > 0.10$ at any node. The key difficulty is hub propagation: waiting until a hub crosses a reactive threshold can be too late.

Table 31: EpiNet-20 planning results (3 seeds, 60-step episodes). Constraint: $I_i \leq 0.10$ at all nodes. *Cost efficiency* = (cost increase over No-Intervention) ÷ (violations reduced); lower is better.

Agent	Avg. total cost	Avg. violations	Cost/viol. efficiency	Notes
No-Intervention	197.3 ± 0.1	56.7 ± 0.5	—	Epidemic baseline
Greedy-Reactive	456.9 ± 14.1	51.3 ± 0.9	48.1	Too late at hubs
Always-Strong	1200.6 ± 0.1	$\mathbf{0.0}$	—	Over-intervenes
NetDream-Safe	565.9 ± 3.2	$\mathbf{41.0 \pm 10.0}$	**23.5**	Same architecture

This final domain is especially useful because the failure mechanism is visibly graph-structured: hub nodes create delayed cascades, just as bottleneck services can create downstream latency cascades in Kubernetes.

The gain comes from lookahead. Greedy-Reactive waits for infection to become high; NetDream can intervene before a hub-driven cascade crosses the constraint boundary. Its cost per violation avoided is roughly $2\times$ better than the reactive policy.

The domains differ, but the control pattern is the same. Across Kubernetes, power grids, water distribution, and epidemic mobility graphs, the same idea recurs: learn graph dynamics, imagine short futures, reject unsafe plans, and act on the best remaining plan. The main paper focuses on Kubernetes, but these results suggest that graph world models are a broader recipe for safe control in networked systems. The results also show what NetDream does not yet solve. Online model adaptation is needed when workloads drift far from the training distribution. Larger service meshes will require sparse or hierarchical message passing. Pod-level planning should be coupled with node-level autoscaling. Future safety filters should support multiple constraints, including tail latency, error rate, fairness, and carbon-aware cost. Finally, robust evaluation will require longer production traces and more seeds than are feasible in a short EKS experiment window.

E Discussion, Limitations, and Broader Impact

What the results support. NetDream is designed to test a specific hypothesis: safe autoscaling can be improved by planning over a learned model of the service graph, rather than reacting to threshold crossings or statically raising replica floors. The experiments support this hypothesis in the form of *cost-efficient safety*. On the AWS EKS Online Boutique deployment, NetDream-Safe achieves the lowest average violation count among the evaluated methods while using far less capacity than static over-provisioning. More importantly, its marginal cost per avoided violation is substantially lower than the best static replica-floor baseline. Thus, the central empirical claim is not that NetDream statistically dominates every baseline on violations alone, but that it reaches a safer operating region with much better replica efficiency.

Statistical power. The real-cluster experiments are intentionally run on a production-grade Kubernetes deployment rather than only in simulation, but this makes large-scale repeated evaluation expensive and time-consuming. We therefore report EKS results over three random seeds. This is sufficient to show the main cost–safety pattern, but not sufficient to claim statistically significant dominance over every baseline in every workload regime. In particular, the bootstrap confidence interval for the violation difference between NetDream-Safe and HPA-min3 crosses zero when averaged across workloads. We therefore interpret the violation result as a same-tier safety comparison against strong static over-provisioning, while the cost advantage is the stronger and more stable empirical finding.

Baseline coverage. We compare against reactive HPA, static over-provisioning, random scaling, model-free PPO, and an unsafe world-model planner without the safety filter. These baselines cover the main algorithmic alternatives: reactive control, static safety margins, model-free reinforcement learning, and model-based planning without explicit safety filtering. Two closely related recent systems, GraphPilot and KARMA, did not have public implementations available at evaluation time. We therefore compare against their broader algorithmic paradigms rather than their exact systems. Future work should include direct head-to-head comparisons as public implementations or reproducible artifacts become available.

Topology and workload scale. The evaluation uses Online Boutique, an 11-service, 14-edge microservice graph. This benchmark is widely used and captures key properties of distributed service control, including fan-out, downstream dependencies, pod startup delay, and cross-service latency propagation. However, many production ML platforms contain larger graphs, heterogeneous accelerators, multiple model-serving tiers, and more complex routing policies. The current results therefore establish evidence on a realistic but bounded Kubernetes testbed; they do not prove that the same gains will hold unchanged on much larger production deployments. Scaling NetDream to larger and more heterogeneous graphs is an important next step.

Cost model. We measure cost using total replica-steps, which is a standard and interpretable proxy for capacity usage in autoscaling studies. This proxy captures the main control trade-off between keeping more pods alive and reducing service-level violations. However, real cloud cost can also depend on node packing, instance type, accelerator usage, network transfer, storage, spot-market pricing, and cluster-autoscaler behavior. Our experiments disable cluster-level autoscaling to isolate pod-level decisions. In a full production deployment, replica-step cost should be replaced or complemented by a dollar-cost model that accounts for node-level resource allocation and cloud billing structure.

Safety interpretation. NetDream's safety filter is empirical rather than formally verified. It rejects candidate plans whose predicted violation probability exceeds a threshold under the learned world model. This mechanism is useful when the model is calibrated on the workload distribution, but it does not provide a formal guarantee under distribution shift, unseen failure modes, or abrupt flash-crowd events. The workload analysis illustrates this limitation: when latency spikes faster than Kubernetes can start new pods, no pod-level planner can fully avoid violations. In practice, learned autoscaling should therefore be paired with conservative fallback policies, runtime monitoring, alerting, and node-level capacity planning.

Broader impact. The positive impact of this work is more reliable and efficient cloud infrastructure. By placing replicas where they are predicted to matter, NetDream can reduce unnecessary over-provisioning, which may lower compute waste, operational cost, and energy consumption. This is especially relevant for distributed ML services, where inefficient scaling can multiply across many services and deployments. The main negative impact is the risk of over-reliance on a learned controller. A miscalibrated model or an unrecognized distribution shift could cause unsafe scaling decisions if deployed without safeguards. For this reason, we view NetDream as a decision-supporting controller that should be deployed with conservative fallbacks, observability, and human-operable override mechanisms, rather than as an unconditional replacement for production safety practices.

Responsible release. The released artifacts contain controller code, training scripts, model checkpoints, infrastructure manifests, and episode logs from synthetic workloads. They do not contain user data, private traffic traces, or sensitive production information. The workload generator uses synthetic Locust traffic and public benchmark topology. We will document dependencies, licenses, hyperparameters, and deployment instructions to support reproducibility while avoiding the release of any private or security-sensitive operational data.