

ARGOS: Reinforcement Learning-Driven Multidimensional Elasticity for Service Orchestration in the Computing Continuum

Javier Mateos-Bravo^{a,*}, Sergio Laso^{a,b}, Juan Luis Herrera^a, Ilir Murturi^c,
Pantelis Frangoudis^d, Schahram Dustdar^{d,e}

^a*Department of Computer Science and Telematics Engineering, University of Extremadura, Spain*

^b*Global Process and Product Improvement S.L., Spain*

^c*Department of Mechatronics, University of Prishtina, Kosova*

^d*Distributed Systems Group, TU Wien, Austria*

^e*ICREA, Barcelona, Spain*

Abstract

Data-intensive services in the Computing Continuum must balance analytics quality, resource usage, and cost across heterogeneous nodes with limited and uneven capacity. This balance becomes especially difficult when resource scaling reaches capacity limits, because changes in demand and cluster pressure must then be absorbed without violating client-defined quality ranges. Existing orchestrators mainly adapt resources, placements, or replicas, while analytics requirements such as coverage, sample, and freshness remain fixed. This article presents ARGOS, the Adaptive Reinforcement Learning-Driven Governance for Orchestrated Services, an end-to-end controller that formulates multidimensional elasticity as a per-request Markov decision process over analytics quality and cluster pressure, supported by capacity-aware admission. ARGOS is evaluated under controlled workloads and time-varying multi-tenant arrivals on a heterogeneous cluster. Across the controlled scenarios, the deep reinforcement learning policies consistently outperform the non-learning baselines and approach the independently tuned best-fixed reference. A separate live evaluation reports improvements over the static midpoint under realistic and saturated arrivals, with no recorded CPU or memory violations but remaining coverage violations. These results support deep reinforcement learning as an adaptive mechanism for multidimensional elasticity when resource scaling alone is insufficient.

Keywords: Multidimensional elasticity, Reinforcement learning, Service orchestration, Computing Continuum, Adaptive quality of service

1. Introduction

Data-intensive services increasingly run across the Computing Continuum, a layered fabric in which cloud, fog, edge, and mist resources cooperate to move processing closer to where data is produced while preserving access to larger compute pools [1–3]. In this article, the mist tier denotes the far-edge Internet of Things (IoT) devices, smartphones, and embedded sensors that generate or first process the data. This fabric is heterogeneous by design because its nodes differ widely in compute capacity, network conditions, energy profile, and availability. Consequently, elasticity cannot be reduced to adding or removing replicas as in a homogeneous cloud. The original notion of elastic processes already framed adaptation along three axes of

*Corresponding author. Email: javiermb@unex.es

resources, cost, and quality [4]. Recent work on adaptive data analytics in the Computing Continuum extends that view to three application-level requirements [5, 6]. These requirements are coverage, the fraction of spatial or user partitions included in an analysis, sample, the fraction of available observations processed within those partitions, and freshness, the interval between successive result updates. For instance, in an urban-mobility heatmap service, coverage controls geographic breadth, sample controls observation density within those partitions, and freshness controls result recency. Similar quality-resource trade-offs may arise in other configurable analytics services, although the relevant dimensions and their interpretation are application-dependent. Their accepted ranges therefore form part of the request’s Service Level Objective (SLO) and expose explicit trade-offs between analytical quality, CPU load, memory pressure, network traffic, and operational cost.

The practical problem starts when infrastructure elasticity reaches its limits. On the constrained tiers of the Continuum, physical capacity is finite and cannot be expanded on demand. Horizontal and vertical scaling therefore cease to provide additional capacity once the available nodes reach their limits [7, 8]. An unanticipated demand burst on a capacity-limited edge node can then leave the orchestrator with no additional resources to allocate, while requests already accepted by the system must still satisfy their client-defined SLOs. Multidimensional elasticity provides an additional adaptation space by allowing application-level requirements to vary within explicit client ranges. Existing frameworks for multidimensional elasticity have studied this principle for adaptive analytics [5, 6]. For configurable analytics, jointly controlling coverage, sample, and freshness on heterogeneous nodes while accounting for observed resource pressure and per-request ranges remains an operational challenge.

Three gaps prevent existing orchestration work from closing this challenge. The first gap (G1) concerns the control space: much reinforcement learning (RL) work for systems optimizes placement, scheduling, replica count, or batching rather than the analytics requirements embedded in per-request contracts [9–11]. The second gap (G2) concerns evaluation. Online reward traces may still include exploration and parameter updates, so they do not necessarily represent the frozen policy that would be deployed [12, 13]. Comparisons with non-learning alternatives must also use the same evaluation conditions. The third gap (G3) concerns reproducibility: evaluations frequently rely on single seeds, uncontrolled hyperparameter changes, or incomplete reporting of experimental settings and artifacts despite repeated calls for sufficient seed counts, explicit variability reporting, and auditable evidence [12, 14–16]. Accordingly, G1 requires a controller that adjusts coverage, sample, and freshness while observing resource pressure in the same decision step. G2 requires a protocol that trains each learned policy before evaluating the frozen policy artifact without exploration or parameter updates, alongside non-learning policies under the same conditions. G3 requires the preservation of policy fingerprints, decision logs, telemetry, and provenance for each comparison.

This article presents ARGOS, the Adaptive Reinforcement Learning-Driven Governance for Orchestrated Services, an orchestration framework for per-request multidimensional elasticity in the Computing Continuum. Structurally, ARGOS combines capacity-aware admission of analytics requests, per-request action selection, configuration distribution to heterogeneous worker nodes, runtime monitoring, and append-only audit records. Algorithmically, ARGOS represents each admitted request as a Markov decision process (MDP) whose state combines cluster pressure and the current analytics configuration, while its actions adjust coverage, sample, and freshness within the accepted request ranges. This common formulation supports both learning and non-learning policies with a shared state representation, action space, safety constraints, and configuration path, enabling systematic comparison under common orchestration conditions.

The contributions of this article are threefold. (i) An end-to-end orchestration design for multidimensional elasticity on heterogeneous clusters, coupling capacity-aware request admission with per-request control of coverage, sample, and freshness under observed resource pressure. The design constrains target

values to client-defined ranges, applies resource-safety guards, and records target and realized configurations separately, and its implementation is released as open-source software. (ii) A reproducible methodology for comparing decision policies under a shared orchestration model. Learned policies are trained before frozen evaluation, learning and non-learning approaches share the same state representation, action space, and operational constraints, and each run retains policy fingerprints, decision logs, reward components, node telemetry, and provenance. (iii) A two-stage empirical evaluation on an urban-mobility heatmap service over Seville using simulated per-user movement trajectories [17]. The controlled stage compares tabular Q-learning [18, 19], Deep Q-Network (DQN) [20], and Proximal Policy Optimization (PPO) [21] against the static midpoint and reactive threshold baselines and an independently tuned best-fixed reference across five workload profiles and two runtime modes. The live stage then compares the frozen deep policies with the static midpoint under realistic and saturated request arrivals, separating admission behavior, contract compliance, and deployability from controlled model ranking. The evaluation scenario, scripts, result tables, and figure-generation code are also released with the public artefact.

The remainder of this article is organized as follows. Section 2 grounds the operational problem in a city-scale urban-mobility service. Section 3 reviews multidimensional elasticity in the Computing Continuum, RL controllers for orchestration, and reproducible evaluation practices for learned-versus-non-learning comparisons. Section 4 presents the ARGOS architecture, its components, its operational lifecycle, and its decision model. Section 5 reports the controlled benchmark, frozen-policy comparison, runtime and resource-pressure analyses, and live-trial validation. Section 6 interprets the evidence and states the limits of the claims. Section 7 summarizes the findings and outlines future extensions.

