# OFFLINE LEARNING OF FLEXIBLE AUTOSCALING POLICIES FROM CONTROLLER LOGS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Autoscaling is the primary mechanism through which cloud platforms adapt compute capacity to changing demand. Yet many production systems continue to rely on hand-designed heuristics. For example, Kubernetes primarily scales resources based on CPU utilization. Reinforcement learning (RL) offers a way to learn autoscaling policies from logs, but online exploration is impractical in these systems. Moreover, deployment objectives can vary substantially across operators and workloads, particularly since different users have different latency-cost trade-offs. Training a separate policy for each objective is computationally expensive and scales poorly with the number of users. We therefore study whether logs generated by autoscaling controllers can enable offline policy training and support different downstream objectives without retraining. Using strong controller baselines and realistic workloads in a distributed-system simulator, we answer both questions affirmatively. Our experiments show that pooling logs from multiple controllers and applying bounded local perturbations to their actions enable offline learning that matches the performance of an objective-aligned expert. Building on this result, we adapt Forward-Backward representations to learn a reward-free model from the same controller logs. After training, when queried with rewards, it comes within $1.3\%$ of policies trained for each objective on latency and rejection and outperforms a preference-conditioned baseline on a new capacity objective ($22.27$ versus $21.36$), while utilization remains harder ($5.59$ versus $6.94$). To our knowledge, this is the first demonstration of reward-free offline policy extraction for autoscaling from controller-generated logs.

## 1 INTRODUCTION

Efficient resource allocation is a central concern for cloud operators: services need enough capacity to meet demand and maintain reliability, but excess capacity incurs unnecessary cost (Flexera, 2025; Gartner, 2024; International Energy Agency, 2025). Autoscaling addresses this problem by adjusting compute resources as demand changes. Existing controllers make these decisions through threshold rules, target tracking, and feedback control (Dogani et al., 2023; Qu et al., 2018; Souza & Netto, 2015; Taherizadeh & Stankovski, 2019). Each controller encodes a particular notion of desirable operation: a latency-oriented controller may provision aggressively, while a utilization-oriented controller may operate closer to saturation. When priorities change, operators must retune or replace the controller.

Offline Reinforcement Learning (RL) (Levine et al., 2020) offers a different possibility. Deployed autoscalers already generate logs of the conditions they encounter, the allocations they choose, and the resulting service outcomes. These logs could support learning new autoscaling policies without further system interaction during training. However, training a separate offline policy for every objective leaves another problem unresolved: although the generated logs are reusable, the policy must still be retrained. Ideally, we would learn once from controller logs and specify the desired operating behavior afterward (Figure 1).

Behavior foundation models (BFMs) (Pirotta et al., 2024; Tirinzoni et al., 2025) offer a framework for this separation. Pretrained on logs, they can be conditioned on different tasks or objectives without task-specific retraining. This capability is particularly appealing for autoscaling, where operators may revise their service-cost preferences, introduce a new resource budget, or reconsider how service quality should be measured. Such changes alter what the controller should achieve



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.

without necessarily changing the system it controls. We therefore ask: *can controller-generated logs support a reward-free BFMs for autoscaling?*

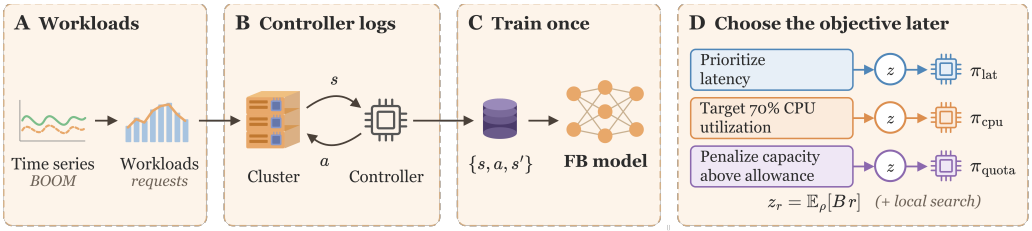



Figure 1: **Train once from controller logs; choose objectives afterward.** Workloadsdrive controller data collection; the logs train one reward-free FB model, later queried with frozen parameters for latency, utilization, or capacity objectives.

Autoscaling logs, however, come with their own set of challenges. Feedback controllers are designed to operate consistently, often choosing the same allocation under similar conditions. Even when demand varies substantially, their logs may reveal little about the consequences of alternative decisions. Our autoscaling formulation makes this distinction explicit. Workload evolves exogenously, while allocation decisions influence utilization and service outcomes. Learning reusable control behavior therefore requires both diverse workloads and logs with varied responses to allocation.

We investigate this question in a distributed search engine simulator with popular heuristic based controllers (Section 4). We first use Implicit Q-Learning (IQL) (Kostrikov et al., 2022), a canonical offline RL algorithm, to identify logs that support objective-specific offline learning. Pooling controller logs raises IQL's mean utilization return, and combining it with local action perturbations helps it match the strongest baselines (Section 4.3). We then train one reward-free Forward–Backward (FB) (Touati & Ollivier, 2021; Touati et al., 2023) model on pooled, explored logs, adapting its occupancy targets and reward queries (Sections 5.1 and 5.2). Our results suggest that the FB model can produce policies that match reward-specific and preference-conditioned multitask learning baselines, even when the reward is provided only at inference time (Section 6.1). Furthermore, the same model can generate policies that respond to rewards incorporating previously unseen penalties (Section 6.2).

## 2 Related Work

**Autoscaling and learned control.** Autoscaling combines feedback control, workload prediction, and resource models to match capacity to demand (Lorido-Botran et al., 2014; Qu et al., 2018; Dogani et al., 2023), with systems such as Autopilot (Rzadca et al., 2020) demonstrating their practical use. ScaleX (Quattrocchi et al., 2024) is a PI-style controller designed to meet Service Level Agreements (SLAs) for latency, CPU utilization, and other metrics. It has been shown to outperform existing industrial approaches and therefore serves as our main baseline. RL approaches learn allocation policies through interaction and adapt them across workloads (Qiu et al., 2023; Xue et al., 2022; Agarwal et al., 2024). We instead reuse fixed controller logs when the operating objective changes without further environment interactions.

**Offline RL and data coverage.** Offline RL enables learning from existing logs (Levine et al., 2020), whose behavioral quality and state–action coverage influence policy performance (Schweighofer et al., 2022). In autoscaling, these considerations separate into the demand encountered and the decisions observed under that demand. Diverse workloads alone need not reveal the consequences of alternative allocations. This distinction motivates our use of controller pooling and local exploration to broaden the experience available for subsequent objectives.

**Reusing behavior across objectives.** Preference-conditioned policies accommodate changing trade-offs among reward components specified during training (Abels et al., 2019; Zhu et al., 2023; Lin et al., 2024). Successor features separate future feature occupancies from reward weights (Barreto et al., 2018); Forward–Backward representations extend this principle by learning successor occupancies that can be queried with rewards supplied after pretraining (Touati & Ollivier, 2021; Touati et al., 2023; Jeen et al., 2024). Such reuse underpins BFMs for imitation, humanoid control, and language-directed

tasks (Pirotta et al., 2024; Tirinzoni et al., 2025; Sikchi et al., 2025a). Related approaches construct behavior-space bases (Agarwal et al., 2025), learn latent-predictive representations (Bagatella et al., 2026; Jajoo et al., 2026), or model future occupancy with generative models (Janner et al., 2020; Farebrother et al., 2025; Zheng et al., 2026b). We apply such reuse to changing autoscaling objectives.

**Representation and task inference.** The effectiveness of reuse depends on both the learned representation and how a task selects behavior. Recent work analyzes learnability and low-rank approximation limits of successor representations (Zheng et al., 2026a; Dubail et al., 2025; Azad et al., 2026). At inference, fast adaptation improves behavior through online interaction (Sikchi et al., 2025b), while optimistic task inference uses interaction to identify an initially unknown reward (Rupf et al., 2026). Our setting supplies the reward explicitly and selects among represented policies using frozen predictions on logged states, without further interaction.

## 3 Problem Setup

We ask whether existing controllers' logs can support objectives specified after training.

### 3.1 A factored MDP for autoscaling

We model autoscaling as a continuing factored MDP Guestrin et al. (2003); Mohan et al. (2024) $(\mathcal{S}, \mathcal{A}, P, \gamma)$ with five-minute decisions and $\gamma = 0.99$. Data collection and evaluation use finite windows of this process. The state separates workload, allocation, and service measurements:

$$s_t = (\nu_t, c_t, y_t), \qquad c_t = (n_t, q_t), \qquad y_t = (\rho_t, \ell_t, u_t, \mathrm{rss}_t, b_t). \tag{1}$$

where $\nu_t$ is the request count during the preceding interval, $c_t$ specifies the number of workers $n_t$ and capacity units per worker $q_t$, and $y_t$ contains rejection rate $\rho_t$, mean request latency $\ell_t$, CPU utilization $u_t$, memory utilization $\mathrm{rss}_t$, and a steady-state indicator $b_t$. Section C gives the full encoding. In the simulator we use for experimentation, each action specifies the allocation for the next interval:

$$a_t \in \mathcal{A} = \{1, \ldots, 42\} \times \{0.5, 1, 2, 4, 8, 16\}, \tag{2}$$

which gives 252 possible allocations.

**Factored dynamics.** Incoming demand evolves independently of the autoscaler's decisions, while the resulting service measurements depend on both demand and allocation. We write

$$P(\nu', c', y' \mid \nu, c, y, a) = P_\nu(\nu' \mid \nu)\, \mathbf{1}\{c' = a\}\, P_y(y' \mid \nu, c, y, \nu', a). \tag{3}$$

Thus, workload is exogenous but affects the controlled system. The same allocation can yield different service outcomes under different demand. This assumes a Markov workload description.[1]

**Operating objectives** The reward family combines penalties for rejection, latency, provisioned capacity, and deviations from a target utilization; the weights $\omega$ specify the operating priorities:

$$r_\omega = 1 - \omega^\top \mathbf{p}, \qquad \mathbf{p} = \left(p_{\mathrm{rej}}, p_{\mathrm{lat}}, p_{\mathrm{cost}}, p_{\mathrm{under}}, p_{\mathrm{over}}\right)^\top, \qquad \omega \geq 0. \tag{4}$$

We evaluate single-priority objectives $r_{\mathrm{util}} = 1 - p_{\mathrm{under}} - p_{\mathrm{over}}$, $r_{\mathrm{lat}} = 1 - p_{\mathrm{lat}}$, and $r_{\mathrm{rej}} = 1 - p_{\mathrm{rej}}$. The utilization terms penalize deviations below and above 70% CPU utilization, while the capacity term measures provisioned resources (Section C.2).

All penalties are determined by the resulting state $s_{t+1}$, which contains both the applied allocation and its service outcomes. We therefore also write the reward as $r_\omega(s_{t+1})$.

For a downstream state reward $r$, the discounted learning objective is

$$J_r^\gamma(\pi) = \mathbb{E}_{s_0 \sim \mu_0, \pi, P}\left[\sum_{t=0}^{\infty} \gamma^t r(s_{t+1})\right], \tag{5}$$

[1] The observed request count is a workload summary. An exact Markov description of a general trace-driven process may require additional workload history or state.

where $\mu_0$ is the initial-state distribution. Evaluation uses the undiscounted return over a fixed window.

Changing $\omega$ changes the trade-off among the predefined penalties. An operator may instead change the reward itself, for example by penalizing only capacity above a reserved allowance rather than all provisioned capacity (Section 6.2). Such an objective cannot generally be expressed by reweighting the predefined penalties. We evaluate both forms of change.

### 3.2 Offline data and reward availability

The offline corpus is generated by a mixture of behavior controllers $\{\beta_j\}_{j=1}^{J}$. For each collected episode,

$$j_i \sim \lambda, \qquad \tau_i \sim p(\tau \mid \beta_{j_i}), \qquad \mathcal{D} = \{(s_{i,t}, a_{i,t}, s_{i,t+1})\}_{i,t}, \tag{6}$$

where $\lambda_j$ denotes the contribution of controller $\beta_j$ to the corpus.[2] In the resulting fixed dataset $\mathcal{D}$, $a_{i,t}$ is the allocation actually applied.

We distinguish three settings. A *reward-specific* learner receives its objective before training. A *preference-conditioned* learner receives a reward family before training and a preference $\omega$ at inference. A *reward-free* learner receives its downstream reward only after pretraining. In the reward-free setting, all model parameters remain frozen while the requested reward is used to select a policy.

**Successor measures.** For a policy $\pi$, the successor measure records discounted future visits to a measurable state set $E$:

$$M^{\pi}(s, a, E) := \sum_{k=0}^{\infty} \gamma^k \Pr_{\pi}(s_{k+1} \in E \mid s_0 = s, a_0 = a). \tag{7}$$

It weights states likely to be reached after $(s, a)$, with earlier visits weighted more. Let $\rho_D$ denote the successor-state distribution sampled by the logging process, with expectations estimated from logged successor states. Where $M^{\pi}(s, a, \cdot)$ admits a density relative to $\rho_D$, write

$$M^{\pi}(s, a, \mathrm{d}s^+) = m^{\pi}(s, a, s^+)\rho_D(\mathrm{d}s^+). \tag{8}$$

where $m_t^{\pi}(s, a, s^+)$ is the successor density relative to the states represented in the offline data. For any bounded state reward $r$, integrating this density gives

$$Q_r^{\pi}(s, a) = \mathbb{E}_{s^+ \sim \rho_D}\left[m^{\pi}(s, a, s^+) r(s^+)\right]. \tag{9}$$

## 4 Collecting controller logs for offline learning

We use the factored dynamics to guide workload generation and controller-log collection, then assess the resulting data with IQL and behavior cloning.

### 4.1 Two sources of coverage

Under a shared workload process and initialization, the normalized discounted visitation over $T$ decisions (Appendix C.1) factors as

$$d_T^{\pi}(\nu, c, y, a) = d_{\nu,T}(\nu)\, d_T^{\pi}(c, y, a \mid \nu). \tag{10}$$

The workload marginal is policy-independent, while the conditional distribution reflects allocation decisions and their service outcomes. Thus, workload diversity determines which demand conditions appear, and controller behavior determines the decisions observed under them. Varying demand alone need not provide evidence about alternative allocations at comparable system states.

### 4.2 Constructing the offline datasets

**Simulation environment.** We collect transitions in a discrete-event simulator (Amazon Science, 2025) that takes synthetic requests and generates simulated system metrics CPU, memory, disk resources, and scaling overhead. An allocation $(n, q)$ provides $C = nq$ compute units (CUs), each comprising one vCPU and 6 GiB RAM. Scaling changes capacity and temporarily disrupts service through reconfiguration. Appendix F describes the simulator and its validation against two workloads on OpenSearch Serverless.

[2] For locally explored data, $\beta_j$ includes the corresponding exploration mechanism.

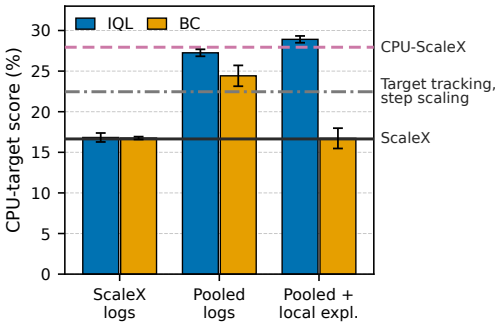



Figure 2: **Collection choices affect reward-based learning and imitation differently.** Utilization score (100% at exactly 70% CPU); means ± s.e.; lines mark the pooled controllers.

| | Utilization | | Latency | | Rejection | |
|---|---|---|---|---|---|---|
| Logs | IQL | BC | IQL | BC | IQL | BC |
| ScaleX | 4.04 | 4.02 | 19.13 | 19.21 | 23.54 | 23.55 |
| Pooled | 6.54 | 5.86 | 18.92 | 18.45 | 23.21 | 22.84 |
| + local expl. | **6.94** | 4.01 | 18.80 | 17.90 | 23.04 | 22.16 |
| ScaleX | 4.00 | | 19.28 | | 23.51 | |
| CPU-ScaleX | 6.71 | | 15.96 | | 22.57 | |

Table 1: **Local exploration and pooling are routes to the utilization gain.** Return per objective, five-seed means (s.e. $\leq 0.38$); ScaleX logs have 100k transitions, the pools 402k.

**Workload diversity.** We split BOOM's 954 rate-type observability series into training, validation, and test sets before workload generation (Cohen et al., 2025). Their standardized values supply temporal patterns; we set mean arrival rates to 0.1, 1, or 10 requests/s and sample piecewise-constant Poisson arrivals. Five CPU–memory requirement profiles vary resource pressure independently of temporal shape. Section E gives the construction and split sizes.

**Controller diversity and local exploration.** On a fixed subset of 400 of the 10,800 training workloads (40 BOOM series, each at 1 and 10 requests/s with all five CPU–memory profiles), we collect (i) latency-oriented ScaleX logs; (ii) pooled logs from ScaleX, CPU-ScaleX, target tracking, and step scaling; and (iii) logs from the same pool with bounded, temporally correlated allocation perturbations. Perturbations are executed during collection; transitions record the applied action and resulting state. Appendix H has more detail for all the controllers.

**Collection protocol.** Each controller replays the same workload segments from a reset simulator and records 12 decisions with their actions and system metrics per episode, about 8,370 episodes per controller; the fixed corpora contain approximately 100k, 402k, and 402k transitions (33,482 episodes in each pooled corpus) and 82, 168, and 252 distinct allocations, respectively. Pooling changes both controller composition and data volume; the local-exploration comparison uses the same pool and approximately matched volume (Sections G.1 and K).

### 4.3 Evaluating the usefulness of controller logs

We test whether these collection choices support a change in operating priority, from ScaleX's latency-oriented behavior to a target CPU utilization of 70%. The reward is $r_{\text{util}} = 1 - p_{\text{under}} - p_{\text{over}}$, which penalizes deviations on either side of the target. We train Implicit Q-Learning (IQL) (Kostrikov et al., 2022) on each corpus to test whether reward-aware learning can exploit the logged consequences, and behavior cloning (BC) to assess what can be recovered by imitating the collected actions. CPU-ScaleX provides an objective-aligned reference and also contributes to the pooled data. The target differs from the base controller's priority, but is represented among the contributing controllers.

**Training and evaluation.** We train each learner for 100k updates with five random seeds. All policies are evaluated on the same 266 held-out windows from 90 BOOM-derived test traces, selected independently of learned-policy performance. Each window contains 12 warmup and 24 scored decisions, with common initialization and controller resets; learned policies act greedily. We report undiscounted return in $[0, 24]$, higher meaning closer to the utilization target, with standard errors across the five seed means (Appendix G).

On ScaleX logs, IQL and BC obtain 4.04 and 4.02, close to ScaleX's 4.00 (Figure 2). Pooling raises IQL to 6.54 and BC to 5.86. With local exploration, IQL improves further to $6.94 \pm 0.10$, exceeding

CPU-ScaleX's 6.71, while BC falls to 4.01. IQL also exceeds target tracking and step scaling on this objective, although ScaleX remains stronger on latency and rejection (Appendix A.1).

Adding local exploration to the pool raises IQL's return by 0.40 while reducing BC's, indicating that the resulting experience helps reward-aware learning even when imitation suffers. Having established that these logs support objective-specific learning, we use the pooled, explored corpus to test whether one model can support objectives supplied only after training.

# 5 Reward-Free Reuse of Controller Logs

Section 4 established that pooled controller logs with local exploration support objective-specific policy improvement. We now ask whether a single model trained on this corpus can serve different operating objectives without retraining. Our approach separates learning the consequences of allocation decisions from specifying how those consequences should be valued. We first learn a family of policies and their successor occupancies without downstream rewards during training (Section 5.1). We then map a supplied reward to a policy query during inference (Section 5.2) and optionally refine policy selection using the model's frozen predictions (Section 5.3). Both inference procedures use only the learned model and logged transitions, with no parameter updates or additional environment interaction.

## 5.1 Reward-free pretraining

Forward–Backward (FB) representations (Touati & Ollivier, 2021; Touati et al., 2023) approximate the successor density defined in Equation (8) for a family of policies $\{\pi_z\}$ indexed by vectors $z \in \mathbb{R}^d$:

$$m^{\pi_z}(s, a, s^+) \approx F_\theta(s, a, z)^\top B_\psi(s^+). \tag{11}$$

Here, $F_\theta$ and $B_\psi$ are neural networks with parameters $\theta$ and $\psi$ and $d$-dimensional outputs. The backward map encodes a possible future state $s^+$; the forward map supplies coefficients whose inner product with that encoding predicts its successor density under $\pi_z$. We use $d = 100$ and two forward networks, with $F_\theta$ denoting their mean. Appendix D.3 gives architecture details.

A categorical actor $\pi_\phi(a \mid s, z)$, with parameters $\phi$, assigns probabilities to the allocations in Equation (2). Its greedy action defines the policy used for continuation predictions and deployment:

$$\pi_z(s) \in \arg\max_{a \in \mathcal{A}} \pi_\phi(a \mid s, z). \tag{12}$$

During pretraining, task vectors lie on the sphere $\|z\|_2 = \sqrt{d}$. We mix random directions, normalized backward embeddings of logged states, and previously sampled task vectors; the sampling probabilities and task buffer are specified in Appendix D.4. These vectors index training tasks without requiring downstream reward labels.

**Learning successor occupancies.** Separating the first transition from later visits in Equation (7) gives the successor-measure Bellman recursion (Touati et al., 2023):

$$M^{\pi_z}(s, a, E) = P(E \mid s, a) + \gamma \mathbb{E}_{s' \sim P(\cdot \mid s, a)} \left[ M^{\pi_z}(s', \pi_z(s'), E) \right], \tag{13}$$

for any measurable state set $E \subseteq \mathcal{S}$. The first term accounts for the immediate successor; the second accounts for subsequent visits under the same policy. We fit Equation (11) using the FB temporal-difference objective (Touati et al., 2023, Section 5.2): a squared term fits discounted continuation predictions, and a linear term incorporates the observed successor. Continuation predictions use target networks $\bar{F}$ and $\bar{B}$, copies updated by exponential moving averages of the learned parameters and held constant during each gradient update. Appendix D.4 gives the minibatch loss, backward-feature regularizer, target updates, and optimization schedule.

**Learning the policy family.** We train the actor by advantage-weighted regression (AWR) (Peng et al., 2019; Cetin et al., 2024), weighting logged actions by their estimated advantage; for task $z$,

$$Q_z(s, a) = F_\theta(s, a, z)^\top z, \qquad A_z(s, a) = Q_z(s, a) - \sum_{b \in \mathcal{A}} \pi_\phi(b \mid s, z) Q_z(s, b). \tag{14}$$

The actor fits logged actions using exponentiated, standardized advantages. Appendix D.4, paragraph "Actor extraction," specifies our modified weight normalization, loss, and detached quantities. Occupancy and actor updates use only the transitions in Equation (6) and sampled task vectors.

**Stabilizing occupancy targets.** We cap the scale of the continuation predictions used in the occupancy loss. For a minibatch of $N$ transitions $(s_i, a_i, s'_i)$ with tasks $z_i$, the target matrix is

$$\bar{M}_{ij} = \bar{F}(s'_i, \pi_{z_i}(s'_i), z_i)^\top \bar{B}(s'_j), \tag{15}$$

where $\bar{F}$ is the mean of the two target forward networks. Row $i$ predicts continuation occupancy after transition $i$, and column $j$ evaluates that prediction at reference state $s'_j$. We cap each row's mean positive mass:

$$m_i^+ = \frac{1}{N}\sum_{j=1}^{N}[\bar{M}_{ij}]_+, \qquad \widetilde{M}_{ij} = \bar{M}_{ij}\frac{C_{\text{mass}}}{\max(C_{\text{mass}}, m_i^+)}, \qquad C_{\text{mass}} = 21, \tag{16}$$

Here, $[v]_+ = \max(v, 0)$; rescaling precedes discounting and preserves row ratios. The cap is a heuristic motivated by the 24-step discounted mass ($\approx 21.4$; continuing-task mass is 100). In a comparison on an earlier plain-MLP model, a looser cap of 100 diverges on all seeds while the cap of 21 completes 100k updates (Appendix A.3).

### 5.2 Constructing a reward query

After pretraining, a supplied state reward selects a policy from the learned family. Substituting Equation (11) into Equation (9) gives the FB reward readout (Touati & Ollivier, 2021):

$$Q_r^{\pi_z}(s, a) \approx F_\theta(s, a, z)^\top \xi_r. \tag{17}$$

$$\xi_r = \mathbb{E}_{s^+\sim\rho_{\mathcal{D}}}\left[B_\psi(s^+)r(s^+)\right]. \tag{18}$$

The reward embedding $\xi_r$ specifies how the predicted occupancy is valued. Its expectation uses the same reference distribution $\rho_{\mathcal{D}}$ that defines the successor density in Equation (8). We estimate it using 2,000 states sampled without replacement from the training corpus, collected in $\mathcal{S}_{\text{ref}}$ (Appendix D.5).

**Centering and normalizing the query.** For continuing discounted control, adding a constant $b$ to the reward changes every policy's value by $b/(1-\gamma)$ and leaves action rankings unchanged. However, Equation (18) gives $\xi_{r+b} = \xi_r + b\,\mathbb{E}_{\rho_{\mathcal{D}}}[B_\psi(s)]$. This changes the raw query direction when the mean backward feature is nonzero, which can change the behavior selected by an approximate learned model. We therefore center support-set rewards before projecting onto the training task sphere:

$$z_r = \text{proj}_z\left[\frac{1}{|\mathcal{S}_{\text{ref}}|}\sum_{s\in\mathcal{S}_{\text{ref}}} B_\psi(s)\big(r(s) - \bar{r}_{\text{ref}}\big)\right], \tag{19}$$

where $\bar{r}_{\text{ref}} = |\mathcal{S}_{\text{ref}}|^{-1}\sum_{s\in\mathcal{S}_{\text{ref}}} r(s)$ and $\text{proj}_z(v) = \sqrt{d}\,v/\|v\|_2$ for $v \neq 0$. Centering removes dependence on additive reward offsets, and projection matches the task radius defined in Section 5.1. Direct inference executes $\pi_{z_r}$ using Equation (12), with all parameters frozen. On frozen checkpoints of that earlier model, centering produces distinct operating points where uncentered queries concentrate around a common direction (Appendix A.3).

This interface accepts rewards that can be evaluated from the represented state in Equation (1), including functions outside the predefined penalty basis in Equation (4). A new operating objective therefore changes the reward query while reusing the pretrained representation and actor.

### 5.3 Inference-time policy search

Approximation and actor-extraction errors can make another policy in the learned family outperform the direct query on the supplied reward. We therefore also evaluate an optional search over policy indices, keeping the requested reward fixed. The candidate set $\mathcal{Z}_r$ contains $z_r$ and 256 vectors of the form $\text{proj}_z(z_r + 0.5\epsilon)$, with independent $\epsilon \sim \mathcal{N}(0, I_d)$ and projection as defined after Equation (19). Appendix D.5 also specifies the wider-search variant.

For each candidate $z$, Equation (12) supplies a policy $\pi_z$. Using the value readout in Equation (17), we score its predicted occupancy against the same fixed query $z_r$ on 512 states $\mathcal{S}_{\text{score}} \subset \mathcal{S}_{\text{ref}}$:

$$\widehat{S}_r(z) = \frac{1}{|\mathcal{S}_{\text{score}}|}\sum_{s\in\mathcal{S}_{\text{score}}} F_\theta(s, \pi_z(s), z)^\top z_r, \qquad z_r^\star \in \arg\max_{z\in\mathcal{Z}_r}\widehat{S}_r(z). \tag{20}$$

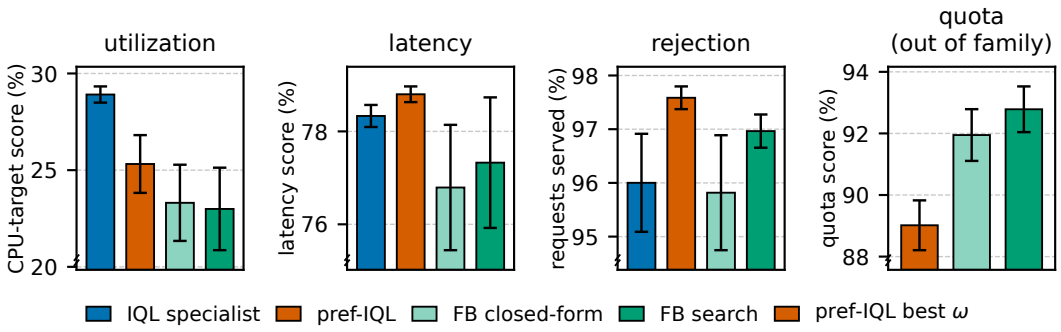



Figure 3: **Reuse nears specialists on latency and rejection; utilization is harder.** Score = $100\times$return$/24$; five-seed means $\pm$ s.e.; quota uses preference-IQL's best tested weights.

The candidate $z$ selects the policy; the fixed query $z_r$ specifies how it is scored. Centering and normalization make $\widehat{S}_r$ a ranking score rather than a return in original reward units (Appendix D.5). The selected vector remains fixed during deployment. Including $z_r$ guarantees a non-decreasing predicted score, but approximation error and differences between scoring and evaluation can prevent a gain in realized return (Section 6.2).

## 6 Evaluating Reward-Free Reuse

We evaluate reuse across operating objectives (Section 6.1), adaptation to a new capacity reward, and policy selection with frozen predictions (Section 6.2).

**Training and evaluation.** All models train for 100k updates on the pooled logs with local exploration. We consider two baselines: a) IQL (Kostrikov et al., 2022) trained separately for each reward; b) a multi-task baseline preference-IQL, which conditions its critics, value function, and actor on reward weights. Our method FB receives rewards after pretraining and reuses one frozen checkpoint per seed. Following Section 4.3, we report undiscounted returns on 266 held-out windows, summarized as five-seed means $\pm$ standard errors. Returns lie in $[0, 24]$ for these objectives; comparisons are within each reward. FB uses the same query settings for every objective; quota comparisons use test-selected preferences, as specified below. Section D and Appendix G give details.

### 6.1 Reusing one model across operating objectives

We query the utilization, latency, and rejection objectives of Equation (4). Preference-IQL uses the corresponding weights;[3] FB uses closed-form queries or local search (Section 5.2–Section 5.3). FB search approaches IQL on latency (18.56 versus 18.80) and rejection (23.27 versus 23.04), but closed-form FB trails IQL on utilization (5.59 versus 6.94; Figure 3). All five seeds are included: one returns 3.88 on utilization, and the other four also remain below the specialist mean (Appendix A.3).

### 6.2 Adapting to a new capacity objective

We supply a capacity-threshold reward absent from preference-IQL's training family:

$$r_{\text{quota},Q}(s') = 1 - 0.25p_{\text{rej}}(s') - 0.15p_{\text{lat}}(s') - 0.6\,\text{clip}\left(\tfrac{C(s')-Q}{8}, 0, 1\right), \tag{21}$$

where $C(s') = n'q'$ and allowance $Q$ are in compute units (CUs). Preference-IQL receives its best tested query on the test suite. At $Q = 20$, closed-form FB and FB search return $22.07 \pm 0.20$ and $22.27 \pm 0.18$, versus $21.36 \pm 0.19$ for preference-IQL. FB search uses more capacity (10.32 versus 8.30 CU) but exceeds the allowance less often (2.1% versus 7.2%; Figure 4).

Re-querying the same checkpoints at $Q \in \{10, 20, 40\}$ gives higher mean returns than the best tested preference at each allowance (Figure 5 and Appendix A.4). At $Q = 10$, re-querying reduces searched

[3] Its training basis also includes switching and resizing penalties (Section D).

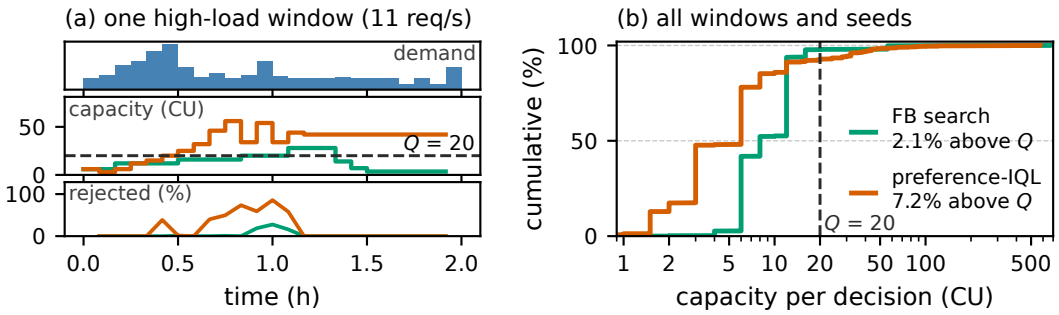



Figure 4: **FB exceeds the allowance less often while using more capacity on average.** (a) One high-load window, $Q = 20$ CU. (b) Capacity per decision over all windows and seeds.

FB's overshoot from 48% with the unchanged $Q = 20$ query to 19%. These rewards do not enforce a hard capacity limit.

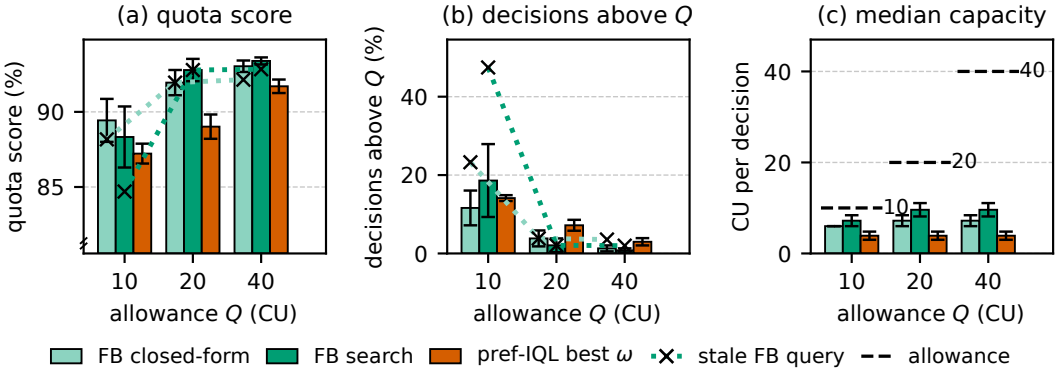



Figure 5: **Re-querying frozen models as the capacity allowance changes.** Quota score ($100\times$return$/24$), decisions exceeding $Q$, and median capacity; crosses keep the $Q = 20$ FB query.