2. Motivational use case

As a concrete motivating scenario, consider a city that operates a shared analytics service for urban mobility. The service ingests movement trajectories gathered from sources such as public-transport fleets and connected user devices and computes heatmaps of where movement concentrates within a requested geographic scope [17]. A single service can support several municipal use cases concurrently, submitted as independent requests by public bodies and private operators. The following four examples illustrate how their requirements differ:

- **Infrastructure planning.** The city planner analyzes long-term mobility patterns to plan new transport lines, bus stops, or pedestrian areas. This use case needs a broad city-wide view (high coverage), yet it can tolerate a heatmap refreshed only every few minutes (a long freshness interval).
- **Event management.** City authorities monitor large public events such as concerts or football matches, where crowds concentrate around one district. A high sample fraction of the local trajectories is needed to reveal the build-up and support the redirection of pedestrian flows.
- **Traffic and emergency management.** The traffic department reacts to forming jams and clears routes for emergency vehicles such as ambulances. The heatmap must be refreshed within seconds (a short freshness interval), because stale information can hide a rapidly changing incident.
- **Passenger-flow optimization.** Public-transport operators increase bus frequency where passengers currently concentrate and redesign routes from historical patterns. Current adjustments require a short freshness interval, whereas longer-term planning benefits from a broad city-wide view (high coverage).

These requests run concurrently and impose different coverage, sample, and freshness requirements on the finite resources of the same infrastructure.

Each request therefore arrives with an SLO contract that specifies accepted ranges for coverage, sample, and freshness and may also include a cost budget. Moving toward broader coverage, a larger sample fraction, or a shorter freshness interval increases the amount or frequency of processing and therefore raises compute, memory, network demand, and estimated operating cost. The platform can absorb growing demand for a while by distributing work across available nodes, delaying new analyses, or rejecting requests whose minimum requirements cannot be accommodated. Once admitted analyses compete for the finite capacity of the same infrastructure, however, the accepted quality ranges provide an additional adaptation space.

If the platform selects one operating point at admission and then holds coverage, sample, and freshness fixed for the lifetime of a request, that flexibility cannot be used as shared resource pressure changes. Figure 1 illustrates this problem for three of the use cases above. Panel (A) represents coverage in infrastructure planning, Panel (B) represents sample in event management, and Panel (C) represents freshness in traffic and emergency management. In each case, the corresponding requirement remains fixed after admission even though the conditions under which concurrent analyses share the infrastructure can change. An analysis fixed at a demanding operating point continues consuming scarce capacity when a less demanding configuration would remain within its accepted range. Conversely, an analysis fixed at a conservative operating point cannot use newly available capacity to improve its result. A fixed operating point therefore cannot exploit the flexibility already granted by the client as resource pressure varies.

Within this scenario, the flexibility granted by the accepted ranges motivates an orchestration mechanism that can adjust coverage, sample, and freshness as shared resource pressure changes, while preserving client-defined target bounds and resource-safety constraints. ARGOS addresses this need by treating the three requirements as per-request control dimensions rather than fixed values. This allows each admitted request to move among feasible operating points according to the measured state of the infrastructure. Section 4 describes the admission, control, and monitoring mechanisms that implement this behavior.

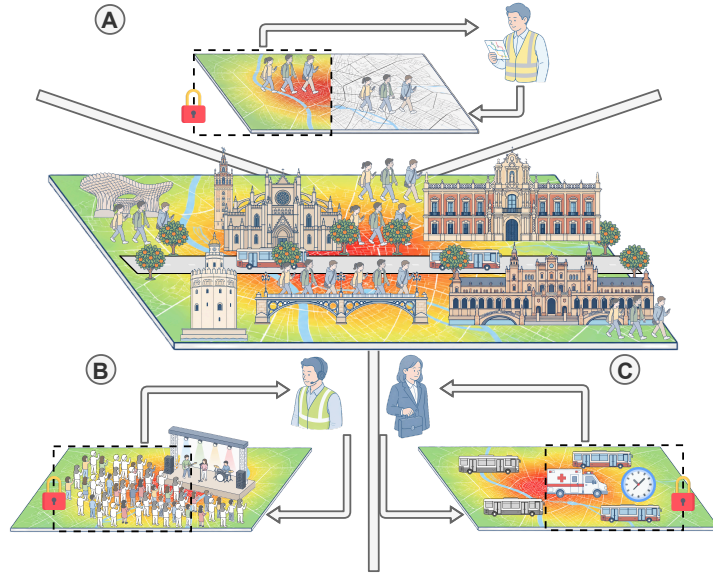


Figure 1: (A) Coverage, (B) sample, and (C) freshness fixed after admission. Padlocks mark what is held fixed in each use case.

3. Related work

The literature relevant to ARGOS spans three connected areas. Section 3.1 reviews the evolution from infrastructure autoscaling toward application-level and multidimensional elasticity in the Computing Continuum. Section 3.2 examines reinforcement-learning controllers for resource management and orchestration, including scaling, placement, and scheduling. Section 3.3 considers the baselines and reproducibility practices used to evaluate learned controllers. The analysis of each area then identifies the limitation that motivates the corresponding design choices in ARGOS.

3.1. Multidimensional elasticity in the Computing Continuum

Cloud and edge elasticity initially focused on adapting infrastructure resources to changing demand. Lorido-Botran et al. [8] review autoscaling techniques that adjust provisioned resources while balancing service objectives and cost. Casalicchio [22] examines the metrics used to trigger container autoscaling and evaluates CPU-based replica adaptation in Kubernetes. Rossi et al. [23] extend this resource-oriented view by coordinating horizontal and vertical scaling through reinforcement learning. Other approaches move adaptation closer to application requirements. Trihinas et al. [24] adapt monitoring through dynamic sampling and filtering on constrained IoT devices, Khaleq and Ra [25] couple reinforcement-learning autoscaling with response-time constraints, and Murturi and Dustdar [26] capture application elasticity requirements for decentralized runtime enforcement at the edge. These works broaden elasticity beyond fixed resource thresholds, but they do not jointly control coverage, sample, and freshness within the accepted ranges of each analytics request.

A complementary line treats analytical quality itself as elastic. Laso et al. [6] introduce elastic data analytics as a way to adapt analytics behavior to shared and constrained edge infrastructures. Their framework formalizes multidimensional elasticity through coverage, sample, and freshness for analytics services in the Computing Continuum [5]. This framework provides the conceptual base for ARGOS. Its testbed characterizes the impact of all three analytics requirements, while its Q-learning study adapts resource capacity, sample, and freshness over homogeneous simulated nodes and leaves coverage outside the learned state and action spaces. ARGOS builds on that foundation by controlling coverage, sample, and freshness with observed resource pressure in one per-request MDP deployed over heterogeneous nodes.

3.2. Reinforcement-learning controllers for orchestration

Reinforcement learning has been applied to several resource-management problems in cloud and cluster systems. Tesauro et al. [27] combine reinforcement learning with queueing models for autonomic server allocation, using offline learning before the learned policy controls the system. DeepRM [9] formulates cluster resource management as a deep reinforcement-learning problem in which jobs with multiple resource demands are scheduled from experience. Decima [10] extends learned scheduling to data-processing clusters by optimizing the execution of dependency-aware workloads. These systems demonstrate that learned controllers can support resource allocation and scheduling, but their actions determine resource assignment or execution order rather than the analytics requirements of individual requests.

Later work applies reinforcement learning to increasingly heterogeneous orchestration settings. Garí et al. [28] formulate scientific-workflow autoscaling as an MDP and use Q-learning to trade execution time against cloud cost. Goudarzi et al. [29] use distributed deep reinforcement learning for application placement across edge and fog resources. Wang et al. [30] implement a learned scheduler in FogBus2 that selects execution nodes while considering response time and load balance. Mampage et al. [31] apply deep reinforcement learning to function scheduling in a multi-tenant serverless environment and evaluate it on a deployed Kubernetes cluster. These studies extend learned orchestration to heterogeneous and deployed

infrastructures, but their control targets remain scaling, placement, or scheduling. ARGOS instead controls coverage, sample, and freshness within client-defined ranges while observing shared resource pressure.

3.3. Evaluation rigor: baselines and reproducibility

The conclusions drawn from a learned controller depend in part on the policies used for comparison. Rossi et al. [13] use reinforcement learning to adapt scaling thresholds rather than relying on fixed single-metric thresholds, showing how the reference policy can affect the resulting comparison. Mampage et al. [31] evaluate their learned serverless scheduler against several scheduling alternatives on a deployed cluster. Garí et al. [11] survey RL-based autoscaling and report substantial variation in state representations, actions, rewards, and evaluation designs. These works motivate comparisons against several meaningful alternatives rather than against a single simple baseline.