**Selecting policies with frozen predictions.** Local search changes return by $+0.13$ (latency), $+0.28$ (rejection), $+0.20$ (quota), and $-0.08$ (utilization); wider search offers no consistent gain (Figure 6).

## 7 Conclusion

We demonstrate, to our knowledge, the first reward-free offline policy extraction for autoscaling from controller-generated logs. Pooling controller logs with bounded local action perturbations supports both objective-specific offline learning and reward-free FB pretraining. The resulting model can be queried with objectives specified after training, including a capacity penalty outside the preference baseline's reward family, while keeping its parameters frozen.

**Limitations and future work.** Our model handles some objectives better than others, and learning from a wider range of controllers may help close this gap. We plan to include predictive and adaptive autoscalers and study which kinds of experience make the learned model more broadly reusable. We also want to understand how far this reuse extends to new workload families and resource configurations, and when adaptation requires additional logs or training. These experiments currently take place in simulation; deploying the learned policies on real systems would let us test whether their flexibility carries over to the variability and scaling overheads of production.

## AI USAGE STATEMENT

In this work, we used generative AI tools to refine mathematical claims, experimental design and interpretation of results. We have not used generative AI tools for generating synthetic datasets, implementing methods, translating content, or cleaning and reformatting datasets, and qualitative and thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to edit the manuscript for readability. We have reviewed all AI-assisted work. Specifically, we independently checked mathematical claims and proofs, verified literature references against their original sources, and checked interpretations against experimental results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES

A. Abels, D. Roijers, T. Lenaerts, A. Nowé, and D. Steckelmacher. Dynamic weights in multi-objective deep reinforcement learning. In K. Chaudhuri and R. Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, volume 97, pp. 11–20. Proceedings of Machine Learning Research, 2019.

S. Agarwal, M. Rodriguez, and R. Buyya. A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions. *IEEE Transactions on Services Computing*, (5): 1899–1910, 2024.

S. Agarwal, H. Sikchi, P. Stone, and A. Zhang. Proto successor measure: Representing the behavior space of an rl agent. In *Proceedings of the 42nd International Conference on Machine Learning (ICML'25)*, 2025.

Amazon Science. Distributed search engine simulator. GitHub repository, 2025.

S. Azad, J. Hoffmann, I. Nematollahi, H. Zhu, A. Valada, and J. Boedecker. Spectral alignment in forward-backward representations via temporal abstraction. *arXiv preprint arXiv:2603.20103*, 2026.

M. Bagatella, M. Pirotta, A. Touati, A. Lazaric, and A. Tirinzoni. TD-JEPA: Latent-predictive representations for zero-shot reinforcement learning. In *The Fourteenth International Conference on Learning Representations (ICLR'26)*, 2026.

A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. van Hasselt, and D. Silver. Successor features for transfer in reinforcement learning, 2018.

D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.). *Proceedings of the 39th International Conference on Advances in Neural Information Processing Systems (NeurIPS'25)*, 2025. Curran Associates.

E. Cetin, A. Touati, and Y. Ollivier. Finer behavioral foundation models via auto-regressive features and advantage weighting. *Reinforcement Learning Journal*, 2024.

B. Cohen, E. Khwaja, Y. Doubli, S. Lemaachi, C. Lettieri, C. Masson, H. Miccinilli, E. Ramé, Q. Ren, A. Rostamizadeh, J. O. du Terrail, A. Toon, K. Wang, S. Xie, Z. Xu, V. Zhukova, D. Asker, A. Talwalkar, and O. Abou-Amal. This time is different: An observability perspective on time series foundation models, 2025.

J. Dogani, R. Namvar, and F. Khunjush. Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. *Computer Communications*, 209:120–150, 2023. doi: 10.1016/j.comcom.2023.06.010.

B. Dubail, S. Stojanovic, and A. Proutiere. Shift before you learn: Enabling low-rank representations in reinforcement learning. In Belgrave et al. (2025).

J. Farebrother, M. Pirotta, A. Tirinzoni, R. Munos, A. Lazaric, and A. Touati. Temporal difference flows. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

Flexera. 2025 state of the cloud report. Technical report, Flexera, 2025. See also the press release: New Flexera Report Finds that 84% of Organizations Struggle to Manage Cloud Spend.

Gartner. Gartner forecasts worldwide public cloud end-user spending to total $723 billion in 2025. Press release, November 2024.

C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient solution algorithms for factored MDPs. *jair*, 19:399–468, 2003.

International Energy Agency. Energy and ai. Technical report, International Energy Agency, 2025.

P. Jajoo, H. Sikchi, S. Agarwal, A. Zhang, S. Niekum, and M. White. Regularized latent dynamics prediction is a strong baseline for behavioral foundation models. In *The Fourteenth International Conference on Learning Representations (ICLR'26)*, 2026.

M. Janner, I. Mordatch, and S. Levine. Gamma-models: Generative temporal difference learning for infinite-horizon prediction. In H. Larochelle, M. Ranzato, R. Hadsell, M.-F. Balcan, and H. Lin (eds.), *Proceedings of the 34th International Conference on Advances in Neural Information Processing Systems (NeurIPS'20)*. Curran Associates, 2020.

S. Jeen, T. Bewley, and J. Cullen. Zero-shot reinforcement learning from low quality data. *Advances in Neural Information Processing Systems*, 37:16894–16942, 2024.

I. Kostrikov, A. Nair, and S. Levine. Offline reinforcement learning with implicit Q-learning. In *The Tenth International Conference on Learning Representations (ICLR'22)*, 2022.

S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv:2005.01643 [cs.LG]*, 2020.

Q. Lin, C. Yu, Z. Liu, and Z. Wu. Policy-regularized offline multi-objective reinforcement learning. *arXiv preprint arXiv:2401.02244*, 2024.

T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

A. Mohan, A. Zhang, and M. Lindauer. Structure in deep reinforcement learning: A survey and open problems. *Journal of Artificial Intelligence Research*, 79, 2024.

X. Peng, A. Kumar, G. Zhang, and S. Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*, 2019.

M. Pirotta, A. Tirinzoni, A. Touati, A. Lazaric, and Y. Ollivier. Fast imitation via behavior foundation models. In *The Twelfth International Conference on Learning Representations (ICLR'24)*, 2024.

H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference*, pp. 387–402. USENIX Association, 2023.

C. Qu, R. N. Calheiros, and R. Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, 51(4), 2018. doi: 10.1145/3148149.

G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani, and L. Baresi. Autoscaling solutions for cloud applications under dynamic workloads. *IEEE Transactions on Services Computing*, 17(3):804–820, 2024.

T. Rupf, M. Bagatella, M. Vlastelica, and A. Krause. Optimistic task inference for behavior foundation models. In *The Fourteenth International Conference on Learning Representations (ICLR'26)*, 2026.

K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

K. Schweighofer, A. Radler, M.-C. Dinu, M. Hofmarcher, V. Patil, A. Bitto-Nemling, H. Eghbal-zadeh, and S. Hochreiter. A dataset perspective on offline reinforcement learning, 2022.

H. Sikchi, S. Agarwal, P. Jajoo, S. Parajuli, C. Chuck, M. Rudolph, P. Stone, A. Zhang, and S. Niekum. Rlzero: Direct policy inference from language without in-domain supervision. In Belgrave et al. (2025).

H. Sikchi, A. Tirinzoni, A. Touati, Y. Xu, A. Kanervisto, S. Niekum, A. Zhang, A. Lazaric, and M. Pirotta. Fast adaptation with behavioral foundation models. *arXiv:2504.07896*, 2025b.

A. Souza and M. Netto. Using application data for sla-aware auto-scaling in cloud environments. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 252–255. IEEE, 2015.

S. Taherizadeh and V. Stankovski. Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, 62(2):174–197, 2019.

A. Tirinzoni, A. Touati, J. Farebrother, M. Guzek, A. Kanervisto, Y. Xu, A. Lazaric, and M. Pirotta. Zero-shot whole-body humanoid control via behavioral foundation models. In *The Thirteenth International Conference on Learning Representations (ICLR'25)*, 2025.

A. Touati and Y. Ollivier. Learning one representation to optimize all rewards. In M. Ranzato, A. Beygelzimer, K. Nguyen, P. Liang, J. Vaughan, and Y. Dauphin (eds.), *Proceedings of the 35th International Conference on Advances in Neural Information Processing Systems (NeurIPS'21)*. Curran Associates, 2021.

A. Touati, J. Rapin, and Y. Ollivier. Does zero-shot reinforcement learning exist? In *The Eleventh International Conference on Learning Representations (ICLR'23)*, 2023.

S. Xue, C. Qu, X. Shi, C. Liao, S. Zhu, X. Tan, L. Ma, S. Wang, S. Wang, Y. Hu, L. Lei, Y. Zheng, J. Li, and J. Zhang. A meta reinforcement learning approach for predictive autoscaling in the cloud. In A. Zhang and H. Rangwala (eds.), *Proceedings of the 28th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'22)*, pp. 4290–4299. ACM Press, August 2022. doi: 10.1145/3534678.3539063.

C. Zheng, R. Jayanth, and B. Eysenbach. Can we really learn one representation to optimize all rewards? *arXiv preprint arXiv:2602.11399*, 2026a.

C. Zheng, S. Park, S. Levine, and B. Eysenbach. Intention-conditioned flow occupancy models. In *The Fourteenth International Conference on Learning Representations (ICLR'26)*, 2026b.

B. Zhu, M. Dang, and A. Grover. Scaling pareto-efficient decision making via offline multi-objective rl. *arXiv preprint arXiv:2305.00567*, 2023.

# Appendix

## A Additional experimental results

### A.1 Reference controllers and objective-specific improvement

Table 6 gives the complete comparison of reward-specific IQL, behavior cloning, and rule controllers on pooled logs with local exploration. The standard errors describe variation across five training seeds, each evaluated on the same 266 windows. Rule controllers are deterministic and are evaluated once. The utilization improvement does not imply improvement on every objective: ScaleX remains stronger on latency and rejection.

### A.2 Preference-conditioned reuse

We freeze preference-IQL and vary its weights along two families (Table 2) with cost weight $c \in \{0, 0.25, 0.5, 0.75, 1\}$: a service–cost family that weights rejection and latency equally against capacity ($\omega_{\text{rej}} = \omega_{\text{lat}} = (1-c)/2$, $\omega_{\text{cost}} = c$) and a rejection–cost family ($\omega_{\text{rej}} = 1 - c$, $\omega_{\text{cost}} = c$). Increasing the cost weight lowers provisioned capacity monotonically in both families, from $70.4$ and $60.6$ CU to $8.3$ CU. Each point is scored under its own reward, so returns at different points are not a ranking. Where a reward-specific IQL specialist exists on the same corpus, the frozen preference model matches or exceeds it: $23.42 \pm 0.05$ versus $23.04 \pm 0.22$ for rejection only and $23.31 \pm 0.04$ versus $23.19 \pm 0.04$ at rejection–cost weight $0.5$ (and $18.91$ versus $18.80$ for latency only). Utilization, which lies outside both families, is the exception: $6.08 \pm 0.36$ versus $6.94 \pm 0.10$.

Table 2: **Frozen preference-IQL across preference weights.** Return under the queried reward and mean provisioned CU per decision; five seeds, 266 windows, pooled logs with local exploration. Specialist: reward-specific IQL trained on the same corpus for that reward (none exists at the other points). At $c = 1$ both families reduce to the cost-only reward.

| | Service–cost | | Rejection–cost | | |
|---|---|---|---|---|---|
| Cost weight $c$ | Return | CU | Return | CU | Specialist |
| 0 | $21.16 \pm 0.05$ | 70.4 | $23.42 \pm 0.05$ | 60.6 | $23.04 \pm 0.22$ |
| 0.25 | $21.50 \pm 0.06$ | 32.2 | $23.32 \pm 0.03$ | 24.2 | – |
| 0.5 | $22.11 \pm 0.05$ | 16.3 | $23.31 \pm 0.04$ | 16.8 | $23.19 \pm 0.04$ |
| 0.75 | $22.68 \pm 0.01$ | 11.9 | $23.29 \pm 0.02$ | 11.7 | – |
| 1 | $23.70 \pm 0.02$ | 8.3 | $23.70 \pm 0.02$ | 8.3 | – |

### A.3 FB training variation and adaptations

Table 3 gives the numerical results plotted in Figure 3 in the main text.

All five FB checkpoints contribute to every objective, including seed 46, which is weak on utilization. That seed returns $3.88$ with the closed-form utilization query; the other four also remain below the utilization specialist's mean. Its final FB loss is $425$, compared with 320–376 for the other runs, and it changes allocation on $0.04$ of decisions, compared with $0.11$–$0.16$. The tested closed-form, local-search, and wide-search queries leave it between $3.60$ and $4.74$ on utilization. This associates the weak result with both training diagnostics and limited policy variation; it does not establish failure for every possible task vector. Preference-IQL's mean also includes a low utilization seed, with return $4.85$. No corresponding collapse is reported on latency or rejection for the same FB checkpoints.

**Occupancy-mass clipping and reward centering.** Both adaptations were selected on an earlier FB model with plain MLP maps, trained on the broad pool; the residual models inherit them, and we did not repeat the ablations on the residual backbone. With a row-mass cap of $100$ instead of $21$, all four plain-MLP seeds (42–45) diverged between $15{,}000$ and $25{,}000$ updates, with the FB loss reaching $1.4$–$4.4 \times 10^3$ times its value at $1{,}000$ updates; with the cap of $21$, all five seeds (42–46) completed $100{,}000$ updates without divergence. On the same frozen plain-MLP checkpoints (seeds 42–45,

| Objective | IQL specialist | pref-IQL | FB closed-form $z$ | FB + $z$-search |
|---|---|---|---|---|
| Utilization | $6.94 \pm 0.10$ | $6.08 \pm 0.36$ | $5.59 \pm 0.47$ | $5.52 \pm 0.51$ |
| Latency | $18.80 \pm 0.06$ | $18.91 \pm 0.04$ | $18.43 \pm 0.32$ | $18.56 \pm 0.34$ |
| Rejection | $23.04 \pm 0.22$ | $23.42 \pm 0.05$ | $23.00 \pm 0.26$ | $23.27 \pm 0.07$ |
| Quota (out of family) | — | $21.36 \pm 0.19$ | $22.07 \pm 0.20$ | $22.27 \pm 0.18$ |

Table 3: **Numerical results for the main comparison.** Undiscounted return, mean $\pm$ standard error across five seeds on the common evaluation suite. Each FB seed reuses one checkpoint across objectives. For quota ($Q = 20$), preference-IQL uses its best tested preference on the test suite (Section 6.2).

100,000 updates), the raw reward estimator varies mean provisioned capacity by a factor of 2.1 across the five-point service–cost preference sweep, while centering raises this factor to 18.3 (rejection–cost: 2.1 versus 21.7); in a probe of seed 42 at 50,000 updates, the median cosine similarity between raw reward embeddings is 0.74, versus 0.10 after centering. All five residual seeds train with the cap and without divergence; the cap is active on 47–58% of target rows at 100,000 updates (Table 10). These observations motivate the adaptations in the evaluated configuration; neither establishes universal necessity. Appendix D.2 gives the definitions.

**Backbone.** The main FB comparisons use two residual blocks of width 256, with pre-LayerNorm, Mish activations, and skip connections. The design follows MetaMotivo and BREEZE and uses two forward maps. The saved architecture setting is `fb_arch=residual`; Appendix J lists the full configuration.

### A.4 Capacity allocation under the quota reward

Table 4 records the operating statistics at $Q = 20$.

The selected preference-IQL query is the best evaluated weight for each seed on the test suite; this grants test-return information to the comparator but searches only the evaluated preference set. FB search provisions more capacity on average than this comparator, while exceeding the reservation on fewer decisions. Figure 4 shows one individual rollout and the aggregate allocation distribution. The illustrative rollout uses seed 43; the distribution and reported aggregate results use all five seeds.

| Method | Mean CU | Frac. CU$>20$ | Per-seed p95 CU | Mean rej. rate | Return |
|---|---|---|---|---|---|
| FB + $z$-search | 10.32 | 0.02 | 21.20 | 0.0834 | 22.27 |
| FB closed-form $z$ | 12.52 | 0.04 | 28.80 | 0.0764 | 22.07 |
| pref-IQL (best $\omega$) | 8.30 | 0.07 | 27.00 | 0.0892 | 21.36 |

Table 4: **How the quota objective is served** (pooled logs with local exploration, $Q = 20$ CU). Provisioned CU per decision, fraction of decisions above the reservation, 95th-percentile CU, mean rejection rate, and mean return, for frozen FB (closed-form and local-search queries) and for preference-IQL's best weight per seed selected on the test suite. Every column is computed per seed over its 266 windows and then averaged over five seeds; the p95 column is therefore the mean of per-seed 95th percentiles, not the 95th percentile of the pooled allocations.

**Reservation sweep.** Figure 5 shows the allowance sweep. At each $Q \in \{10, 20, 40\}$, we re-query the same frozen FB checkpoints and re-score the preference roster's $Q = 20$ rollouts under the new reward. The cost-only preference wins at every tested reservation. Its mean returns are 20.93, 21.36, and 22.01, respectively. FB's closed-form re-queries return $21.46 \pm 0.34$, $22.07 \pm 0.20$, and $22.33 \pm 0.09$; searched queries return $21.20 \pm 0.49$, $22.27 \pm 0.18$, and $22.41 \pm 0.06$. All $\pm$ values are standard errors across five training seeds.