Reproducibility introduces a concern. Henderson et al. [12] show that deep reinforcement-learning results can vary substantially with random seeds, network architectures, hyperparameters, and implementation choices. Colas et al. [14] study how the number of random seeds affects the reliability of comparisons between stochastic learning algorithms. Agarwal et al. [15] show that aggregate conclusions can be unstable with limited runs and propose more robust statistical summaries, while Pineau et al. [16] consolidate reproducibility practices for machine-learning research. For orchestration experiments on heterogeneous hardware, these results motivate separating training from evaluation, retaining seed-level outcomes, and preserving the configuration, decision, and telemetry evidence required to trace each policy.

Across the three areas reviewed above, the remaining gap lies in their combination. Multidimensional elasticity establishes coverage, sample, and freshness as adaptable analytics requirements, while RL-based orchestration provides mechanisms for sequential control over shared infrastructure but primarily targets scaling, placement, or scheduling. Evaluation research further shows the importance of meaningful comparison policies, repeated runs, and reproducible experimental evidence. ARGOS combines these elements by controlling all three analytics requirements under observed resource pressure, comparing learned and non-learning policies under the same orchestration model, and separating policy training from frozen evaluation while retaining traceable experimental evidence. Table 1 summarizes this positioning through the control target, evaluation setting, and reported comparison design of representative orchestration approaches.

Study	Approach and control target	Evaluation setting	Reported comparison
Autonomic allocation [27]	Hybrid RL, server allocation	Web-server resource management	Model-based allocation
DeepRM [9]	Deep RL, cluster resources	Simulation	Resource-management heuristics
Decima [10]	RL, dataflow scheduling	Data-processing cluster	Scheduling alternatives
Workflow autoscaling [28]	Q-learning, workflow autoscaling	Cloud workflow evaluation	Autoscaling alternatives
Application placement [29]	Distributed deep RL, IoT placement	Simulation and testbed	Placement alternatives
Dynamic thresholds [13]	RL-tuned thresholds, application scaling	Simulation and prototype	Scaling policies
Continuum scheduling [30]	Deep RL, edge/fog scheduling	FogBus2 testbed	Scheduling alternatives
Serverless RL [31]	Deep RL, serverless scheduling	Kubernetes cluster	Baseline schedulers
ARGOS, this work	Per-request MDP, analytics quality	Heterogeneous cluster	Midpoint, threshold, and best-fixed

Table 1: Representative orchestration approaches and evaluation designs.

4. ARGOS

ARGOS is a deployment-level orchestration framework that mediates between tenant applications and analytics runtimes distributed across the Computing Continuum. Figure 2 separates this deployment into three architectural domains: external clients that express requests as analytics contracts, the ARGOS orchestrator that admits and controls those requests, and the managed Continuum whose nodes execute the applications. Using the current runtime context, the orchestrator selects values for coverage, sample, and freshness within the ranges accepted by each request and translates them into placement and configuration instructions, while every managed node exposes a uniform control and telemetry interface. This separation keeps the orchestration logic independent of a particular analytics implementation and confines service-specific behavior to the node-side runtime adapter.

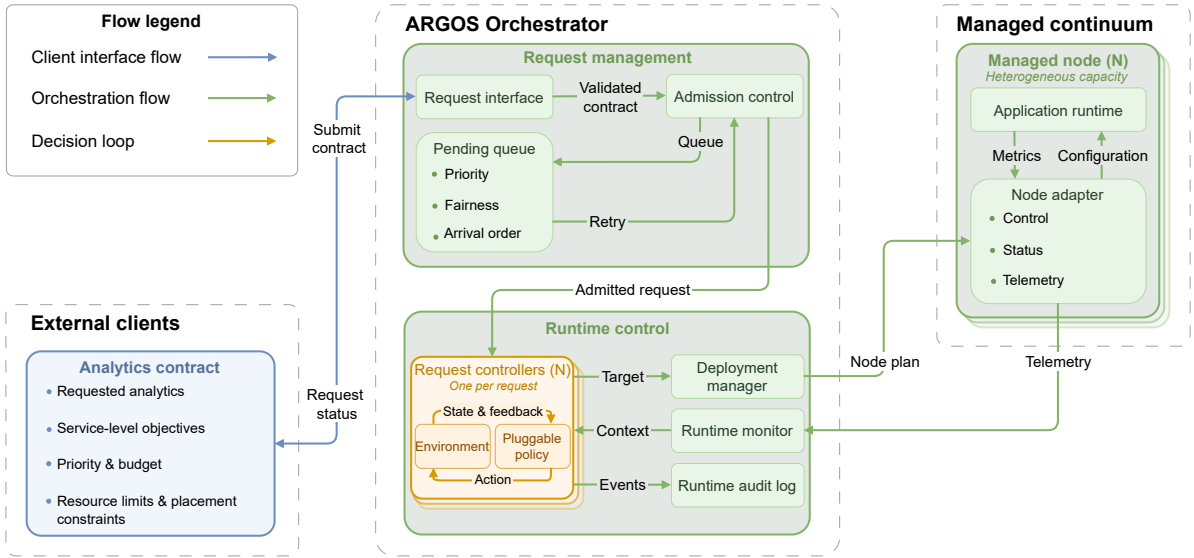


Figure 2: ARGOS deployment architecture and runtime interfaces.

The architecture supports configurable analytics services through a common service interface. The urban-mobility heatmap described in Section 2 is the current prototype instantiation of this interface. A service integrated with ARGOS must accept target quality values, execute through the placement selected by the orchestrator, and report realized quality and resource telemetry during execution. In the current prototype, each assigned node processes one trajectory partition and produces one heatmap shard. Target coverage determines the desired number of partitions, sample controls the fraction of trajectory observations processed within each partition, and freshness controls the update interval. Section 4.1 details the three domains in Figure 2, Section 4.2 describes the operational lifecycle, and Section 4.3 formalizes the decision model. The ARGOS source code, configuration files, evaluation scripts, and reproducibility artefacts are publicly available on Zenodo [32].

4.1. Components

External clients comprise tenant applications, operators, and services that request analytics without depending on the internal controller. They submit an **Analytics contract** through the **Request interface**, the public application programming interface (API) that also exposes request and cluster status. The contract carries accepted ranges for coverage, sample, and freshness together with the tenant identity, priority,

resource caps, and optional cost and placement constraints. This public interface is the architectural entry point and is also used by experimental workload generators acting as ordinary external clients during validation.

The **ARGOS orchestrator** coordinates request admission, per-request control, and deployment across the managed Continuum. It maintains the cluster view, request lifecycle, admission state, per-request controllers, and the node plans sent to managed nodes. Although one active orchestrator is sufficient for the reference deployment, the role is not tied to a specific machine. In distributed deployments, peer heartbeats, capacity-aware leader election, and a message bus allow another eligible node to assume the active orchestrator role.

The **Admission control** component validates each contract and determines whether its minimum requirements can be placed under the current capacity and placement limits. A structurally infeasible request is rejected, and the Request interface returns the rejection status and reason to the client. A feasible request with available capacity is admitted, whereas a temporarily blocked request enters the **Pending queue** and is returned with its current queue position. Queued requests are ordered by priority, tenant fairness debt, and arrival order, then reconsidered when capacity changes. Admission therefore protects the managed infrastructure before any policy begins to adapt an accepted request.

Each admitted request receives an independent **Per-request controller**. This design keeps the control state and policy history of each request separate, so the state or learning history of one request is not used directly to select actions for another. The controller combines the **Environment** and **Pluggable policy** roles shown together in Figure 2. The environment converts contract position, cluster telemetry, service measurements, and admission context into the state, masks ineffective or unsafe actions, applies the selected adjustment with a final range clamp, and computes its reward. Its policy then maps that state to an action and may be deterministic or learned, but every alternative uses the same state, actions, masking rules, and configuration path described in Section 4.3.

The **Runtime monitor** polls the node adapters and aggregates CPU, memory, bandwidth, latency, realized quality, and request-lifecycle information into the context consumed by each controller. The **Deployment manager** converts target coverage into a desired node count, selects nodes with available capacity, and distributes the complete target configuration. With N active nodes, the desired count is $\lceil \text{coverage} \cdot N \rceil$, lower-bounded at one and subject to the request’s placement limits, while realized coverage is the assigned-node count divided by N . This discrete mapping can differ from the continuous target, especially in a small cluster, so the **Runtime audit log** records both values together with lifecycle transitions, decisions, configurations, telemetry, violations, and errors. Runtime auditing is part of ARGOS because an operator needs traceability, whereas campaign aggregation and figure generation remain external validation tasks.