Retaining the $Q = 20$ query and re-scoring it at $Q = 10$ gives 21.16 for closed-form inference and 20.33 for search, compared with 21.46 and 21.20 after re-querying. At $Q = 40$, the unchanged queries return 22.11 and 22.28, compared with 22.33 and 22.41 after re-querying.

At $Q = 10$, exceedance is $0.12 \pm 0.04$ for closed-form re-querying, $0.19 \pm 0.09$ for searched re-querying, and $0.14 \pm 0.01$ for the selected preference policy. At $Q = 20$, the corresponding values are $0.04 \pm 0.02$, $0.02 \pm 0.02$, and $0.07 \pm 0.01$; at $Q = 40$, they are $0.01 \pm 0.01$, $0.01 \pm 0.01$, and $0.03 \pm 0.01$. For the unchanged $Q = 20$ queries evaluated at $Q = 10$, exceedance is $0.23$ for closed-form inference and $0.48$ for search. One searched re-query has exceedance $0.54$ at $Q = 10$, so the means do not imply reliable satisfaction of a hard limit. At $Q = 10$, the mean of per-seed median allocations is $6.0$ CU for closed-form FB and $7.2$ for searched FB, compared with $3.9$ for preference-IQL.

### A.5 Inference-time selection diagnostics

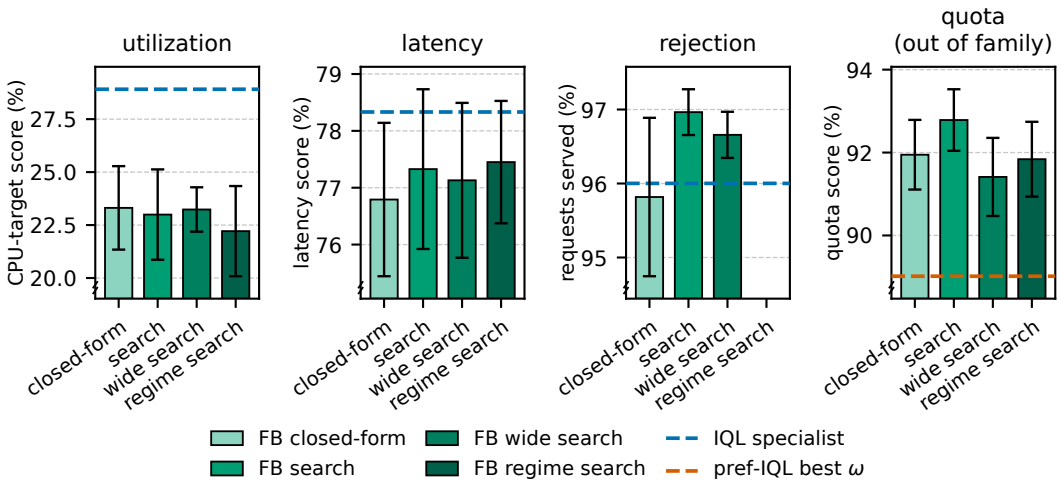



Figure 6: **Local search helps some objectives; broader search offers no consistent gain.** Score is $100\times$return$/24$; means and standard errors across five seeds. Regime search is the demand-gated query described below (not run on rejection). Dashed lines show reward-specific IQL for the first three objectives and the test-selected preference baseline for quota ($Q = 20$).

Figure 6 reports closed-form, local-search, wide-search, and demand-gated (regime) results. Wide-search means are $5.58$, $18.51$, $23.20$, and $21.94$ for utilization, latency, rejection, and quota, respectively. Thus wide search slightly exceeds local search on utilization ($5.52$) but remains below closed-form inference ($5.59$); on the other three objectives its means are lower than local search. This comparison does not by itself establish critic over-optimism. Such a diagnosis requires candidate-level predicted and realized rankings, with the scoring distribution, discount, and evaluation horizon accounted for.

**Demand-dependent task selection.** We also infer one task vector per demand regime and select among them using the observed request rate at each step. Four bins use edges $1.0$, $2.0$, and $9.9$ requests/s from support-set quantiles. Across the five utilization checkpoints, low- and high-demand task vectors have cosine similarity $-0.03$–$0.16$, while neighbouring vectors within each half have similarities $0.67$–$0.87$. This changes the execution rule from a single task vector fixed for the rollout to a demand-gated policy.

Demand gating gives utilization return $5.33 \pm 0.51$, compared with $5.59 \pm 0.47$ for the closed-form query. One seed improves by $1.3$, three fall by $0.6$–$1.1$, and the weak seed returns $3.72$. These evaluated task partitions do not consistently recover the utilization gap. On latency and quota, demand gating returns $18.59 \pm 0.26$ and $22.04 \pm 0.22$, within $0.25$ of closed-form inference ($18.43 \pm 0.32$, $22.07 \pm 0.20$) and local search ($18.56 \pm 0.34$, $22.27 \pm 0.18$); it was not evaluated on rejection.

### A.6 Changing service-level targets

Table 5 reports three target changes that preference-IQL's reward basis cannot express: a tightened latency target ($0.5$ s target and $0.9$ s cap instead of $0.8$ and $1.2$ s), a utilization setpoint moved from

$70\%$ to $85\%$ CPU, and an availability cliff that adds a penalty of $-2$ to the main reward whenever rejection reaches $3\%$. Frozen FB is queried with each new reward on all five seeds. Preference-IQL receives the weight vector of the corresponding in-family reward, since the changed thresholds and the cliff are not among its inputs. Its means are higher than FB's on all three targets, although FB varies strongly across seeds on the availability cliff. For tightened latency, taking per seed the best of this weight vector and two alternatives, $(\omega_{\text{rej}}, \omega_{\text{lat}}) = (0.5, 0.5)$ and $(\omega_{\text{lat}}, \omega_{\text{cost}}) = (0.75, 0.25)$, gives $17.88 \pm 0.06$ over five seeds. These are supplementary results and do not support a main-text finding.

| Objective | pref-IQL (reward-matched $\omega$) | FB closed-form $z$ | FB + $z$-search |
|---|---|---|---|
| Tightened latency SLO (0.5 s) | $17.89 \pm 0.05$ $(n = 4)$ | $17.16 \pm 0.28$ | $17.39 \pm 0.22$ |
| Utilization setpoint 0.85 | $5.40 \pm 0.32$ | $4.76 \pm 0.42$ | $4.79 \pm 0.37$ |
| Availability cliff | $15.92 \pm 0.76$ | $14.67 \pm 1.55$ | $15.41 \pm 1.03$ |

Table 5: **Additional target changes.** Returns under each changed objective; $\pm$ is the standard error across five training seeds on the 266-window suite, except the preference-IQL tightened-latency cell.

# B Reward-specific reuse across objectives

# C Detailed problem and reward definitions

We study how fixed logs generated by an existing autoscaling controller can be reused for new control objectives. The transition data remain fixed throughout; what changes is when the desired objective becomes available to the learner. We consider rewards specified before training, a family of rewards specified during training but selected at deployment, and reward-free pretraining in which the task itself is supplied only after training.

## C.1 States, actions, and returns

**States.** The controller acts every five minutes. At decision epoch $t$, it observes

$$s_t = \big[\rho_t,\ \ell_t,\ u_t,\ \mathrm{rss}_t,\ n_t,\ q_t,\ b_t,\ \nu_t\big], \tag{22}$$

where $\rho_t \in [0, 1]$ is the fraction of requests rejected and $\ell_t \in \mathbb{R}_{\geq 0}$ the mean request latency in seconds, both measured over the previous control interval; $u_t \in [0, 1]$ is cluster CPU utilization and $\mathrm{rss}_t \in [0, 1]$ memory utilization; $n_t \in \{1, \ldots, 42\}$ and $q_t \in \{0.5, 1, 2, 4, 8, 16\}$ are the current number of workers and capacity units per worker; $b_t \in \{0, 1\}$ indicates steady state; and $\nu_t \in \mathbb{Z}_{\geq 0}$ is the number of requests observed in the interval.

**Actions.** The controller proposes an allocation $a_t^{\text{ctrl}} \in \mathcal{A}$. The allocation actually applied is

$$a_t = (n_{t+1}, q_{t+1}) \in \mathcal{A}, \tag{23}$$

with total provisioned capacity $C(a_t) = n_{t+1} q_{t+1}$. Without exploration, $a_t = a_t^{\text{ctrl}}$. During collection with local exploration, a wrapper may perturb the proposal, giving $a_t \neq a_t^{\text{ctrl}}$. The logged transition records $(s_t, a_t, s_{t+1})$. Thus each unperturbed proposal yields one applied allocation, while exploration can yield several applied allocations for the same proposal.

**Returns.** Training trajectories are finite windows sampled from a continuing control process. For learning, we use discounted values with discount $\gamma \in (0, 1)$,

$$J_r^{\gamma}(\pi) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1})\right]. \tag{24}$$

Evaluation reports the undiscounted sum of rewards over the fixed evaluation horizon.

Table 6: **Where improvement is available.** Learned policies are trained on pooled logs with local exploration and specialized to each objective; rule controllers are the same four throughout. The last row is the IQL − SCALEX difference of means, with the standard error of the IQL mean.

| | utilization | latency | rejection |
|---|---|---|---|
| IQL on $\mathcal{D}_2$ | **6.94** $\pm$ 0.10 | 18.80 $\pm$ 0.06 | 23.04 $\pm$ 0.22 |
| BC on $\mathcal{D}_2$ | 4.01 $\pm$ 0.30 | 17.90 $\pm$ 0.11 | 22.16 $\pm$ 0.07 |
| cpu-SCALEX@0.70 (utilization reference) | 6.71 | 15.96 | 22.57 |
| step scaling | 5.40 | 16.94 | 21.84 |
| target tracking | 5.39 | 15.92 | 21.68 |
| SCALEX (logging controller) | 4.00 | **19.28** | **23.51** |
| IQL − SCALEX | $+2.94 \pm 0.10$ | $-0.48 \pm 0.06$ | $-0.47 \pm 0.22$ |

$\pm$ = standard error over five training seeds (each seed's value is its mean over the 266 windows). Rule controllers are deterministic: one run, no $\pm$. Last row: difference of means, $\pm$ = SE of the IQL mean (SCALEX is fixed).

Table 7: **Interior reward trade-offs.** Reward-specific IQL policies are trained from the existing transitions after relabeling for utilization+latency or utilization+rejection. The table reports paired comparisons against the fixed rule controllers on the common held-out evaluation windows.

| | util+lat | util+rej |
|---|---|---|
| offline RL (best corpus) | **12.19** | **14.89** |
| SCALEX (logging controller) | 11.64 | 13.75 |
| cpu-SCALEX@0.70 (analytic reference) | 11.33 | 14.64 |
| target tracking | 10.65 | 13.53 |
| step scaling | 11.17 | 13.62 |
| $\Delta$ vs. SCALEX, paired | $+0.55 \pm 0.20^*$ | $+1.14 \pm 0.31^*$ |
| $\Delta$ vs. cpu-SCALEX@0.70, paired | $+0.86 \pm 0.49^*$ | $+0.26 \pm 0.27^*$ |
| $\Delta$ vs. target tracking, paired | $+1.54 \pm 0.41^*$ | $+1.36 \pm 0.39^*$ |
| $\Delta$ vs. step scaling, paired | $+1.02 \pm 0.30^*$ | $+1.28 \pm 0.39^*$ |

**Finite-window visitation.** The normalized discounted visitation used in Section 4.1 is

$$d_T^\pi = \frac{\sum_{t=0}^{T-1} \gamma^t \mathcal{L}_\pi(s_t, a_t)}{\sum_{t=0}^{T-1} \gamma^t},$$

where $\mathcal{L}_\pi(s_t, a_t)$ is the joint distribution at decision $t$. With a shared workload process, initialization, and window length, its workload marginal is independent of the policy.

**Offline training.** A behavior controller $\pi_\beta$ generates a fixed transition dataset $\mathcal{D} = \{(s_i, a_i, s_i')\}_{i=1}^N$. The offline learner then optimizes a policy using these transitions but cannot collect additional interaction during training. The controllers and excitation mechanisms used to construct these datasets are described in Appendix H.1, and the information the resulting logs contain is studied in Section 4.3 and Appendix G.1.

### C.2 ONE LOG, MANY REWARDS

Autoscaling is inherently multi-objective. Different operators may prefer lower request rejection, lower latency, higher resource utilization, or lower provisioned capacity. We represent these objectives using five penalties computed from each logged transition,

$$\mathbf{p}(s_t, a_t, s_{t+1}) = [p_{\text{rej}}, p_{\text{lat}}, p_{\text{cost}}, p_{\text{under}}, p_{\text{over}}]^\top. \tag{25}$$

Because $a_t$ determines the allocation during the following control interval, service and utilization penalties are computed from the resulting measurements in $s_{t+1}$. We define

$$p_{\text{rej}} = \rho_{t+1}, \qquad p_{\text{cost}} = \frac{C(a_t)}{C_{\max}}, \qquad p_{\text{lat}} = \text{clip}\left(\frac{\ell_{t+1} - 0.8}{1.2 - 0.8}, 0, 1\right),$$
$$p_{\text{under}} = \frac{\left[u^\star - u_{t+1}\right]_+}{u^\star}, \qquad p_{\text{over}} = \frac{\left[u_{t+1} - u^\star\right]_+}{1 - u^\star}.$$

where $[x]_+ = \max(x, 0)$, $C(a_t) = n_{t+1}q_{t+1}$, and $C_{\max} = 42 \times 16 = 672$. Thus, each penalty lies in $[0, 1]$. The latency penalty is zero below $0.8$ seconds and increases linearly to one at $1.2$ seconds. The utilization penalties measure normalized deviation below and above the target utilization $u^\star = 0.70$.

For a non-negative preference vector $\omega$, we define

$$r_\omega(s_t, a_t, s_{t+1}) = 1 - \omega^\top \mathbf{p}(s_t, a_t, s_{t+1}). \tag{26}$$

Because every component of $\mathbf{p}$ is available from the logged transition, the same dataset can be assigned different rewards after collection without collecting new trajectories. Although we write the reward as $r_\omega(s_t, a_t, s_{t+1})$, every penalty above can be evaluated from the resulting state $s_{t+1}$. In particular, $s_{t+1}$ contains the applied allocation $(n_{t+1}, q_{t+1})$, so its provisioned capacity is $C(a_t) = n_{t+1}q_{t+1}$. We can therefore treat these objectives as state rewards $r(s)$ when inferring tasks for the reward-free model in Section 5.1.

Our experiments use three primary objectives: *rejection*, *latency*, and *utilization*. We define the corresponding rewards as

$$r_{\text{rej}} = 1 - p_{\text{rej}} \quad r_{\text{lat}} = 1 - p_{\text{lat}} \quad r_{\text{util}} = 1 - (p_{\text{under}} + p_{\text{over}}).$$

Since $p_{\text{under}}$ and $p_{\text{over}}$ cannot be nonzero simultaneously, $r_{\text{util}} \in [0, 1]$. This objective rewards operation near $70\%$ CPU utilization, and does not directly reward lower provisioned capacity. Any capacity reduction observed under $r_{\text{util}}$ is therefore a consequence of improved utilization rather than a directly counted cost saving.

We additionally consider mixed service-resource objectives by assigning nonzero weights to multiple components of $\mathbf{p}$. This linear family gives preference-conditioned policies a fixed task interface: changing $\omega$ changes the trade-off, but every objective must still be expressible through the penalties chosen before training. It does not cover all objectives that an operator might specify later. For example, a reserved-capacity tariff that is free below a threshold and costly above it cannot, in general, be represented by any $\omega$. This distinction lets us separate two forms of reuse in our experiments: selecting a new preference within a task family known during training, and supplying the learner with a genuinely new task after training.

### C.3 Learning objectives

**Reward specified before training.** For a fixed reward $r_\omega$, we seek a policy

$$\pi^\star_\omega \in \arg\max_\pi J^\gamma_{r_\omega}(\pi). \tag{27}$$

The policy is trained using only the fixed dataset relabeled with $r_\omega$. This asks whether the controller logs contain enough information to improve on the logging controller for one specified objective.

**Reward family specified before training.** We next allow the requested objective to change after training, provided it belongs to a family known beforehand. We seek a single conditional policy $\pi_\theta(a \mid s, \omega)$ with objective

$$\theta^\star \in \arg\max_\theta \mathbb{E}_{\omega \sim p(\Omega)} \left[J^\gamma_{r_\omega}\left(\pi_\theta(\cdot \mid \cdot, \omega)\right)\right]. \tag{28}$$

The same fixed transitions are relabeled for different $\omega$ during training. At deployment, changing $\omega$ changes the requested trade-off without collecting new data or updating the policy.

**Task specified after training.** The previous setting still fixes the space of possible objectives before training. We therefore also consider reward-free pretraining: learn a reusable representation from $\mathcal{D}$ without evaluating any downstream reward, and supply the task only after training. Let $\rho_{\mathcal{D}}$ denote the state distribution in the logged data and $m^{\pi_z}(s, a, s^+)$ the discounted successor-state density of policy $\pi_z$ relative to $\rho_{\mathcal{D}}$. We seek representations $F$ and $B$ satisfying

$$m^{\pi_z}(s, a, s^+) \approx F(s, a, z)^\top B(s^+), \qquad \pi_z(s) \in \arg\max_{a \in \mathcal{A}} \pi_\phi(a \mid s, z).. \tag{29}$$

After training, a reward, goal, or desired state distribution is mapped to $z$ and thereby selects a policy. Following Pirotta et al. (2024); Tirinzoni et al. (2025), we refer to this setting as a *behavior foundation model*: one behavioral model is pretrained once and reused across downstream control tasks without task-specific optimization.