The **Managed Continuum** contains heterogeneous nodes that expose the same **Node adapter** to the orchestrator. Each adapter applies versioned node plans, supervises one or more **Application runtimes**, and reports host and service telemetry through its probe. The architectural role of an application runtime is generic, but the current prototype implements a geographic heatmap adapter. That adapter partitions trajectories by assigned node, applies the selected sample and freshness values, and reports processing time, data volume, spatial fidelity, and update age. Thread and process execution are implementation modes of this adapter rather than separate architectural components.

4.2. Operational lifecycle

The architecture exposes two runtime paths over the life of a request, drawn in Figure 2 as its flow legend. The request path follows the contract-submission and orchestration flows: it validates, admits, queues, or rejects a submitted contract. The adaptation path corresponds to the decision loop inside each request controller together with the orchestration flows around it: it repeatedly observes an admitted request

and updates its operating point. Before either path becomes active, the orchestrator registers the managed nodes, checks their health, and records the initial capacity view. Request processing begins only after one active orchestrator and the required node adapters are ready.

The request path starts with tenants such as those behind the use cases of Section 2. Each contract carries ranges $[c_{\min}, c_{\max}]$, $[\sigma_{\min}, \sigma_{\max}]$, and $[\phi_{\min}, \phi_{\max}]$ for coverage, sample, and freshness. It also supplies the tenant identity, a critical, standard, or best-effort priority, an optional cost budget, per-request CPU and memory caps, and placement limits on node count and concurrent runtimes per node. These fields provide the complete information needed to validate the request, order it fairly, and determine whether its minimum operating point is feasible.

The request path carries the contract from its source to a placement decision. After validation, ARGOS initializes a target configuration within the accepted ranges, using their midpoints by default. The orchestrator then computes the minimum number of nodes required by the lower coverage bound and applies one of three transitions: immediate assignment when capacity exists, temporary queueing when the request remains feasible, or rejection when its placement limits make minimum coverage impossible. Every transition is written to the runtime audit log, which preserves the complete admission history for operational inspection and later validation.

The adaptation path repeats at the polling cadence and turns the latest runtime context into the next operating point. ARGOS collects node and service telemetry, updates request context, encodes the MDP state, and builds the admissible action set. The mask excludes actions that would have no effect because the corresponding target is already at a contract bound, as well as actions that would increase load when placement capacity is exhausted or the mean observed CPU or memory usage approaches the request’s cap. The active controller therefore selects only actions that can be applied, after which the environment updates the target configuration and the placement component distributes a new node plan when required. At a configurable polling cadence, the loop checks elapsed time before decision and distribution against an iteration budget. If the budget is exhausted, the next phase is skipped and the event is recorded. The audit log stores the state, action, target and realized configuration, reward components, algorithm metadata, and operational errors for the path summarized in Algorithm 1.

Policy training is a lifecycle activity. ARGOS can update a policy in training mode, serialize the resulting artifact, and later load it in deployment or evaluation mode with exploration and parameter updates disabled. The deployed architecture only requires the policy interface inside the per-request controller. The training budget, artifact freezing procedure, comparator selection, and provenance checks belong to the validation protocol and are specified in Section 4.3 and Section 5.

4.3. Decision model

Each per-request controller represents its request as an MDP over an aggregated operational state. The state at iteration t is encoded as

$$s_t = (\text{cpu}_t, \text{mem}_t, \text{bw}_t, c_t, \sigma_t, \phi_t, d_t, x_t, j_t, h_t, u_t, \ell_t, v_t),$$

where cpu_t , mem_t , and bw_t are observed resource-pressure buckets. The terms c_t , σ_t , and ϕ_t encode the current position of coverage, sample, and freshness relative to the accepted range of the request. The service-duty bucket d_t relates processing time to the requested refresh interval, while x_t encodes the offered input-load level. The remaining terms encode active jobs j_t , free capacity h_t , pending demand u_t , latency ℓ_t , and recent SLO violations v_t . The latter counts decision epochs with any contract or resource violation over a configurable rolling window of length W . The prototype fixes $W = 20$ to retain recent violations across several decisions without accumulating the full request history. This value is fixed a priori and is not

tuned during evaluation. Resource and runtime dimensions use non-uniform cuts chosen around operational pressure regions, from idle through warning to overload. Contract positions are canonicalized, normalized to $[0, 1]$, and divided at $\{0.2, 0.4, 0.6, 0.8\}$, which prevents two arithmetically equivalent operating points from entering different state buckets. The resulting 13-dimensional state is used directly by Q-learning and vectorized for DQN and PPO. The rolling count summarizes the recent history but does not expose the order of the W binary events to the agent. The model is therefore an operational MDP approximation over the telemetry available to the controller rather than a claim of complete observability.

Algorithm 1 ARGOS runtime orchestration loop.

Require: Polling interval Δ , iteration budget B , active requests $\mathcal{R}_t$ each with policy π_r , mode m_r , environment E_r , and target configuration θ_t^r , and node set $\mathcal{N}$.

```

1: loop
2:    $t_0 \leftarrow \text{NOW}$ 
3:    $\mathbf{m} \leftarrow \text{POLLMETRICS}(\mathcal{N})$ 
4:   if  $\text{NOW} - t_0 < B$  then
5:      $\mathcal{P} \leftarrow \emptyset$  ▷ node plans
6:     for all  $r \in \mathcal{R}_t$  do
7:        $s_t^r \leftarrow E_r.\text{ENCODESTATE}(\mathbf{m}, \text{ctx}_r)$ 
8:        $\mathcal{A}_t^r \leftarrow E_r.\text{AVAILABLEACTIONS}$ 
9:        $a_t^r \leftarrow \pi_r.\text{SELECTACTION}(s_t^r, \mathcal{A}_t^r)$ 
10:       $(s_{t+1}^r, \theta_{t+1}^r, \rho_t^r, \text{comp}_t^r) \leftarrow E_r.\text{APPLY}(a_t^r)$ 
11:      if  $m_r = \text{TRAIN}$  then
12:         $\pi_r.\text{UPDATE}(s_t^r, a_t^r, \rho_t^r, s_{t+1}^r)$ 
13:      end if
14:       $\text{PERSIST}(r, s_t^r, a_t^r, \theta_{t+1}^r, \rho_t^r, \text{comp}_t^r)$ 
15:      if  $\theta_{t+1}^r \neq \theta_t^r$  then
16:         $\mathcal{P} \leftarrow \text{UPDATEPLANS}(\mathcal{P}, r, \theta_{t+1}^r)$ 
17:      end if
18:    end for
19:    if  $\mathcal{P} \neq \emptyset$  then
20:      if  $\text{NOW} - t_0 < B$  then
21:         $\text{DISTRIBUTE}(\mathcal{P}, \mathcal{N})$ 
22:      else
23:         $\text{RECORDBUDGETEVENT}$ 
24:      end if
25:    end if
26:  else
27:     $\text{RECORDBUDGETEVENT}$ 
28:  end if
29:   $t_1 \leftarrow \text{NOW}$ 
30:   $\text{SLEEP}(\max(0, \Delta - (t_1 - t_0)))$ 
31: end loop

```

The action space contains seven actions:

$$\mathcal{A} = \{\text{hold}, \text{inc}_c, \text{dec}_c, \text{inc}_\sigma, \text{dec}_\sigma, \text{inc}_\phi, \text{dec}_\phi\}.$$

Coverage and sample actions move the corresponding target value by 0.05. Freshness actions change the target interval by 10 seconds, with dec_ϕ producing fresher updates and inc_ϕ relaxing the requirement. Before selecting an action, ARGOS masks actions that would have no effect because the corresponding target is already at a contract bound, together with actions that would increase load when placement capacity is exhausted or the mean observed CPU or memory usage approaches the request’s cap. A final clamp remains as a defensive check, so no target selected by an applied action can leave the accepted range.