# D Learning objectives and implementation details

All learners use fixed transitions and the allocations actually executed. The IQL references use reward labels; behavior cloning and FB do not use downstream rewards in their training objectives. This appendix first specifies the reference learners, then gives the architecture, losses, and inference procedure of the residual FB model used in the main experiments. Settings for the reference learners appear in Appendix J; Table 8 specifies the residual FB recipe.

## D.1 Reward-aware reference policies

**Reward-specific IQL.** For a specified reward $r_\omega$, we relabel the controller logs to obtain

$$\mathcal{D}^\omega = \{(s_i, a_i, r_\omega(s_i, a_i, s_i'), s_i')\}_{i=1}^N. \tag{30}$$

We use Implicit Q-Learning (IQL) (Kostrikov et al., 2022) to learn from these transitions. Its Bellman target uses a state-value estimate rather than a maximization over next-state actions, reducing reliance on critic estimates for allocations poorly represented in the logs.

The learner contains two critics $Q_{\theta_1}, Q_{\theta_2}$, a value function $V_\psi$, and a categorical actor $\pi_\phi$. For a minibatch $\mathcal{B}$, write

$$q(s, a) = \min_{k \in \{1,2\}} Q_{\bar{\theta}_k}(s, a), \tag{31}$$

using the target critics $Q_{\bar{\theta}_k}$. The value function is fitted by expectile regression:

$$\mathcal{L}_V(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}^\omega}\left[L_2^\tau\big(q(s, a) - V_\psi(s)\big)\right], \tag{32}$$

$$L_2^\tau(v) = |\tau - \mathbf{1}\{v < 0\}|\, v^2. \tag{33}$$

The critic values are treated as fixed targets in this update. For $\tau > 0.5$, the loss emphasizes the higher values among actions represented in the dataset. The critics minimize

$$y_\omega = r_\omega(s, a, s') + \gamma V_{\bar{\psi}}(s'), \tag{34}$$

$$\mathcal{L}_Q(\theta_1, \theta_2) = \mathbb{E}_{\mathcal{D}^\omega}\left[\sum_{k=1}^{2}\big(Q_{\theta_k}(s, a) - y_\omega\big)^2\right]. \tag{35}$$

Here $\bar{\psi}$ denotes target parameters, and the Bellman target is detached from the critic update. Training windows are truncations of a continuing process, rather than task terminations; the bootstrap therefore remains present at a window boundary.

The actor is trained by advantage-weighted regression (Peng et al., 2019). We compute

$$A(s, a) = q(s, a) - V_\psi(s), \qquad \widetilde{A}(s, a) = \frac{A(s, a) - \mu_{A,\mathcal{B}}}{\sigma_{A,\mathcal{B}} + \epsilon}, \tag{36}$$

where $\mu_{A,\mathcal{B}}$ and $\sigma_{A,\mathcal{B}}$ are the minibatch mean and standard deviation. The actor loss is

$$w(s, a) = \min\{100, \exp(3\widetilde{A}(s, a))\}, \tag{37}$$

$$\mathcal{L}_\pi^{\mathrm{IQL}}(\phi) = -\mathbb{E}_{(s,a)\sim\mathcal{D}^\omega}\left[w(s, a) \log \pi_\phi(a \mid s)\right]. \tag{38}$$

The weights are detached during the actor update. This objective favors logged actions with higher estimated advantage while retaining a supervised-learning policy update.

**Behavior cloning.** Behavior cloning (BC) uses the same state–action pairs without reward labels or successor states. We train a categorical policy with focal loss and entropy regularization:

$$\begin{aligned}\mathcal{L}_{\mathrm{BC}}(\phi) = & -\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[(1-\pi_\phi(a\mid s))^2 \log \pi_\phi(a\mid s)\right] \\ & -10^{-3}\mathbb{E}_{s\sim\mathcal{D}}\left[H\big(\pi_\phi(\cdot\mid s)\big)\right].\end{aligned} \tag{39}$$

The same trained BC policy is evaluated under every objective. Its comparison with IQL tests whether reward-aware learning improves on imitation of the available controller behavior.

**Preference-conditioned IQL.** A specialist is tied to the reward used during training. Preference-conditioned IQL instead conditions every learned component on the requested preference:

$$Q_{\theta_k}(s,a,\omega), \qquad V_\psi(s,\omega), \qquad \pi_\phi(a\mid s,\omega). \tag{40}$$

For the reward family defined in Appendix C.2, we sample $\omega \sim p_\Omega$ and relabel transitions with $r_\omega = 1-\omega^\top \mathbf{p}$. The corresponding target and advantage are

$$y_\omega = r_\omega(s,a,s') + \gamma V_{\bar{\psi}}(s',\omega), \tag{41}$$

$$A_\omega(s,a) = \min_{k\in\{1,2\}} Q_{\theta_k}(s,a,\omega) - V_\psi(s,\omega). \tag{42}$$

The losses otherwise follow Equations Equation (32)–Equation (38), with the same sampled preference used throughout each transition's update. At deployment, changing $\omega$ selects an operating trade-off without updating the model. The reward basis, however, is fixed before training.

**Implementation distinction.** The preference implementation documented in Appendix J exposes eight penalty coordinates: five service–resource coordinates and three additional penalties for switching, upward resizing, and downward resizing. Its utilization thresholds match the evaluated objective ($u_{\mathrm{target}} = u_{\max} = 0.70$ in the saved configurations of all five seeds). Thus, this training configuration uses a superset of the five-component reward family stated in the problem setup. The service–cost sweeps set the resizing weights to zero, so the extra coordinates do not alter the rewards requested in those sweeps, although they do change the objectives encountered during training. The full training basis and preference-sampling distribution must therefore accompany comparisons with reward-specific policies.

### D.2 Successor occupancy and reward readout

FB represents the future joint state, including workload and system measurements. The factored dynamics motivate data collection; they do not impose a factorization on the network architecture. For a measurable state set $E$, the unnormalized discounted successor measure is

$$M^{\pi_z}(s,a,E) = \mathbb{E}_{\pi_z}\left[\sum_{t=0}^{\infty}\gamma^t \mathbf{1}\{s_{t+1}\in E\}\,\middle|\, s_0=s, a_0=a\right]. \tag{43}$$

Where this measure admits a density relative to the reference distribution $\rho_{\mathcal{D}}$, FB approximates

$$M^{\pi_z}(s,a,\mathrm{d}s^+) \approx F_\theta(s,a,z)^\top B_\psi(s^+)\rho_{\mathcal{D}}(\mathrm{d}s^+). \tag{44}$$

For a state reward $r$, integrating this prediction gives

$$\xi_r = \mathbb{E}_{s\sim\rho_{\mathcal{D}}}[B_\psi(s)r(s)], \tag{45}$$

$$Q_r^{\pi_z}(s,a) \approx F_\theta(s,a,z)^\top \xi_r. \tag{46}$$

The raw embedding $\xi_r$ retains the reward's scale and offset; the centered, normalized query $z_r$ selects a policy. All rewards considered through this interface must be evaluable from the represented state.

The ideal value-greedy extraction rule is

$$\pi_z^Q(s) \in \arg\max_{a\in\mathcal{A}} F_\theta(s,a,z)^\top z. \tag{47}$$

Our primary implementation instead learns a categorical actor and uses

$$\pi_z(s) = \arg\max_{a\in\mathcal{A}} \pi_\phi(a\mid s,z). \tag{48}$$

This actor supplies continuation actions in the bootstrap, actions scored during search, and deployed allocations. We do not assume that the actor and the value-greedy policy coincide.

### D.3 RESIDUAL FB ARCHITECTURE

**Observation and action encoding.** The network input is

$$\Big[\rho, \ell, u, \mathrm{rss}, n/42, q/16, b, \mathrm{clip}\Big(\frac{\nu}{300 \cdot 100}, 0, 1\Big)\Big]. \tag{49}$$

Latency remains in seconds; no additional input standardization is applied. The joint allocation $(n, q_j)$ has index $6(n-1)+j$, where $j \in \{0, \dots, 5\}$ indexes the ordered capacity tiers. The forward map receives its 252-dimensional one-hot encoding. Tasks have dimension $d = 100$ and projection radius $\sqrt{d} = 10$.

**Forward and backward maps.** Each map uses an input projection, two residual blocks of width 256, an output LayerNorm, and a linear readout:

$$\begin{aligned} h_0 &= W_{\mathrm{in}} x + b_{\mathrm{in}}, \\ h_\ell &= h_{\ell-1} + W_\ell^{(2)} \,\mathrm{Mish}\Big(W_\ell^{(1)} \,\mathrm{LN}(h_{\ell-1}) + b_\ell^{(1)}\Big) + b_\ell^{(2)}, \quad \ell = 1, 2, \\ y &= W_{\mathrm{out}} \,\mathrm{LN}(h_2) + b_{\mathrm{out}}. \end{aligned} \tag{50}$$

The two forward heads $F_\theta^{(1)}, F_\theta^{(2)}$ have independent parameters. Each takes the concatenated 360-dimensional state, task, and action encoding and outputs 100 features. Single-prediction readouts use their mean, denoted $F_\theta$ in the main text. This includes bootstrap targets, actor advantages, and search scores; no disagreement penalty is applied. The backward map takes the eight state features and normalizes its output to $\|B_\psi(s)\|_2 = 10$. Target copies of both maps are updated by Polyak averaging.

**Actor.** The categorical actor maps $[s, z] \in \mathbb{R}^{108}$ through three ReLU layers of width 256 to 252 logits, with Xavier-uniform weight initialization. The actor uses an MLP even though the forward and backward maps use residual blocks.

### D.4 RESIDUAL FB TRAINING

**Replay and task sampling.** The pooled-plus-local-exploration corpus contains 401,631 transitions and fits entirely in the replay buffer of capacity $10^6$. Every update samples $N = 1{,}024$ transitions uniformly. Rewards computed by the data-loading pipeline are diagnostic only: neither the representation update nor the actor update receives them. The auxiliary $Q$-regression loss is disabled.

Each minibatch row initially receives a Gaussian task projected onto the task sphere. With probability 0.71 per row, it is replaced by $B_\psi(s'_{\sigma(i)})$, where $\sigma$ is a random permutation of the minibatch. These goal embeddings retain their gradient path through the online backward map during the representation update. With probability 0.5 per update, a further 30% of rows are replaced by tasks sampled from a FIFO buffer of the 10,000 most recent training task vectors. This buffer contains past tasks, not additional environment rollouts. The actor receives the same mixed tasks with their gradients detached. This training mixture does not establish that arbitrary downstream reward queries have the same distribution.

**Core occupancy objective.** For a single forward map, write $m_z(s, a, s^+) = F_\theta(s, a, z)^\top B_\psi(s^+)$. The standard FB objective is

$$\mathcal{L}_{\mathrm{FB}}^{\mathrm{core}} = \mathbb{E}\Big[\big(m_z(s, a, s^+) - \gamma \bar{m}_z(s', \pi_z(s'), s^+)\big)^2 - 2 m_z(s, a, s')\Big], \tag{51}$$

where $s^+$ is an independent reference state and target-network predictions are detached. The squared term propagates future occupancy; the linear term anchors it to the observed successor. The implementation below uses two heads, an off-diagonal minibatch estimator, and capped targets.

**Bootstrap targets and the mass cap.** For minibatch rows $(s_i, a_i, s'_i)$ and tasks $z_i$, define

$$\begin{aligned} M_{ij}^{(k)} &= F_\theta^{(k)}(s_i, a_i, z_i)^\top B_\psi(s'_j), \\ \bar{M}_{ij} &= \frac{1}{2} \sum_{k=1}^{2} \bar{F}^{(k)}(s'_i, \pi_{z_i}(s'_i), z_i)^\top \bar{B}(s'_j). \end{aligned} \tag{52}$$

Columns are the next states from the same minibatch; distinct rows supply reference states for the off-diagonal regression. We cap the mean positive mass of each target row before applying its discount:

$$m_i^+ = \frac{1}{N}\sum_j [\bar{M}_{ij}]_+, \qquad \widetilde{M}_{ij} = \bar{M}_{ij}\frac{21}{\max(21, m_i^+)}. \tag{53}$$

The cap rescales all entries in a row together; entrywise clipping is disabled. Its threshold approximates $\sum_{t=0}^{23} 0.99^t \simeq 21.4$. It is a heuristic rather than a bound on continuing-task occupancy, whose total discounted mass is 100.

With terminal indicator $d_i$, define

$$\Delta_{ij}^{(k)} = M_{ij}^{(k)} - \gamma(1-d_i)\widetilde{M}_{ij}, \qquad \overline{\Delta}_{ii} = \frac{1}{2}\sum_k \Delta_{ii}^{(k)}. \tag{54}$$

An artificial window boundary is a truncation of the continuing task, not a terminal state. Accordingly, the data loader stores $d_i = 0$ for every transition, including the final transition of each window, so targets bootstrap through window ends. The implemented representation loss is

$$\mathcal{L}_{\mathrm{FB}} = \frac{1}{2N(N-1)}\sum_{k=1}^{2}\sum_{i\neq j}\big(\Delta_{ij}^{(k)}\big)^2 - \frac{2}{N}\sum_i \overline{\Delta}_{ii}. \tag{55}$$

The linear term uses the diagonal residual, including the bootstrap, rather than only the online prediction. Because the entire bootstrap target is detached, this addition has zero gradient during the update. It changes the reported scalar loss but not the gradient of that linear term.

**Backward-feature regularization.** Let $\Sigma_{ij} = B_\psi(s_i')^\top B_\psi(s_j')$. We add

$$\mathcal{L}_{\mathrm{orth}} = \frac{1}{2N(N-1)}\sum_{i\neq j}\Sigma_{ij}^2 - \frac{1}{N}\sum_i \Sigma_{ii} \tag{56}$$

with coefficient 1, giving total representation loss $\mathcal{L}_{\mathrm{FB}} + \mathcal{L}_{\mathrm{orth}}$. Since backward outputs have squared norm $d = 100$, the diagonal term equals $-100$ and is constant; only the off-diagonal term contributes a gradient. Representation gradients update the forward and backward maps.

**Actor extraction.** The actor uses advantage-weighted regression from the first update. For every logged state, we enumerate all actions and compute

$$\begin{aligned} Q_z(s,a) &= \frac{1}{2}\sum_k F_\theta^{(k)}(s,a,z)^\top z, \\ V_z(s) &= \sum_a \pi_\phi(a \mid s,z) Q_z(s,a), \\ A_i &= Q_{z_i}(s_i,a_i) - V_{z_i}(s_i). \end{aligned} \tag{57}$$

The actor probabilities in the baseline are detached. Advantages are divided by their minibatch standard deviation, then converted to self-normalized exponential weights at temperature $\beta = 1$. A leave-one-out correction yields

$$u_i = \frac{\exp(\widetilde{A}_i/\beta)}{\sum_j \exp(\widetilde{A}_j/\beta)}, \qquad w_i = \frac{u_i/(1-u_i)}{\sum_j u_j/(1-u_j)},$$
$$\mathcal{L}_{\mathrm{actor}} = -\sum_i w_i \log \pi_\phi(a_i \mid s_i, z_i). \tag{58}$$

All advantages and weights are detached during the actor step. This weighting differs from the clipped exponential weights used by IQL. The saved actor-pessimism setting is inactive in AWR mode; actor advantages use the mean of the forward heads.

**Optimization.** Separate Adam optimizers update the forward maps, backward map, and actor, each at learning rate $10^{-4}$ with zero weight decay. An update consists of the representation step, actor step, and target update, in that order. Gradient norms are clipped at 10 separately for each optimizer's parameters. The discount is 0.99; target networks use $\bar{\theta} \leftarrow 0.99\bar{\theta} + 0.01\theta$ after each update, and likewise for the backward map. All five runs use the same settings, seeds 42–46, and 100,000 updates. Checkpoints are saved every 10,000 updates; main comparisons use each seed's final checkpoint, without choosing a seed by test performance.

| Setting | Value |
|---|---|
| State / task / action dimensions | 8 / 100 / 252 |
| Forward and backward backbones | Two residual blocks, width 256, LayerNorm, Mish |
| Forward heads / aggregation | Two / mean; no pessimism penalty |
| Backward output / task radius | L2 normalized / 10 |
| Actor | Three ReLU hidden layers, width 256; 252 logits |
| Mass cap / entrywise cap | 21 / disabled |
| Orthogonality / auxiliary $Q$ coefficients | 1 / 0 |
| Goal-task probability / gradient | 0.71 / attached through online $B$ |
| Task-history mixture / buffer | 0.5 per update, 30% of rows / 10,000 |
| Actor / temperature / weighting | AWR / 1 / standardized advantages, leave-one-out |
| Optimizers / learning rates | Three Adam optimizers / all $10^{-4}$ |
| Weight decay / gradient-norm cap | 0 / 10 |
| Batch size / update count | 1,024 / 100,000 |
| Discount / target-update coefficient | 0.99 / 0.01 |
| Replay capacity / sampling | $10^6$ / uniform over 401,631 rows |
| Checkpoint cadence / training seeds | 10,000 updates / 42–46 |

Table 8: Residual FB architecture and training settings. Inference settings are reported in Table 9.

### D.5 Reward queries and inference-time search

**Matched support and reward evaluation.** The representation and actor remain frozen throughout inference. The primary support set contains 2,000 states sampled without replacement from the model's training corpus using seed 42. We undo allocation scaling before evaluating a supplied reward. The empirical raw reward embedding is

$$\widehat{\xi}_r = \frac{1}{|\mathcal{S}_{\mathrm{ref}}|} \sum_{s \in \mathcal{S}_{\mathrm{ref}}} B_\psi(s) r(s). \tag{59}$$

The reference distribution is part of the FB density representation; substituting an arbitrary corpus does not preserve this readout without accounting for the distribution change.

**Centering and projection.** Write $\widehat{\mu}_B = |\mathcal{S}_{\mathrm{ref}}|^{-1} \sum_s B_\psi(s)$ and let $\bar{r}_{\mathrm{ref}}$ be the mean support reward. The centered query is

$$\widehat{\xi}_r^{\mathrm{ctr}} = \widehat{\xi}_r - \bar{r}_{\mathrm{ref}} \widehat{\mu}_B, \qquad z_r = \mathrm{proj}_z(\widehat{\xi}_r^{\mathrm{ctr}}). \tag{60}$$

For nonzero $v$, $\mathrm{proj}_z(v) = \sqrt{d}\, v / \|v\|_2$. The implementation uses a small norm floor where a zero centered embedding, including for constant support rewards, maps to the zero task vector. Using the same support, $\widehat{\xi}_{r+b} = \widehat{\xi}_r + b\widehat{\mu}_B$, so centering makes the query invariant to additive reward offsets. This is consistent with the continuing-task invariance $Q^\pi_{r+b} = Q^\pi_r + b/(1-\gamma)$. It removes the common offset from task selection without changing pretrained parameters. The evaluation launchers explicitly select the centered estimator, overriding the checkpoint's default raw estimator.

**Local and wide candidate sets.** For the primary local search, the candidate set is

$$\mathcal{Z}_r^{\mathrm{local}} = \{z_r\} \cup \{\mathrm{proj}_z(z_r + 0.5\epsilon_m)\}_{m=1}^{256}, \qquad \epsilon_m \sim \mathcal{N}(0, I_d). \tag{61}$$

It contains the closed-form query and 256 perturbations, with no independent sphere proposals. The wide variant uses 256 local perturbations at standard deviation 1.0 and 256 additional uniform sphere proposals, together with the closed-form candidate. It changes both the local perturbation scale and the number of global proposals; it is not a controlled ablation of candidate count alone.

**Scoring and selection.** We subsample 512 scoring states from the support set using seed 0. For each candidate, the greedy categorical actor supplies an action, the two forward heads are averaged, and their occupancy prediction is read out against the same fixed $z_r$:

$$\widehat{S}_r(z) = \frac{1}{|\mathcal{S}_{\text{score}}|} \sum_{s \in \mathcal{S}_{\text{score}}} \left[ \frac{1}{2} \sum_k F_\theta^{(k)}(s, \pi_z(s), z) \right]^\top z_r, \tag{62}$$

$$z_r^\star \in \arg\max_{z \in \mathcal{Z}_r} \widehat{S}_r(z). \tag{63}$$

The chosen vector is fixed throughout the rollout. This is selection of one policy for the requested reward, not statewise generalized policy improvement across all candidate critics. Candidates are ranked without simulator rollouts or gradient updates. Including $z_r$ ensures that the selected model score is at least the closed-form score, but does not ensure a higher realized return.

**What the score measures.** For a nonzero centered embedding, write $k_r = \sqrt{d}/\|\widehat{\xi}_r^{\text{ctr}}\|_2 > 0$. Then $z_r = k_r \widehat{\xi}_r^{\text{ctr}}$. With exact occupancy and reward embeddings, the score would rank the candidates by their discounted values for the same centered reward over the scoring-state distribution. The scale and constant shift would not change this ordering. Approximation error can change the ordering, and the scoring states are logged states rather than the evaluation initial-state distribution. We therefore report held-out, finite-horizon, undiscounted outcomes separately from model scores. Neither policy search nor the quota reward imposes a hard resource budget.

| Setting | Value |
|---|---|
| Reward estimator | Centered, then projected to radius 10 |
| Primary support | 2,000 training-corpus states, no replacement, seed 42 |
| Local search | Closed-form query plus 256 perturbations, standard deviation 0.5 |
| Wide search | Closed-form query plus 256 perturbations (standard deviation 1.0) and 256 sphere draws |
| Search scoring set | 512 support states, seed 0 |
| Actions / forward aggregation | Greedy categorical actor / head mean |
| Task during primary rollout | Fixed after selection |
| Regime-query diagnostic | Four quantile bins, 8,000 support states, no smoothing |
| Action-extraction diagnostics | 32-neighbor support mask; value-greedy head minimum or mean |

Table 9: Inference settings. Local search is the primary searched-policy result; wide search and the final two rows are diagnostic variants.

### D.6 Additional inference diagnostics

**Reward and state proposals.** Earlier extraction diagnostics also considered a roster of reward queries, state-based anchors, and interpolation between anchors. These proposals are separate from the local and wide searches reported for the residual checkpoints. For a desired state set $\mathcal{G}$ and reference set $\mathcal{R}$, define

$$z_{\text{state}} = \text{proj}_z(\overline{B}_{\mathcal{G}} - \overline{B}_{\mathcal{R}}), \qquad \overline{B}_{\mathcal{X}} = \frac{1}{|\mathcal{X}|} \sum_{s \in \mathcal{X}} B_\psi(s), \tag{64}$$

and interpolated tasks

$$z(\alpha) = \text{proj}_z\big(\alpha z_r + (1-\alpha) z_{\text{state}}\big), \qquad \alpha \in [0,1]. \tag{65}$$

High-CPU states supplied the utilization anchor. An anchor proposes a policy; it does not constrain the policy's realized occupancy to the desired region.

**Rate-gated queries.** An evaluated variant partitions $8{,}000$ support states into four quantile bins of the request-rate feature and constructs one centered query per bin. The observed rate selects a query at each decision; out-of-range values use the nearest end bin, and smoothing is disabled. This differs from primary search, which fixes one task for the rollout. Conditioning the support distribution also changes the reward embedding, so this is a heuristic query construction rather than the original unconditional reward projection. The composite policy switches tasks, whereas each forward prediction was trained for continuation under a fixed task. Its behavior is assessed empirically in Appendix A.5.

**Action-extraction variants.** A 32-nearest-neighbor mask restricts the greedy actor to actions observed among neighboring logged states, using Euclidean distance in the eight encoded features. Other variants replace actor scores with the minimum or mean of the two scalar value readouts $F^{(k)}(s, a, z)^\top z$. These variants are not part of the primary actor-based recipe.

### D.7 Training and evaluation diagnostics

The training monitor checks the logged `total_fb` value every $1{,}000$ updates, warning if it is non-finite or exceeds $1{,}000$ times its value at update $1{,}000$. None of the five supplied runs triggered this warning. Table 10 records their diagnostics; all five checkpoints remain in the reported means, including the weaker utilization seeds. Passing the numerical monitor does not establish reliable control performance or identify the cause of variation between seeds. At the final update, the mass cap rescales 47–58% of target rows and the largest pre-cap target entries are approximately $1.4$–$1.6 \times 10^3$. These observations show that the cap is active, but do not by themselves isolate its effect from the backbone or other training choices.

| | Logged `total_fb` | | | At 100k updates | | |
|---|---|---|---|---|---|---|
| Seed | 1k | 50k | 100k | Orth. off-diag. | Cap active | Actor loss |
| 42 | 22.82 | 336.71 | 376.51 | 60.66 | 57.8% | 3.78 |
| 43 | 21.45 | 314.03 | 361.44 | 58.88 | 56.9% | 2.59 |
| 44 | 19.08 | 259.65 | 327.95 | 59.37 | 53.0% | 2.39 |
| 45 | 22.14 | 311.53 | 319.94 | 58.73 | 47.3% | 2.36 |
| 46 | 21.66 | 300.38 | 425.44 | 57.30 | 47.6% | 2.53 |

Table 10: Per-seed diagnostics from the supplied residual-FB training audit. "Orth. off-diag." is the first term of Equation Equation (56); "Cap active" is the fraction of target rows rescaled. `total_fb` is the occupancy loss only (diagonal plus off-diagonal terms) and excludes the orthogonality regularizer. Loss values are compared within this training recipe.

Evaluation follows Appendix G: 266 held-out windows, 12 warmup decisions and 24 scored decisions per window, greedy actions, and evaluation seed 42. Architecture settings are restored consistently with each checkpoint, while inference settings are pinned by the evaluation configuration. Each result must cover all 266 windows with zero failed evaluations; incomplete cells are rerun serially. This coverage check concerns completed evaluations, not service reliability or resource-budget satisfaction.

## E Workload construction

We construct simulator request workloads from the BOOM dataset (Cohen et al., 2025), a collection of 2,807 real-world observability time series spanning infrastructure, networking, database, security, and application domains. We filter to the 954 *rate-type* shards identified by the `dataset_taxonomy.json`, as rate metrics directly represent event-arrival load. The 954 shards are split into train, validation, and test sets at an 80 / 10 / 10 ratio using a fixed random permutation, yielding 764 train, 95 validation, and 95 test shards; after generation, 720 train and 84 validation shards produced workloads (table below).

### E.1 RATE RECOVERY

BOOM stores each time series as a *z-scored* sequence: values are standardized to have approximately zero mean and unit standard deviation, so the original absolute scale is lost. This does not affect our analysis, since we care only about the temporal *shape* of the workload, not its absolute magnitude. We recover a non-negative req/s series from $\mathbf{z}$ at a chosen target mean rate $\mu_{\text{target}}$ via a **zero-floor constraint**. Over long horizons, real request rates probably reach zero at some time, so the minimum of the z-scored series, $z_{\min} = \min(\mathbf{z})$, likely map to a rate of zero in the recovered series. Solving $z_{\min} \cdot \sigma + \mu_{\text{target}} = 0$ for the scale parameter $\sigma$ gives

$$\sigma = \frac{-\mu_{\text{target}}}{z_{\min}}, \qquad r_i = z_i \cdot \sigma + \mu_{\text{target}}, \tag{66}$$

which maps $z_{\min}$ exactly to rate zero while placing the mean at $\mu_{\text{target}}$. This mapping preserves the temporal pattern of the original metric — peaks, troughs, and cyclical structure — while anchoring the scale to a user-specified operating intensity.

For approximately 4 of the 954 rate shards whose minimum z-score is non-negative, Equation Equation (66) is inapplicable. For these shards we fall back to *p95 normalization*: $r_i = (z_i / z_{p95}) \cdot \mu_{\text{target}}$, where $z_{p95}$ is the 95th percentile of the positive z-values.

After recovery, rates are clipped to $\geq 0$ and then globally rescaled so that the mean over the buckets used in event generation closely matches $\mu_{\text{target}}$. To represent low, medium, and high intensity conditions, we generate workloads at three target arrival rates: $\mu_{\text{target}} \in \{0.1,\ 1,\ 10\}$ req/s, and cap each workload at 24 hours of simulated time.

### E.2 BUCKET SIZE AND REQUEST EVENT GENERATION

BOOM time series are sampled at their original reporting frequency, which varies by shard and is stored as a pandas offset alias (e.g. `T` = 60 s, `5T` = 300 s, `H` = 3,600 s). We refer to each time-series interval as a bucket of width $w$ seconds, equal to the shard's native sampling period. Given the recovered per-bucket rates $\{r_i\}$, we generate request arrival timestamps using a piecewise-constant Poisson process: for each bucket $i$, the number of arrivals is drawn as $n_i \sim \text{Poisson}(r_i \cdot w)$, and their timestamps are sampled uniformly within $[t_i,\ t_i + w)$. Each request is assigned a fixed CPU cost (in number of cycles) and memory footprint (in bytes) drawn from one of five predefined *(CPU, memory) pairs* that span two orders of magnitude, representing workloads from lightweight document retrieves to heavy analytical queries. All requests are of type `GET`.

| CPU (cycles) | Memory (bytes) |
|---:|---:|
| 9,375,000 | 187,500 (≈183 KB) |
| 37,500,000 | 750,000 (≈732 KB) |
| 150,000,000 | 3,000,000 (≈2.9 MB) |
| 600,000,000 | 12,000,000 (≈11.4 MB) |
| 2,400,000,000 | 48,000,000 (≈45.8 MB) |

### E.3 DATASET SCALE

**Workloads generated.** For each train and validation shard we generate one workload per ($\mu_{\text{target}}$, CPU/mem pair) combination, yielding $3 \times 5 = 15$ workloads per shard. For each test shard we generate a single workload by sampling one ($\mu_{\text{target}}$, CPU/mem pair) uniformly at random (weighted toward medium intensity: $\mu = 1$ req/s with probability 0.6). This gives:

| Split | Shards | Workloads/shard | Workloads |
|---|---:|---:|---:|
| Train | 720 | 15 | 10,800 |
| Validation | 84 | 15 | 1,260 |
| Test | 95 | 1 | 95 |
| Total | 899 | — | 12,155 |

**One variate per shard.** BOOM shards may contain multiple variates (dimensions). We sample a single variate per shard deterministically using a fixed seed (`VARIATE_SEED = 0` + shard index), so each shard contributes exactly one time-series shape regardless of its dimensionality.

### E.4 Episode windows for offline data collection

Each workload is up to 24 hours long (288 steps at the 300 s agent control frequency). When running the data-collection controllers to generate offline transitions, we tile each workload into **2-hour windows** (24 agent steps) whose recorded episodes do not overlap:

- **Skipped phase** (12 steps, 1 hour): the workload slice starts at the window offset, so these steps are not simulated; the simulator and controller restart at the initial allocation, and the agent's observation history is initialized from the environment reset.
- **Episode phase** (12 steps, 1 hour): transitions $(\text{history}, a, r, s')$ are recorded and added to the dataset.

Windows are placed at offsets $\{12,\ 24,\ 36,\ \ldots\}$ steps from the start of the workload (stepping by the episode length of 12 steps), with the first window starting at step 12 to ensure the initial warmup phase reads real load data rather than the default initial state. A 24-hour workload therefore yields $\lfloor (288 - 24)/12 \rfloor = 22$ non-overlapping episodes.

Recorded episode intervals do not overlap, and simulator and controller state are reset between episodes. Windows from the same workload can nevertheless remain temporally correlated. Training and evaluation workloads are separated by the source-shard split.

## F Simulator

### F.1 Simulator overview

The simulator is a discrete-event simulator to emulate a distributed database system. The simulator models a cluster as a combination of physical components (`CPU`, `RAM`, `Disk`) and logical components (`Index`, `Shard`, `Worker`). A global clock backed by a priority queue coordinates all events, tracking each request from arrival through processing to response.

**Physical components.** Each `CPU` core processes tasks sequentially (no preemptive multithreading) at a fixed clock rate; a task's duration is computed as cycle_count/core_frequency. `RAM` has infinite effective bandwidth but finite capacity, acting as a constraint only when full. `Disk` has configurable read/write speeds and supports concurrent access. Both `CPU` frequency and `RAM` are temporarily taxed during scaling operations to model cluster reorganization overhead.

**Logical components.** An `Index` represents one dataset; a cluster may host multiple indices sharing the same workers. A `Shard` is a partition of an index assigned to one or more workers (replicas); reads are served by any replica, while writes must be sent to all replicas (returning after the first completes). A `Worker` is a database process bundled with a configurable amount of `CPU` cores and `RAM`, determined by the number of compute units assigned to it.

**Requests.** `GET` requests retrieve a document by id: the relevant shard replica is selected on the worker with the least load, which reads from disk and serializes the result using CPU and RAM. `PUT` requests write a document: the primary replica processes and writes the document to disk, then asynchronously replicates to remaining replicas; the request returns after the first write completes.

**Profiler.** Every $t = 300$ simulated seconds, components report utilization statistics, such as average CPU/Memory utilization and queue length, to a central profiler, matching common real-world reporting intervals.

**Scaling actions.** At each profiling period, the agent may issue autoscaling actions which are executed via a blue-green deployment pattern: new workers of the requested size are spun up, data

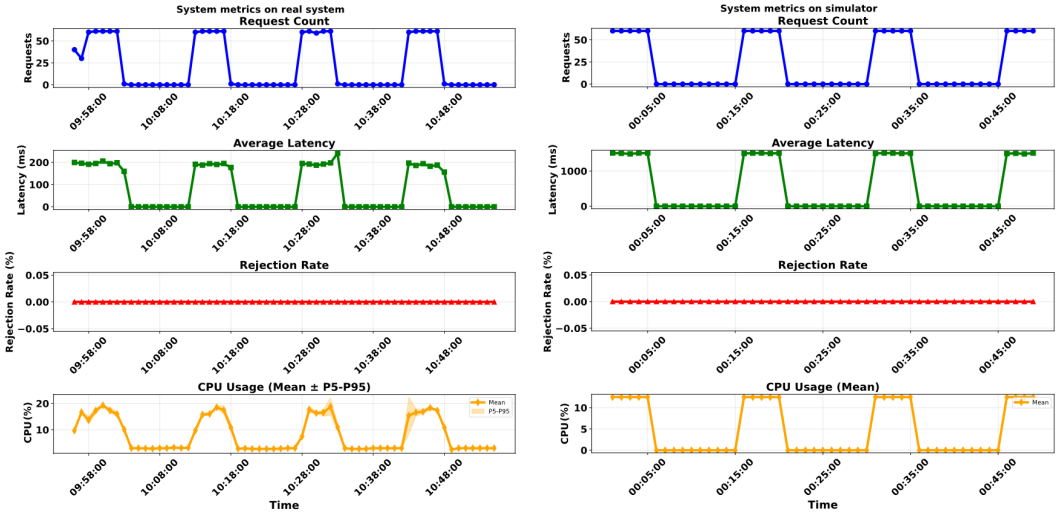
is transferred from old workers, and old workers are decommissioned. This process temporarily increases CPU load, reducing request throughput during the transition.