Each action is scored by a reward that is a clipped linear score over quality and penalties:

$$r_t = \text{clip}_{[-1,1]} \left(q_t - p_t^{\text{resource}} - p_t^{\text{cost}} - p_t^{\text{range}} - p_t^{\text{overload}} - p_t^{\text{fairness}} - p_t^{\text{latency}} - p_t^{\text{capacity}} \right).$$

The quality term q_t is the mean of three normalized scores: coverage and sample are better near the upper end of the accepted range, while freshness is better near the lower end because lower intervals produce more recent analytics. Seven penalties price the operational state of the request. The resource term combines observed CPU, memory, and bandwidth pressure with normalized service duty, using the larger available signal and falling back to requirement-induced pressure when neither is available. Cost follows the estimate induced by coverage, sample, and freshness, while the range penalty independently checks coverage, sample, and freshness against their accepted ranges and penalizes any out-of-range target value. Overload activates when observed CPU or memory exceeds its hard cap, fairness reflects tenant debt, latency captures response-time excess, and capacity captures waiting demand when no placement remains. Quality and cost are evaluated from the updated target configuration, whereas resource penalties use the latest telemetry collected before action selection.

Three fixed weights anchor the aggregator: quality carries 1.0, resource pressure carries 1.2, and cost carries 0.5. Resource and cost are normalized into a shared unit penalty budget whose maximum combined contribution is 1.0, with the larger resource coefficient giving resource pressure more influence than cost within that blend. Overload is computed separately for CPU and memory, with each resource contributing at most 0.5 and their joint maximum therefore equal to 1.0. The remaining maximum contributions are 0.5 for the three range checks together, 0.35 for latency, 0.30 for capacity, and 0.20 for fairness. These limits place the unclipped score between -3.35 and 1.0 before clipping it to $[-1, 1]$, while persistence of every component keeps the original composition auditable.

Three learned controllers are instantiated against the same MDP. The **Q-learning** controller stores values by encoded state and action index, initializes every unvisited action optimistically at 3.0, and resolves ties through a private seeded generator rather than favoring the first action. Its learning rate is 0.1, its discount factor is 0.95, and epsilon decreases from 0.15 to 0.01 over the configured training horizon. The **DQN** controller replaces the table with a neural value approximator of hidden size 64 and combines a replay buffer of 10,000 transitions, a 32-transition warm-up, the Adam optimizer, a smooth L1 Bellman loss, and a target network synchronized every 25 steps. The **PPO** controller uses an actor-critic of the same hidden size, collects 32-step rollouts, estimates generalized advantages with $\lambda = 0.95$, and performs four clipped-update epochs with minibatches of 16. Its clipping ratio is 0.2, value weight is 0.5, and entropy weight is 0.01 [21]. All controllers use private seeded generators, while PPO samples from its masked categorical policy during training and uses masked argmax during frozen evaluation.

The comparison uses two untuned deployable baselines and one tuned static reference. **Static midpoint** holds the midpoint configuration through the same per-request selector and environment as the learned agents, whereas **Threshold** reacts whenever CPU, memory, or service duty reaches its warning region, demand queues, or the recent SLO window contains a violation. Under pressure, Threshold cycles through sample reduction, coverage reduction, and freshness relaxation, then applies the inverse quality improvements after pressure subsides. The **Best-fixed** reference holds one of nine predeclared configurations formed

by the eight contract vertices and their joint midpoint, selected independently for each workload profile through tuning seeds {101, 202, 303}. Training uses disjoint seeds {11, 22, 33, 44, 55}, while frozen evaluation uses held-out seeds {111, 222, 333, 444, 555}. Each learned controller trains for 2048 one-second decisions under the four-phase input-load schedule {2, 16, 64, 4}, with 16 decisions per phase and 32 complete cycles, before its serialized policy is loaded for 64 decisions under the distinct input-load schedule {3, 12, 48, 6} with exploration and updates disabled. This separation changes the random stream and offered-load trace between training and evaluation. The campaign records the SHA-256 hash and parameter fingerprint of every policy artifact, rejecting any run whose loaded fingerprint, iteration count, runtime, source revision, or provenance manifest does not match the declared protocol.

5. Experimental validation

This section validates ARGOS on the analytics service and decision model of Section 4 through a two-stage protocol. The controlled stage compares every frozen learned policy with the three non-learning comparators, the static midpoint, the reactive threshold, and the tuned *best-fixed* reference, under identical load schedules, and answers the model-ranking question. The live stage introduces realistic and saturated request arrivals, and answers the deployability and contract-safety question without conflating it with model ranking. Section 5.1 details the testbed, the workload profiles, and the campaign protocol that both stages share, Section 5.2 reports the controlled comparison, Section 5.3 analyzes the quality-against-pressure trade that the comparison exposes, and Section 5.4 reports the live trials.

5.1. Experimental setup

The reference deployment is a cluster of three heterogeneous nodes, summarized in Table 2. The Large node coordinates the cluster and also executes analytics, while the Medium and Small nodes provide worker capacity under progressively tighter resource limits. This asymmetry is intentional and represents a common Continuum deployment in which a more capable node coordinates smaller worker nodes [1, 2]. The validation pins the active orchestrator to the Large node, a role that Section 4.1 shows is otherwise mobile. The cluster runs on virtualized cloud resources, which supports reproduction, while the node profiles emulate continuum heterogeneity from 1 to 16 virtual CPUs (vCPUs). The service is instantiated with a Seville urban-mobility dataset of 60 per-user movement trajectories generated with the Opportunistic Network Environment mobility simulator [33], together containing 128,103 timestamped location points, representative of smart-city mobility analytics [17]. Each controlled campaign fixes one runtime mode, and the comparison is repeated on the thread and process runtimes, so that each controlled comparison is repeated under both concurrency models.

Node	Role	vCPUs	Memory (GB)
Large	Orchestrator and worker	16	32
Medium	Worker	2	4
Small	Worker	1	2

Table 2: Testbed nodes of the reference deployment.

The controlled stage exercises every controller on five recurrent workload profiles, defined in Table 3, that span the three negotiable requirements at once and instantiate the operating regimes of Section 2. *Lax-background* combines low coverage and sample with a long refresh interval for low-demand background

analytics. *Short-burst* combines a high sample fraction with a shorter refresh interval for event monitoring. *Aggressive-incident* pushes coverage and sample high while holding freshness to a few seconds, the regime in which a stale snapshot would hide the forming jam from the emergency operator. *Standard-operations* captures balanced day-to-day demand, and *Cost-sensitive* narrows coverage and sample to respect a tight cost budget. The set covers both low-pressure and saturation regimes and defines a controlled operating envelope for comparing learned and non-learning policies rather than an exhaustive contract catalog.

Profile	Coverage	Sample	Freshness (s)
Lax-background	[0.25, 0.45]	[0.20, 0.45]	[90, 180]
Short-burst	[0.60, 0.95]	[0.60, 0.90]	[10, 45]
Aggressive-incident	[0.75, 1.00]	[0.70, 1.00]	[5, 30]
Standard-operations	[0.45, 0.75]	[0.35, 0.65]	[30, 90]
Cost-sensitive	[0.33, 0.67]	[0.25, 0.55]	[60, 150]

Table 3: Canonical workload profiles.

Each campaign cell couples one workload profile, one runtime, and one controller. Learned controllers train for 2048 decisions at the one-second cadence under the input-load schedule {2, 16, 64, 4}, with 16 decisions per phase and 32 complete cycles. Their policies are serialized with a SHA-256 hash and a parameter fingerprint. Each artifact is then evaluated for 64 decisions with exploration and parameter updates disabled under the held-out schedule {3, 12, 48, 6}. The *best-fixed* reference is selected from the eight contract vertices and the joint midpoint using the tuning seeds {101, 202, 303}. Training uses {11, 22, 33, 44, 55} and final evaluation uses {111, 222, 333, 444, 555}, so training, tuning, and evaluation have disjoint random streams.

A session enters the campaign index only when it completes its declared decisions and passes the provenance checks of Section 4.3. Failed attempts are isolated and retained as excluded entries with an explicit reason. Per runtime, the accepted campaign comprises 75 training runs, three learners over five profiles and five seeds, 135 tuning runs, nine candidates over five profiles and three tuning seeds, and 150 frozen evaluations, six controllers over five profiles and five held-out seeds. The two runtimes together contribute 720 accepted sessions, and the 300 frozen evaluations score 19,200 decisions in total.