### F.2 Simulator validation

We validate the simulator by replaying two hour-long workloads against a real OpenSearch Serverless cluster and comparing system metrics. Both systems start with 2 workers and 2 CUs per worker (1 CU = 6 GiB RAM + 1 vCPU). The two workloads are: (1) a square-wave workload with capacity held fixed, and (2) a constant-rate workload with a forced scale-up from 2 to 8 CUs at the 30-minute mark.

We target *directional* rather than *numerical* correctness: the simulator intentionally omits OpenSearch-specific background tasks (logging, OS overhead) and does not replicate the precise details of OpenSearch's blue-green deployment or per-request cycle counts.

On the square-wave workload (Figure 7), latency and CPU utilization track each other in both systems and drop to zero during idle periods, with no rejections in either. On the constant-rate workload (Figure 8), both systems are flat until the blue-green deployment, after which latency and CPU utilization drop as expected. A minor discrepancy appears during the deployment itself: the real system reports total system CPU (including OS overhead), while the simulator counts only request-handling CPU, producing a brief dip-and-spike pattern in the simulated trace.




(a) System metrics from the real system.

(b) System metrics from the simulator.

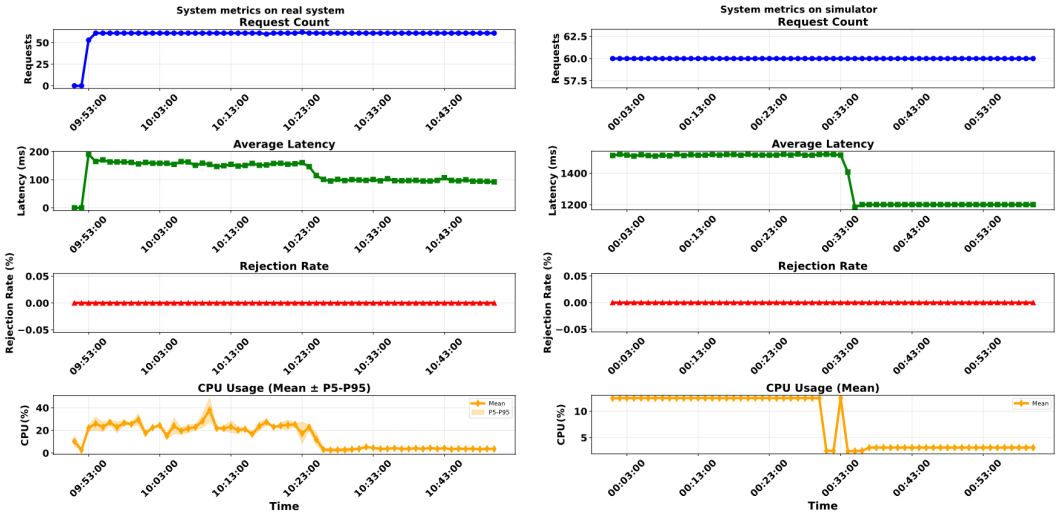
Figure 7: System metrics comparison on step function shape workload. Real system on the left and simulator on the right.

### F.3 Limitations

The simulator is intended as an evaluation harness rather than a perfect replica of production system. It explicitly models compute, memory, disk contention, shard routing, and scaling reconfiguration, but necessarily abstracts away aspects such as unobserved exogenous factors, rare failure modes outside the modeled resource limits, and workload-dependent operational quirks. Our empirical claims are therefore framed in terms of *performance under the simulator dynamics*.

## G Evaluation Protocol

All final results use a frozen suite of 266 disjoint three-hour windows drawn from 90 of the 95 held-out workload traces. Candidate windows are selected independently of learned-policy performance and partitioned into calm, borderline, and overload regimes according to demand.




(a) System metrics from the real system.

(b) System metrics from the simulator.

Figure 8: System metrics comparison on constant workload with capacity change around 30 minutes. Real system on the left and simulator on the right.

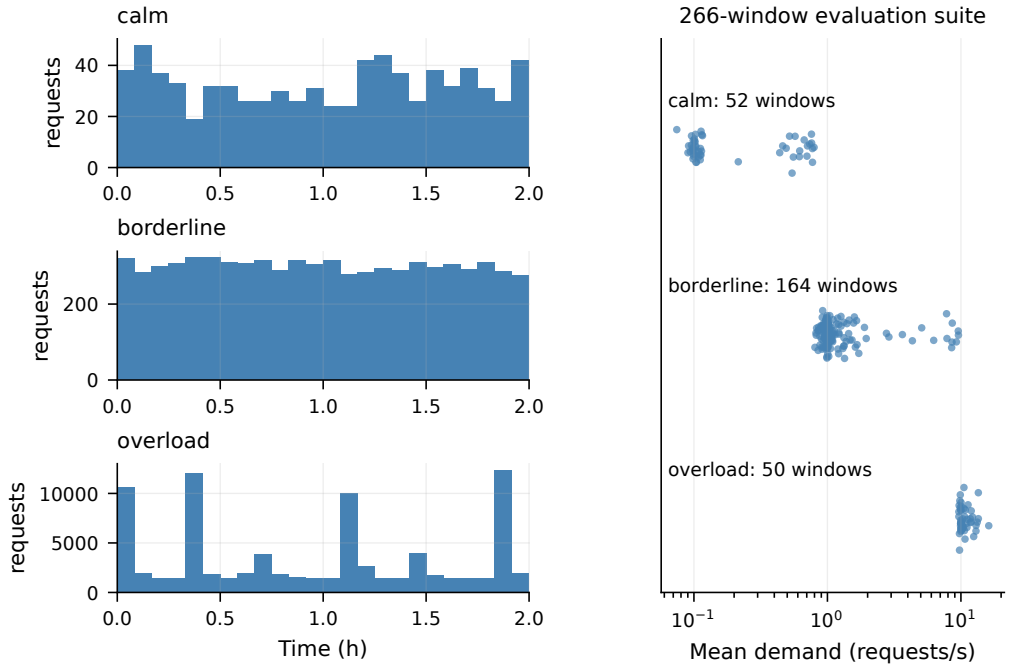



Figure 9: **The held-out evaluation suite.** Left: one lower-median mean-demand window per stratum, selected independently of policy performance. Right: all 266 windows, with fixed cosmetic vertical jitter. The suite contains 52 calm, 164 borderline, and 50 overload windows. Requests are shown for all 24 exported diagnostic rows.

Each evaluation consists of a common unscored warmup followed by 24 scored control decisions. Under the rejection reward, return divided by 24 is the mean interval acceptance fraction; intervals are weighted equally rather than by request count. All learned and rule-based policies are run through the same simulator initialization, warmup, and scoring path. Learned policies act greedily, with exploration disabled.

Unless stated otherwise, learned policies are trained with five random seeds. For each contrast, we first average the learned-policy seeds within each evaluation window and then compute paired differences between methods on the same windows. Throughout the main text, $\pm$ denotes the standard error across the five seed means; the appendix contrast tables additionally report 95% confidence intervals from 10,000 bootstrap resamples of these paired window-level differences. Pairing removes variation due to workload difficulty, which is substantially larger than evaluation noise.

Because multiple windows can originate from the same source trace, we additionally verify headline comparisons using a trace-clustered bootstrap. The rule controllers are re-scored under this same harness following a correction to per-workload controller-state reset, so every reported comparison against them uses the corrected values.

Table 11 resolves the three objectives against target tracking and step scaling as well as the logging controller; only its utilization block averages five training seeds, and every other cell is the single seed 42.

| Objective | Policy | return | Δ SCALEX | Δ target track. | Δ step scal. |
|---|---|---|---|---|---|
| utilization | SCALEX (logger) | 3.995 | | | |
| | target tracking | 5.385 | | | |
| | step scaling | 5.395 | | | |
| | IQL | 4.844† | **+0.85** | −0.54 | −0.55 |
| | BC | 4.135† | +0.14 | −1.25 | −1.26 |
| latency | SCALEX (logger) | 19.277 | | | |
| | target tracking | 15.921 | | | |
| | step scaling | 16.938 | | | |
| | IQL | 18.522 | −0.75 | **+2.60** | **+1.58** |
| | BC | 18.985 | −0.29 | **+3.06** | **+2.05** |
| rejection | SCALEX (logger) | 23.509 | | | |
| | target tracking | 21.679 | | | |
| | step scaling | 21.840 | | | |
| | IQL | 23.441 | −0.07[n.s.] | **+1.76** | **+1.60** |
| | BC | 22.631 | −0.88 | **+0.95** | **+0.79** |

Table 11: **Per-corner breakdown against three rule controllers.** Undiscounted return on the frozen 266-window suite, with paired differences against SCALEX, target tracking and step scaling. Rows are IQL and BC trained on pooled logs from SCALEX, target tracking, and step scaling (301,180 transitions, without cpu-SCALEX). Bold marks a positive contrast whose bootstrap interval excludes zero; [n.s.] marks a contrast whose interval excludes zero but whose paired Wilcoxon test gives $p > 0.05$; other contrasts have intervals that include zero. Seed coverage is asymmetric — see the footnote — so Table 6 is the primary comparison.

### G.1 ACTION SUPPORT IN THE CONTROLLER LOGS

## H REFERENCE CONTROLLERS

We use four controller configurations both to characterize rule-based performance and to construct the controller pool: latency-oriented SCALEX, CPU-targeting SCALEX, target tracking, and step scaling. Single-controller logs contain only latency-oriented SCALEX experience. The pooled datasets contain experience from all four configurations, with local exploration applied during collection only for the third dataset.

At decision epoch $t$, the current allocation is $c_t = (n_t, q_t)$, with total provisioned capacity $C_t = n_t q_t$. We write $C_t^{\text{des}}$ for a controller's continuous capacity request and $a_t^{\text{ctrl}} \in \mathcal{A}$ for its discrete allocation

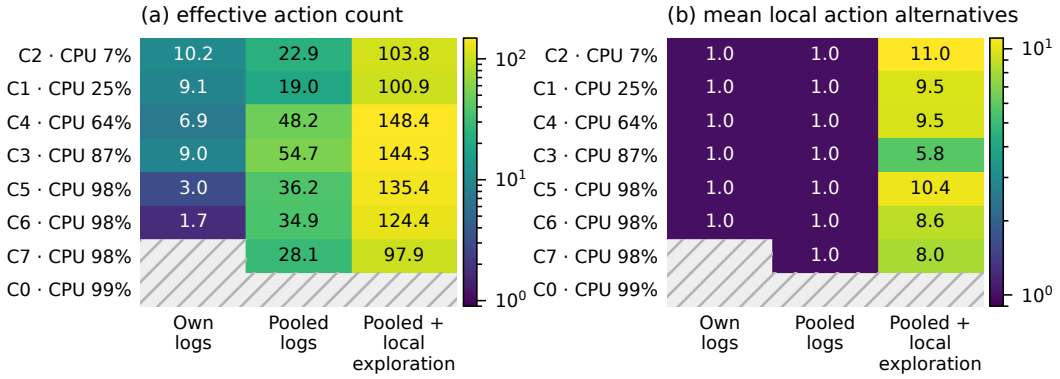



Figure 10: **Pooling broadens action coverage; local exploration adds alternatives.** Each cell combines a corpus and an operating cluster, with clusters ordered by mean CPU utilization. (a) Effective action count, $\exp H(A \mid \text{cluster})$, measures action diversity within a cluster. (b) Mean distinct applied actions per controller proposal measures alternatives introduced by exploration. Panels use separate logarithmic color scales; hatched cells have no measured support. Cluster C0 has no measured support in any corpus. These summaries do not establish action coverage at every individual state.

| Data | Policy | utilization | latency | rejection |
|---|---|---|---|---|
| | SCALEX (logging controller) | 4.00 | 19.28 | 23.51 |
| | cpu-SCALEX@0.70 (utilization reference) | 6.71 | 15.96 | 22.57 |
| | target tracking | 5.39 | 15.92 | 21.68 |
| | step scaling | 5.40 | 16.94 | 21.84 |
| $\mathcal{D}_0$ | IQL | $4.04 \pm 0.30$ | $19.13 \pm 0.11$ | $23.54 \pm 0.01$ |
| | BC | $4.02 \pm 0.09$ | $19.21 \pm 0.04$ | $23.55 \pm 0.01$ |
| $\mathcal{D}_1$ | IQL | $6.54 \pm 0.23$ | $18.92 \pm 0.18$ | $23.21 \pm 0.41$ |
| | BC | $5.86 \pm 0.69$ | $18.45 \pm 0.67$ | $22.84 \pm 0.16$ |
| $\mathcal{D}_2$ | IQL | $6.94 \pm 0.23$ | $18.80 \pm 0.13$ | $23.04 \pm 0.49$ |
| | BC | $4.01 \pm 0.67$ | $17.89 \pm 0.24$ | $22.16 \pm 0.17$ |

Table 12: **The dataset ladder.** Undiscounted return on the fixed 266-window held-out suite, mean $\pm$ std over five training seeds, for each dataset and each of the three corner objectives. Rule controllers are deterministic and reported once. The utilization column is the one in which improvement over the logging controller is available; latency and rejection are discussed in Section 6.1.

proposal. Without an exploration wrapper, the applied action is $a_t = a_t^{\mathrm{ctrl}}$. Every controller is restricted to the same 252 allocations used by the learned policies.

### H.1 LATENCY-ORIENTED SCALEX

Our SCALEX implementation follows the reactive control design of Quattrocchi et al. (2024). It adjusts capacity using the observed mean latency $\ell_t$ and an internal target

$$\ell^\star = \alpha \ell_{\mathrm{SLA}}. \tag{67}$$

The reported configuration uses $\alpha = 0.8$ and $\ell_{\mathrm{SLA}} = 0.2\,\mathrm{s}$, giving $\ell^\star = 0.16\,\mathrm{s}$. This controller target is distinct from the 0.8–1.2 s latency-penalty range used to score learned policies. We express latency in seconds throughout; the implementation converts to milliseconds when required internally.

**Capacity request.** The controller uses an asymmetric relative error,

$$e_t = \begin{cases} \beta_\uparrow(\ell_t/\ell^\star - 1), & \ell_t > \ell^\star, \\ \beta_\downarrow(\ell_t/\ell^\star - 1), & \ell_t \le \ell^\star, \end{cases} \qquad \beta_\uparrow = 0.3, \quad \beta_\downarrow = 0.1. \tag{68}$$

Positive latency errors therefore receive three times the gain of equally sized negative errors. An internal accumulator $I_t$ and a direct response term determine the requested capacity:

$$I_t = I_{t-1} + K_I e_t, \qquad C_t^{\mathrm{des}} = I_t + K_P e_t, \qquad K_I = 50, \quad K_P = 100. \tag{69}$$

The accumulator is initialized to the current capacity on the first call. The gains correspond to the implementation parameters `BC` and `DC`, respectively.

**Discrete allocation.** When more capacity is requested, the controller first increases per-worker capacity $q$ up to 4 units. It then increases the worker count to meet the request, using $n = \lceil C_t^{\mathrm{des}}/q \rceil$ subject to the 42-worker limit. If that limit is reached, it can increase per-worker capacity further, up to 16 units.

When less capacity is requested, the controller prioritizes reducing per-worker capacity while retaining workers. Worker removal is considered when a further reduction in per-worker capacity would undershoot the requested capacity. All proposed allocations remain within the action space. Capacity reductions require the low-demand condition to persist through a 300-second cooldown, corresponding to one control interval. Increases do not wait for this cooldown.

### H.2 CPU-TARGETING SCALEX

CPU-SCALEX uses the same accumulation rule, asymmetric gains, allocation procedure, and cooldown as latency-oriented SCALEX.

Its error signal replaces $\ell_t/\ell^\star$ in Equation Equation (68) with $u_t/0.70$. It therefore targets the same CPU-utilization setpoint used by the utilization reward.

We use this controller as an objective-aligned reference: it shows what an existing control rule achieves when configured for the utilization target. It is not an analytic optimum or a performance upper bound. CPU-SCALEX also contributes logs to the pooled datasets. Consequently, utilization is a different objective from that of the base latency-oriented controller, but it is not absent from the behavior used to construct the pool.

### H.3 TARGET TRACKING

Our target-tracking controller adjusts capacity in proportion to utilization relative to a target.

Let $\bar{u}_t$ and $\overline{\mathrm{rss}}_t$ denote CPU and resident-memory utilization smoothed over $W = 3$ control intervals, and define

$$v_t = \max\{\bar{u}_t, \overline{\mathrm{rss}}_t\}. \tag{70}$$

The target is $v^\star = 0.50$, with dead-band half-width $\delta = 0.15$. When $v_t \in [0.35, 0.65]$, the controller holds the current allocation. Outside this band, the proportional capacity request is

$$\widetilde{C}_t = C_t \frac{v_t}{v^\star}, \qquad C_t^{\mathrm{des}} = \min\Big\{\kappa C_t, \max\Big\{C_t/\kappa, \widetilde{C}_t\Big\}\Big\}, \qquad \kappa = 1.25. \tag{71}$$

This limits changes in the continuous capacity request; discrete quantization can change the realized capacity ratio. If $v_t < 0.01$, the controller holds capacity rather than applying the ratio rule.

A service guard takes precedence over these rules: if $\rho_t > 0.01$ or $\ell_t > 2.0\,\mathrm{s}$, the controller immediately requests $\kappa C_t$. The capacity request is converted to an allocation using a least-waste rule: among feasible allocations meeting the request, select one with the smallest excess capacity.

### H.4 STEP SCALING

Our step-scaling controller uses fixed multiplicative changes when utilization crosses specified thresholds.

Outside its lockout period, it requests

$$C_t^{\mathrm{des}} = \begin{cases} 1.5C_t, & u_t > 0.70 \text{ or } \mathrm{rss}_t > 0.75, \\ C_t/1.3, & u_t < 0.30 \text{ and } \mathrm{rss}_t < 0.35, \\ C_t, & \text{otherwise.} \end{cases} \tag{72}$$

The requested change depends on which thresholds are crossed, rather than the magnitude of the deviation. Following a scaling action, ordinary threshold-based updates are locked out for two control intervals. The same service guard used by target tracking, $\rho_t > 0.01$ or $\ell_t > 2.0\,\mathrm{s}$, bypasses the lockout and requests an immediate capacity increase by a factor of 1.5. The resulting request is mapped to an allocation in $\mathcal{A}$.

## I COLLECTING THE BROAD-POOL CORPUS

This appendix describesbroad-pool, a larger *broad-pool* corpus of 822,584 transitions. It is *not* the training corpus of the main results: every agent reported in the main text (IQL, behavior cloning, preference-IQL, and FB) is trained on the pooled logs with local exploration (401,631 transitions; Section 4 and Appendix K).broad-pool is used only in two supplementary analyses: the full-mix row of Table 7, and the earlier plain-MLP FB runs that selected occupancy-mass clipping and reward centering (Appendix A.3).

**What the dataset contains.** $\mathcal{D}_3$ is a set of 822,584 transitions $(s, a, r, s')$, where $s$ is the 8-dimensional state (Section 3), $a$ is one of the 252 discrete scaling actions, and $r$ is the logging reward (during training we relabel $r$ for the objective being optimised, as for every corpus). The transitions were produced by running a diverse set of *behaviour policies* in the simulator (Appendix F) on request traces derived from real workloads (Appendix E). Unlike the main corpora, which record 12 decisions per episode,broad-pool is collected as 24-step episodes (34,375 episodes). The dataset covers all 252 actions, so no action is unsupported.

**Which behaviour policies generated it.** The defining feature of the broad pool is that its behaviour policies are meant to look like how a real autoscaler is operated over time, not just a single controller run once. We use four families of behaviour policy:

1. **Realistic operation** (largest share). Three ways a deployed autoscaler changes over an operating period: (a) its target operating point slowly drifts (gradual retuning); (b) the operator switches between different autoscaling rules part-way through an episode (deploys, A/B tests, incident response); and (c) the controller's actions are perturbed by smooth, temporally-correlated exploration noise. These policies are the most varied, so we let them fill whatever fraction of the dataset the other families leave.
2. **Base controllers.** The three production autoscaling rules (Appendix H) — SCALEX, target-tracking, and step-scaling — run in their normal configuration with a small amount of exploratory action noise.
3. **Capacity-reference controllers.** Utilisation-targeting controllers at four target-utilisation levels, included to broaden state–action coverage. (The 0.70 level, CPU-SCALEX, is also the utilization topline reported in the main text.)
4. **Exploration and coverage.** A coverage-seeking exploratory policy (higher action noise), an anticipatory trend-following policy, and rollouts on synthetic stress workloads.

Table 13: **Behaviour policies in the broad-pool dataset** $\mathcal{D}_3$ (822,584 transitions, all 252 actions covered). "Cap" is the largest share a family is allowed to contribute; the realistic-operation policies are uncapped and absorb the remainder. A family can fall below its cap if it simply produced fewer transitions (the synthetic-workload policies).

| Behaviour policy | Family (cap) | Transitions | Share |
|---|---|---|---|
| Drifting target point | Realistic (uncapped) | 95,596 | 11.6% |
| Mid-episode controller switching | Realistic (uncapped) | 95,538 | 11.6% |
| Correlated exploration noise | Realistic (uncapped) | 95,620 | 11.6% |
| SCALEX | Base controllers ($\leq$30%) | 82,212 | 10.0% |
| Target-tracking | Base controllers ($\leq$30%) | 82,399 | 10.0% |
| Step-scaling | Base controllers ($\leq$30%) | 82,164 | 10.0% |
| Utilisation-targeting (4 target levels) | Capacity reference ($\leq$15%) | 123,387 | 15.0% |
| Coverage explorer | Exploration (10%) | 82,258 | 10.0% |
| Trend-following | Exploration (10%) | 82,258 | 10.0% |
| Synthetic workloads, normal control | Exploration ($\leq$15%) | 576 | 0.1% |
| Synthetic workloads, high exploration | Exploration ($\leq$15%) | 576 | 0.1% |
| **Total** | | **822,584** | **100%** |

**Balancing the mixture.** Because some behaviour policies produce far more data than others, we cap how much of the final dataset each family may contribute and subsample the rest, so no single behaviour dominates the batch. The realistic-operation policies are left uncapped and fill the remainder. Subsampling is uniform and uses a fixed random seed, so the corpus is deterministic and reproducible. Table 13 lists each behaviour policy, its role, and its final share of the dataset; the realistic-operation family ends up at 34.9%, with the base-controller and capacity-reference families at their caps of 30% and 15%.

**Relation to the other rungs, and held-out data.** The broad pool shares its base controllers with the main corpora; what makes it "broad" is the realistic-operation family and the additional exploration policies. To keep evaluation honest, two behaviour sources are excluded frombroad-pool (and from every training set) and reserved for a held-out test, so that no reported result is measured on data close to what it trained on. We release the collection script, the mapping from behaviour policy to transitions, and an audited datasheet (action coverage, capacity by load level, and behavioural diversity) alongside the dataset.

## J HYPERPARAMETERS AND IMPLEMENTATION DETAILS

The settings below describe the primary experiments, checked against the saved checkpoints, training configurations, and evaluation logs. Ablation-specific changes are identified with their results. BC and reward-specific IQL use seeds 42–46 and 100,000 updates on each data condition; a separate IQL policy is trained for each reward. Preference-IQL and FB use seeds 42–46 and 100,000 updates on pooled logs with local exploration (401,631 transitions); each FB seed contributes one final checkpoint that is reused across all objectives. Uncertainty across evaluation windows is distinct from uncertainty across training seeds.

**Observation and action encoding.** All primary learned policies consume eight state features (Table 14). The six CU tiers are $q_j \in \{0.5, 1, 2, 4, 8, 16\}$ with zero-based tier index $j \in \{0, \ldots, 5\}$. The joint action index is $a = 6(n-1) + j$, for $n \in \{1, \ldots, 42\}$, giving 252 actions. CU values must not be substituted for the tier index.

**Environment and reference controllers.** Control and history windows are both 300 seconds; frame stacking is disabled. The environment uses a steady-wait setting of 15 steps and starts from three workers with two CUs per worker. Evaluation uses 12 warmup intervals followed by 24 scored intervals on each of 266 held-out windows. Latency ScaleX uses a 160 ms internal target (the observed seconds are converted to milliseconds), accumulation gain BC$=50$, direct gain DC$=100$, up/down multipliers $0.3/0.1$, and a one-control-step scale-down cooldown. CPU ScaleX uses the same gains

Table 14: Eight-dimensional observation; request count is converted to requests per second over the 300-second window and scaled by 100 requests/s.

| Index | Signal | Policy input |
|---|---|---|
| 0 | Rejection rate | Fraction of requests rejected |
| 1 | Mean latency | Raw seconds |
| 2 | CPU utilization | Fraction |
| 3 | RSS utilization | Fraction |
| 4 | Workers $n$ | $n/42$ |
| 5 | CU per worker $q$ | $q/16$ |
| 6 | Steady state | Binary indicator |
| 7 | Request count $\nu$ | $\mathrm{clip}(\nu/(300 \cdot 100), 0, 1)$ |

and multipliers with a utilization target of $0.70$. Target tracking uses a three-interval smoothing window, target $0.50$, dead band $\pm 0.15$, and maximum change factor $1.25$; the implementation applies the CPU target to the larger smoothed CPU/RSS signal. Step scaling uses CPU out/in thresholds $0.70/0.30$, RSS thresholds $0.75/0.35$, an out multiplier of $1.5$, an in divisor of $1.3$, and two steps between actions. Both latter controllers have service guards at 2 s latency and 0.01 rejection. Controller state is reset between evaluation windows.

**BC and IQL.** Table 15 gives the primary learning settings. Each listed MLP has three ReLU hidden layers. BC uses orthogonal initialization (gain $\sqrt{2}$); reward-specific IQL initializes its critics, value function, and actor orthogonally. Preference-IQL rebuilds its critics and value function with the shared MLP's Xavier initialization; its actor uses orthogonal initialization. BC uses focal classification loss (exponent 2) and entropy regularization (coefficient $0.001$), without class weights or label smoothing. IQL uses standardized advantages, exponential AWR weights capped at 100, and no subsequent mean normalization of those weights. Its gradient-norm cap is 1; TD targets are bounded and window ends are treated as truncations, with snapshot bootstrapping. Rewards are not normalized.

Table 15: Primary BC and IQL settings. All AdamW optimizers use $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

| Setting | BC | IQL | Preference-IQL |
|---|---|---|---|
| Hidden width | 256 | 256 | 512 |
| Network input dimension | 8 | 8 | $8 + 8 = 16$ |
| Updates | 100,000 | 100,000 | 100,000 |
| Batch size | 256 | 512 | 512 |
| Optimizer | AdamW | AdamW | AdamW |
| Weight decay | $10^{-4}$ | $10^{-4}$ | $10^{-4}$ |
| Policy learning rate | $3\times10^{-4}$ | $10^{-4}$ | $10^{-4}$ |
| Critic/value learning rate | — | $3\times10^{-4}$ | $3\times10^{-4}$ |
| Discount | — | 0.99 | 0.99 |
| Target EMA coefficient | — | 0.005 | 0.005 |
| Expectile | — | 0.95 | 0.95 |
| AWR exponent scale | — | 3 | 3 |

**Preference sampling and reward parameters.** The implementation has eight preference coordinates: the five penalties described in the main text plus switching, upward resizing, and downward resizing penalties. Preferences are sampled independently per transition: 35% corner objectives, 15% preset mixtures, 5% residual resizing coordinates, and 45% continuous Dirichlet draws over the five main coordinates, with concentration $0.5$. The reported service–cost sweeps set the remaining weights to zero. Latency thresholds are $0.8/1.2$ s and capacity normalization is 672 CU. Preference-IQL training uses utilization thresholds $u_{\mathrm{target}} = u_{\max} = 0.70$, the same as the evaluated utilization objective. Service–cost sweeps do not use these utilization terms.

**Stabilized reward-free FB.** The forward ensemble contains two residual networks, each an input projection followed by two pre-LayerNorm residual blocks of width 256 with Mish activations

and skip connections, and a final LayerNorm; each takes the state, 100-dimensional task vector, and one-hot action ($8 + 100 + 252 = 360$ inputs). The backward network uses the same residual backbone on the state and outputs a normalized 100-dimensional embedding. The categorical actor has three ReLU hidden layers of width 256 and $8 + 100 = 108$ inputs. The saved configuration has no auxiliary reward loss ($q$-loss coefficient zero). Table 16 describes the residual primary model (`fb_arch=residual`).

Table 16: Stabilized FB settings. The row-mass cap bounds the mean positive occupancy in each target row, not each individual matrix entry.

| Setting | Value |
|---|---|
| Optimizer | Adam |
| Forward/backward/actor learning rates | $10^{-4}/10^{-4}/10^{-4}$ |
| Batch size / latent dimension | 1024 / 100 |
| Discount / target EMA coefficient | 0.99 / 0.01 |
| Gradient-norm cap / weight decay | 10 / 0 |
| Orthogonality coefficient | 1 |
| Positive target row-mass cap | 21 |
| Entrywise target clipping | Disabled |
| Forward ensemble / FB pessimism penalty | 2 / 0 |
| Actor extraction / inverse temperature | AWR / 1 |
| Actor pessimism penalty | 0.5 (inactive under AWR) |
| Actor advantage standardization / IWIS | Enabled / enabled |
| Training goal mixing ratio | 0.71 |
| Actor task source | Mixed; detached for the actor |
| Auxiliary reward-loss coefficient | 0 |
| Reward-query estimator at evaluation | Centered |

Task projection uses radius $\sqrt{100} = 10$. The saved `norm_z=false` flag does not disable the explicit task projection used in task sampling and reward inference. Reward-query evaluation overrides the saved raw estimator with the centered estimator and uses 2,000 support states by default; goal and occupancy queries use their respective interfaces. Primary configurations use the environment-scaled eight-dimensional observations.

## K Exploration strategies behind the training corpora

Section 4.3 trains on three corpora: ScaleX's own logs, pooled logs, and pooled logs with local exploration. This appendix records how each was collected, using only what is fixed in the generation scripts and dataset metadata. Appendix H describes the controllers themselves; Figure 10 measures the resulting action support.

**Common collection setup.** Every cell is collected with the same generator on the same fixed set of 400 training traces (40 BOOM training series, each at 1 and 10 requests/s with all five CPU–memory profiles), with 300 simulator workers. Each trace is cut into consecutive, non-overlapping one-hour episodes of 12 five-minute decisions, each starting from a freshly reset simulator at the same trace offset, so every controller sees the same episodes. Controller state is reset at every episode boundary. Every row stores both the action the controller asked for and the action actually applied, so any deviation between the two is measurable after the fact.

**Own logs.** The logging controller ScaleX runs on its own with no perturbation: 8,371 episodes, 100,382 transitions, and 82 of the 252 lattice actions ever applied. Since ScaleX is deterministic, every state carries exactly one action (Figure 10b).

**Pooled logs.** Three further rule-based controllers are run on the same 400 traces and the same episodes, each unperturbed: target tracking (100,420 transitions, 80 distinct actions), step scaling (100,378, 49), and cpu-ScaleX@0.70, the ScaleX rule driven by CPU utilization with setpoint 0.70 (100,388, 60). The four cells are concatenated without subsampling or reweighting, so each contributes one quarter of the 401,568 rows; a per-row controller id is kept. The union applies 168

distinct actions, but each controller remains deterministic, so the number of applied actions per controller action is still one.

**Pooled logs with local exploration.** The same four controllers are run again on the same traces and episodes, but each commanded action is passed through a small correlated perturbation before it reaches the simulator. Per lattice axis $k \in \{\text{workers}, \text{CU tier}\}$, a latent offset $x_k$ follows an Ornstein–Uhlenbeck (AR(1)) process,

$$x_k \leftarrow \rho\, x_k + \sigma\, \xi_k, \qquad \xi_k \sim \mathcal{N}(0,1), \qquad \delta_k = \mathrm{clip}\big(\,\mathrm{round}(x_k),\, -R,\, R\big), \tag{73}$$

with $\rho = 0.8$, $\sigma = 0.4$ lattice steps, and $R = 2$; the applied action is the commanded action displaced by $(\delta_{\text{workers}}, \delta_{\text{tier}})$ and then clamped to the lattice (1–42 workers, six CU tiers), so it is always one of the 252 valid actions. The offset is mean-zero, persists for about $1/(1-\rho) = 5$ decisions, and never moves more than two positions per axis; there is no separate hold or dwell rule beyond this autocorrelation, and no other safety guard than the lattice clamp. The latent state is zeroed at every episode boundary, and the noise stream is seeded deterministically from (worker index, episode index) via a SHA-256 hash, so a cell is reproducible. The reward stored with each row is computed for the applied action. Pooling is as above: 401,631 rows, one quarter per controller. The applied action differs from the commanded one on 51.7% of decisions overall (46.7% for SCALEX, 53.3% target tracking, 54.0% step scaling, 52.8% cpu-SCALEX@0.70); all 252 lattice actions occur, and every controller now records several applied actions for the same commanded action.

| Corpus | Sources | Traces | Transitions | Distinct actions |
|---|---|---|---|---|
| Own logs | SCALEX | 400 | 100,382 | 82 / 252 |
| Pooled logs | SCALEX, target tracking, step scaling, cpu-SCALEX@0.70 | 400 | 401,568 | 168 / 252 |
| Pooled + local exploration | the same four, each perturbed as in Eq. Equation (73) | 400 | 401,631 | 252 / 252 |
| SCALEX + local exploration | SCALEX, perturbed as in Eq. Equation (73) | 400 | 100,357 | 218 / 252 |

Table 17: **The training corpora.** The last row is the SCALEX slice of the pooled-plus-exploration corpus, used on its own in Table 1 to separate exploration from pooling. All are collected on the same 400 traces and the same one-hour episodes; "distinct actions" counts lattice actions applied at least once. Per-regime effective action counts are in Figure 10; downstream returns are in Tables 1 and 6.