The analysis rests on three measurements. The per-step reward of each evaluation is recomputed from the persisted reward components, so every aggregate can be audited offline. The paired delta compares a learned controller with each non-learning comparator on the same runtime, profile, and held-out seed. Each seed contributes the mean of its five profile-level deltas. The analysis reports the paired mean and median, a two-sided 95 % Student-t interval over the five seed means, the rank-biserial effect size, and win, tie, and loss counts. With only five paired seed means, the exact two-sided Wilcoxon signed-rank test cannot yield a p-value below 0.0625, so the analysis emphasizes effect direction, effect size, seed win counts, and confidence intervals rather than confirmatory significance testing. In the live stage, contract and resource violations are recorded at the decision-epoch level for admitted requests, while queued-only requests are reported separately and do not count as frozen-policy evaluations. Requests admitted too close to the end of a trial to complete one frozen decision epoch are likewise reported separately as tail-censored, so a measurement boundary is never counted as a policy failure.

ARGOS is implemented in Python, with the orchestrator and worker-node interfaces exposed through FastAPI. The distributed profile uses the Neural Autonomic Transport System (NATS), a lightweight open-source messaging system, to exchange coordination events for capacity-aware leader election, while the local profile uses in-process coordination. The framework configuration, including the workload profiles of

Table 3 and the reward weights of Section 4.3, is loaded from YAML at startup. The DQN and PPO controllers use the PyTorch backend, and every run records its active backend so that any fallback is excluded from the comparison. Decision, telemetry, lifecycle, and violation streams are stored as append-only JSON Lines (JSONL) files. Given a fixed set of accepted evidence packages, the analysis pipeline regenerates every aggregate table and figure deterministically, although the physical telemetry underneath is observed rather than assumed to reproduce bit for bit across executions. To respect space limitations, the paper reports the tables and figures most directly supporting the controlled comparison, resource-pressure analysis, and live validation. The complete result tables, additional figures, and scripts used to regenerate them are available in the public ARGOS artefact [32].

5.2. Controlled comparison

This subsection answers the model-ranking question. Table 4 reports, for every learned controller, its paired reward-per-step advantage over each non-learning comparator on both runtimes. Each cell gives the mean delta, with positive values favoring the learned controller, the number of evaluation seeds that favor it out of five in parentheses, and the fraction of the *best-fixed* reward that it recovers. The 95 % Student-t interval of every cell excludes zero except for Q-learning against the static midpoint. Each seed contributes one profile-averaged delta, so the five evaluation seeds define the paired sample, and the sign is read as the learned controller minus the comparator. The picture is consistent across thread and process. The two deep controllers, DQN and PPO, improve on both untuned baselines in every seed, the tabular Q-learning controller does not improve on the static midpoint, and all three learned controllers stay below the independently tuned *best-fixed* reference.

Runtime	Controller	vs. Static	vs. Threshold	vs. Best-fixed	Best-fixed recovered
Thread	DQN	+0.176 (5/5)	+0.392 (5/5)	−0.129 (0/5)	76 %
	PPO	+0.139 (5/5)	+0.355 (5/5)	−0.166 (0/5)	68 %
	Q-learning	−0.045 (1/5)	+0.171 (5/5)	−0.350 (0/5)	34 %
Process	DQN	+0.179 (5/5)	+0.425 (5/5)	−0.142 (0/5)	72 %
	PPO	+0.151 (5/5)	+0.398 (5/5)	−0.170 (0/5)	67 %
	Q-learning	−0.043 (2/5)	+0.203 (5/5)	−0.364 (0/5)	29 %

Table 4: Paired reward advantage of each learned controller over each comparator.

DQN and PPO improve on both deployable baselines on both runtimes, and the advantage holds in all five seeds of every cell. Figure 3 shows the same comparison as mean frozen reward with held-out-seed intervals. Averaged over the five profiles, DQN and PPO reach a reward per step of 0.398 and 0.361 on thread (Figure 3a) and 0.370 and 0.342 on process (Figure 3b), against 0.222 and 0.191 for the static midpoint and 0.006 and −0.055 for the reactive threshold. The paired gain over the static midpoint ranges from +0.139 to +0.179 reward per step, and the gain over the threshold from +0.355 to +0.425, with rank-biserial effect sizes of +1.0 in every one of these cells. DQN holds a small edge over PPO in mean reward on both runtimes, and the two deep controllers are better described as a tier than as a strict order.

Figure 4 decomposes the comparison across the five workload profiles, for the thread runtime in Figure 4a and for the process runtime in Figure 4b. The deep controllers stay above the deployable baselines in every profile on both runtimes, while neither dominates the other everywhere, which is consistent with reading the two as one tier. The decomposition also explains the weak aggregate of the reactive threshold, whose reward collapses to negative values under the tight short-burst and aggressive-incident regimes,

where cycling quality downward against a demanding contract is costly, and never clears the static midpoint elsewhere. The *best-fixed* reference stays strongest in every profile, so the gap in Table 4 is not an artifact of one workload.

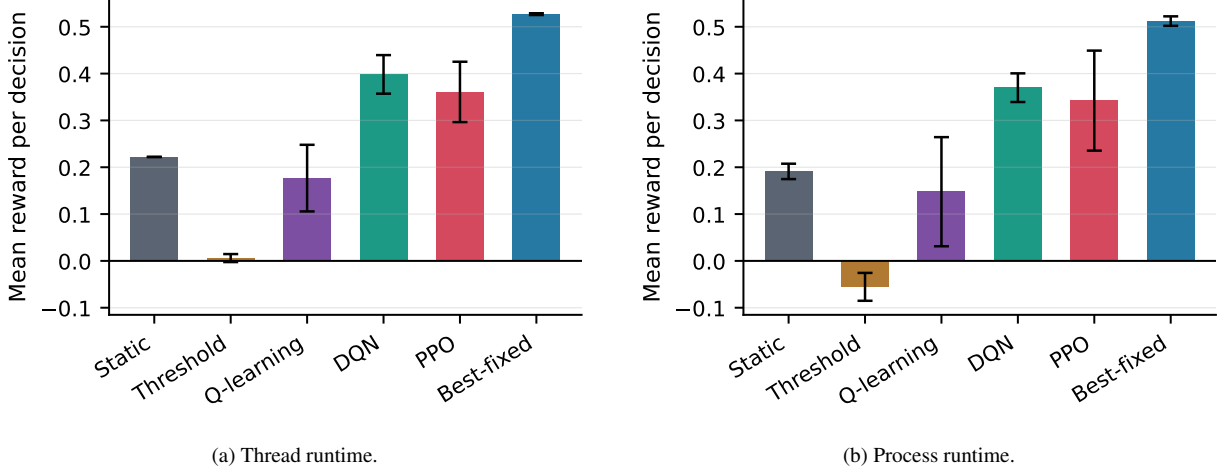


Figure 3: Mean frozen reward per decision for each controller.

The tabular Q-learning controller is not a deployable improvement. It fails to improve on the static midpoint on both runtimes, with paired deltas of -0.045 and -0.043 reward per step, only one and two favorable seeds, and intervals that include zero. Its advantage over the threshold, $+0.171$ and $+0.203$, is measured against a comparator that itself falls below the static midpoint. This result is consistent with the difficulty of covering the thirteen-dimensional discrete state of Section 4.3 within the available training budget. Q-learning therefore serves as a lightweight tabular reference rather than a competitive deployment candidate.

Every learned controller stays below the *best-fixed* reference, by 0.129 to 0.170 reward per step for the two deep controllers and by roughly 0.35 for Q-learning. This gap measures headroom against a comparator that is tuned per workload profile in advance and then held fixed throughout held-out evaluation. That the deep controllers recover roughly two thirds to three quarters of its reward while making online per-request control decisions is the main result of the controlled stage.

5.3. Resource-pressure analysis

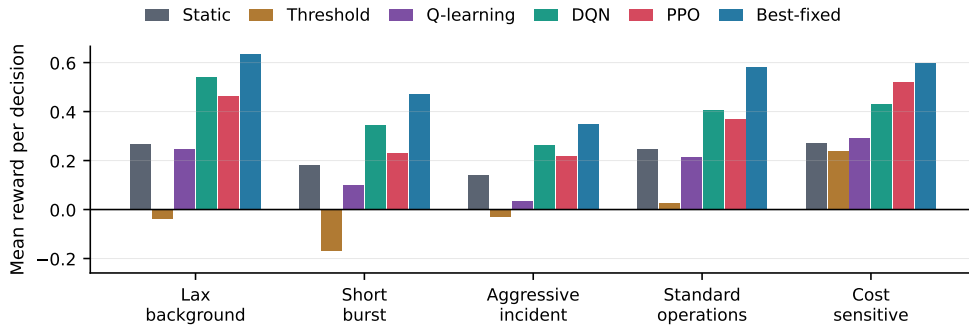
The controlled campaign also records how each controller loads the constrained node, which exposes the quality-against-pressure trade that motivates adaptive control. Spatial fidelity is $1 - \frac{1}{2} \sum_i |p_i - \hat{p}_i|$, where p_i and $\hat{p}_i$ are normalized cell frequencies in the sampled heatmap and its full-sample reference for the same trajectory partition. Table 5 pairs this measure with the mean service duty driven on the assigned nodes, the fraction of each refresh interval spent computing, and the 95th-percentile cluster CPU utilization. Every entry averages the five profiles and five seeds of its runtime. The deep controllers lift fidelity into a 0.931 to 0.940 band, above the 0.909 to 0.913 of the deployable baselines and within 0.02 of the 0.950 reference. On thread, PPO reaches a fidelity of 0.931 at a duty of 10.9 % against 16.8 % for the *best-fixed* reference, so it recovers most of the quality at roughly two thirds of the reference’s service pressure.

The extremes of Table 5 show the trade-off. The static and threshold baselines keep duty minimal, below 7 %, but settle at a fidelity of 0.909 to 0.913, and Q-learning improves only modestly on that band. The *best-fixed* reference reaches the highest fidelity by sustaining the highest service pressure on thread. The

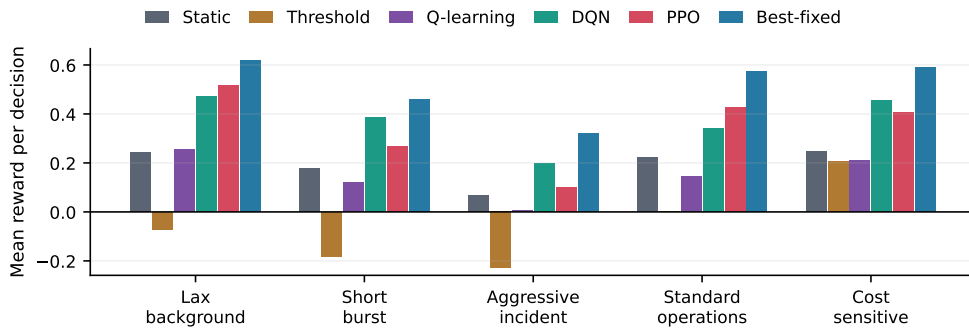
deep controllers sit between these ends and closer to the quality of the reference than to the baselines. PPO is the most efficient on this axis, holding the lowest duty of the deep controllers on both runtimes, whereas DQN reaches its small reward advantage at a higher duty, exceeding that of the reference on process. The 95th-percentile cluster CPU follows the same pattern: PPO stays below the reference on both runtimes, whereas DQN reaches the highest utilization of all controllers on process. This pressure reading separates the two deep controllers that the reward ranking leaves nearly tied, and the next subsection tests whether that separation holds when requests arrive and compete under live traffic.

Controller	Thread			Process		
	Fidelity	Duty	CPU p95	Fidelity	Duty	CPU p95
Best-fixed	0.950	16.8 %	89 %	0.950	16.2 %	66 %
DQN	0.940	13.8 %	87 %	0.939	18.1 %	76 %
PPO	0.931	10.9 %	81 %	0.933	12.7 %	59 %
Q-learning	0.919	12.3 %	80 %	0.917	8.5 %	73 %
Threshold	0.913	4.9 %	58 %	0.910	3.7 %	51 %
Static	0.909	6.3 %	70 %	0.909	5.6 %	54 %

Table 5: Spatial fidelity, service duty, and CPU utilization per controller; p95 denotes the 95th percentile.



(a) Thread runtime.



(b) Process runtime.

Figure 4: Mean frozen reward per decision by workload profile and controller.

5.4. Live-trial validation

The live stage shifts the question from model ranking to deployability under load. Whereas the controlled stage drives one request through a fixed schedule, the live stage submits requests that arrive and depart over time under two arrival regimes: a realistic regime that keeps the cluster below saturation and a concurrency regime that pushes it into overload. Admitted requests that breach any contract or resource bound are reported as service-level objective violations. This separates operational safety, whether the deployed controller keeps the analytics inside the client ranges while the cluster is contended, from the reward ranking of the controlled stage.

The live campaign pairs the static midpoint with each deep controller, DQN and PPO, on the thread runtime. Tabular Q-learning is not carried into this stage because it does not improve on the static midpoint in the controlled comparison, so it is not a deployment candidate, and the live stage evaluates the controllers a deployment would actually consider. The controlled stage already covers both runtimes, so the live stage fixes one and spends its budget on paired repetitions instead.

Each arrival regime is repeated over five live seeds, and each seed drives one static, one DQN, and one PPO trial of 30 minutes with an identical arrival trace, so every comparison is paired at the trace level. Each learned trial loads the frozen artifact trained under the matching training seed and applies it to every admitted request without exploration or parameter updates, while admission, queueing, and placement remain active in every variant. The 30 accepted trials cover 58,285 decision epochs and 369 admitted requests that completed at least one frozen decision epoch. Two further requests were admitted too late in their trial to complete one and are reported as tail-censored. One trial attempt was excluded and repeated after a shutdown-window artifact, and the excluded attempt is retained in the campaign index with its reason.

Figure 5 shows the offered load, averaged over the five paired seeds of each variant. Where the three curves coincide, the thinner DQN and PPO lines are drawn over the wider static line. The realistic regime (Figure 5a) stays below saturation: all 51 requests submitted across its five trials per variant are admitted on arrival, the active set plateaus around four requests, and the queue stays empty in every variant. The concurrency regime (Figure 5b) overloads the cluster by design: of the 180 requests submitted per variant, about 40 are admitted on arrival while 140 wait in the pending queue, which grows toward 28 requests as the active set saturates at the placement capacity of roughly eight. Admission behaves almost identically under all three controllers, with overlapping active and queued trajectories, so the arrival regime rather than the quality controller determines what enters the cluster, and the paired comparison isolates what each controller does with the admitted work.

Figure 6 reports the paired outcome. On the realistic regime (Figure 6a) both frozen controllers improve the mean decision reward over the static midpoint in every pair: DQN by +0.261 reward per step, with a 95 % interval of [0.181, 0.341], and PPO by +0.173, with an interval of [0.128, 0.218]. Under overload (Figure 6b), where the static midpoint averages -0.034 , PPO stays positive at 0.069 and improves in all five pairs with an interval of [0.046, 0.161], while DQN reaches 0.085 but improves in four of five pairs with an interval of $[-0.049, 0.287]$ that includes zero. Delivered quality moves the same way: spatial fidelity rises under PPO in all ten pairs, +0.028 and +0.025, and under DQN by +0.035 on the realistic regime in every pair, against +0.018 with a zero-crossing interval under overload. The frozen deep tier therefore transfers to live multi-tenant traffic, with DQN posting the larger mean gains and PPO the uniformly consistent ones under saturation, the same reliability pattern that Section 5.3 anticipates.

Contract exposure completes the safety reading. Table 6 reports it over the five paired seeds, with admitted requests that completed at least one frozen decision epoch, the mean fraction of decision epochs carrying any violation, and violation counts by dimension. Across the 58,285 decision epochs of the 30

trials, the recorded violation events are almost exclusively coverage events, 27,130 of 27,145, with zero sample, CPU, and memory events and 15 freshness events. No sample, CPU, or memory violations were recorded, and freshness violations were rare.

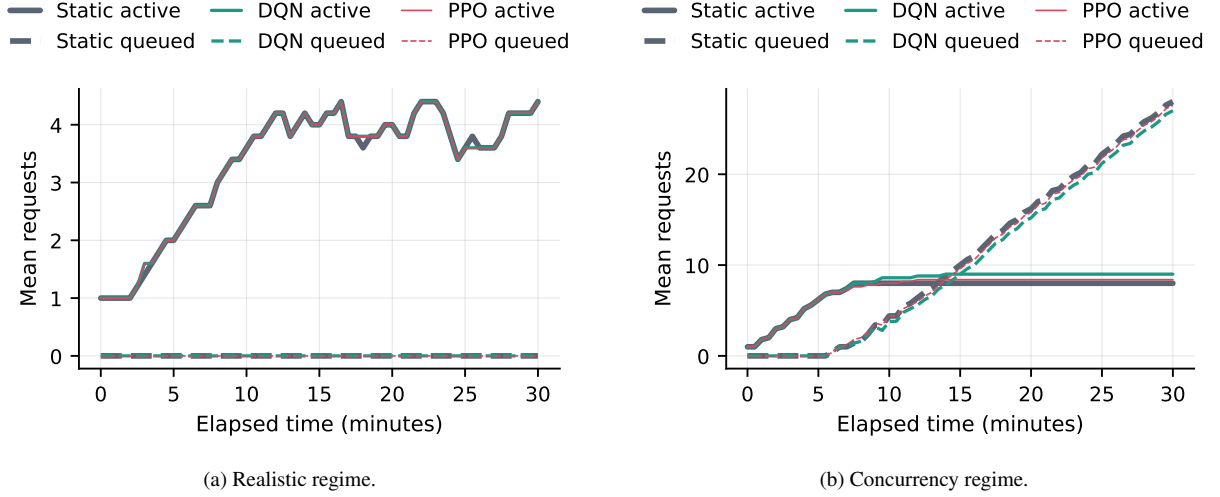


Figure 5: Mean active and queued requests over time in the live stage.

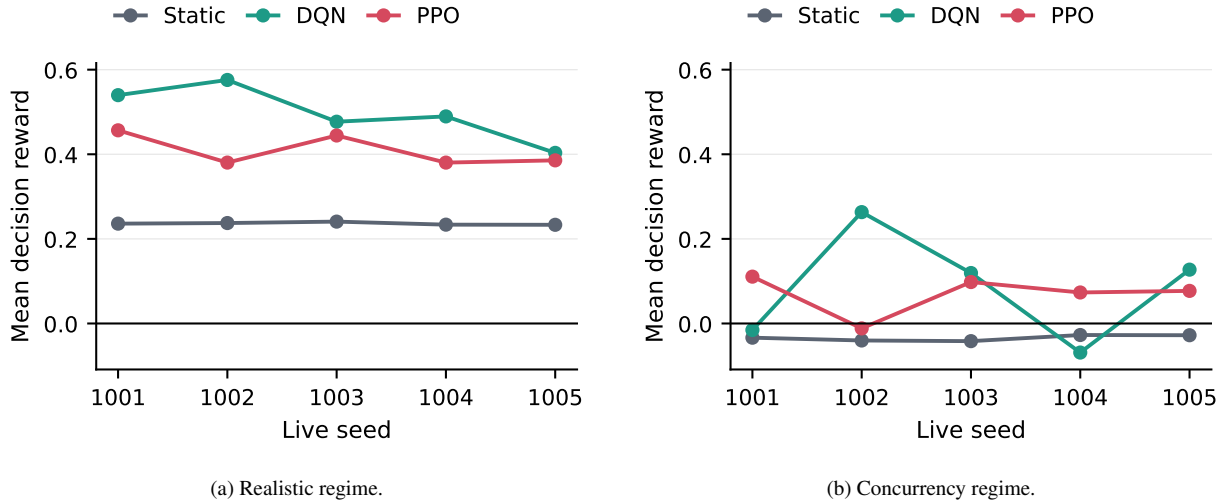


Figure 6: Paired mean decision reward per live seed.

Coverage is realized in thirds on the three-node testbed, so ceiling-based placement can raise the realized value above the maximum accepted by contracts whose upper bound lies between two reachable coverage levels. All 27,130 recorded coverage violations are upper-bound violations; no lower-bound coverage violation is observed. In the realistic regime, the mean fraction of violated decision epochs is 0.39 for the static midpoint, 0.61 for PPO, and 0.71 for DQN, with smaller differences under overload. The live trials therefore show that reward and fidelity gains can coexist with coverage violations caused by placement granularity, despite the absence of recorded CPU or memory violations. This supports operational feasibility on the evaluated testbed, not a guarantee of full contract compliance.

Regime	Variant	Evaluated	Violated epochs	Coverage viol.	Freshness viol.	Sample/CPU/mem. viol.
Realistic	Static	51	39.0 %	2335	0	0
	DQN	50	71.1 %	4246	0	0
	PPO	50	60.5 %	3684	1	0
Concurrency	Static	70	38.2 %	4924	0	0
	DQN	75	40.1 %	5525	2	0
	PPO	73	50.7 %	6416	12	0

Note: viol. = violation; mem. = memory.

Table 6: Live contract exposure per regime and variant.

6. Discussion

This section interprets the experimental evidence in relation to the objectives established in the preceding sections and separates the conclusions supported by the results from the limitations that bound their scope. Section 6.1 discusses the controlled and live-stage findings and their operational implications, while Section 6.2 examines the internal, external, scalability, and construct limitations of the study together with the scope of its reproducibility claims.

6.1. Results discussion

The controlled evidence supports the claim that the two deep controllers improve on both deployable baselines in every seed of both runtimes. At the same time, tabular Q-learning does not improve on the static midpoint, and its advantage over the threshold is measured against a comparator that performs substantially worse than the static midpoint. DQN holds a small mean-reward edge over PPO, yet both improve on each deployable baseline in every seed, so the two are better described as a deep-controller tier than as a strict order. The controlled evidence therefore supports a narrow claim: the deep controllers outperform the deployable heuristics in this setting, whereas the tabular learner does not.

All three learned controllers stay below the *best-fixed* reference, and that gap is read as headroom against a strong static comparator rather than against an oracle. A deployment can use the selected point when its workload profile is known and stable, but that fixed configuration cannot react to changing node pressure during a request. The deep controllers recover roughly two thirds to three quarters of its reward, trailing it by about 0.13 to 0.17 reward per step, while retaining online per-request control. The resource-pressure analysis adds an operational reading: the deep controllers lift spatial fidelity above the deployable baselines without uniformly requiring the highest service pressure. PPO holds the lowest service duty of the deep controllers on both runtimes, while DQN exceeds the duty of the *best-fixed* reference only on process. These results show that adaptive quality control can use available node headroom to improve analytics quality without requiring the highest service duty in every setting, which is the behavior motivated in Section 2.

The live stage of Section 5.4 extends this reading to multi-tenant operation. Paired on identical arrival traces, both frozen deep controllers improve mean decision reward and delivered fidelity over the static midpoint. Under overload, both retain positive mean reward while the static midpoint has negative mean reward. They differ in consistency: PPO improves in every overload pair with an interval that excludes zero, whereas DQN improves in four of five pairs with a wider interval, which is consistent with PPO’s

lower service duty in the resource-pressure analysis. Contract exposure is dominated by upper-bound coverage violations caused by the coarse placement granularity of the three-node testbed. Freshness violations are rare, and no sample, CPU, or memory violations are recorded. The live evidence therefore supports deployability of the frozen deep controllers rather than providing a second model ranking, preserving the separation intended by the two-stage protocol.

For deployments resembling the evaluated setting, either deep controller is a reasonable candidate when reward per request matters. DQN provides the highest mean reward among the learned controllers, while PPO reaches a similar reward with the lowest mean service duty among the deep controllers. PPO also shows the more consistent overload behavior in the live trials, while DQN’s advantage varies more across seeds. Q-learning is better treated as a tabular reference than as a deployment candidate because it does not improve on the static midpoint on either runtime. The static midpoint remains a deterministic fallback when operational simplicity outweighs the adaptive gain. The *best-fixed* reference suits a known and stable workload profile when offline tuning is acceptable, but it remains a comparator rather than a general adaptive solution because its operating point cannot respond to changing resource pressure.

6.2. Limitations

Several factors bound how broadly the contribution transfers. On the internal side, the paired protocol shares the seed schedule, orchestration loop, persistence path, and comparison scripts across controllers, which reduces implementation-level confounds. The main statistical limitation is the number of paired evaluation seeds. Each comparison rests on five seed means, so the exact two-sided Wilcoxon signed-rank test cannot yield a p-value below 0.0625. The Student-t intervals assume approximately normal seed-level mean differences, an assumption that is difficult to assess with only five seeds. The analysis therefore relies on effect direction, effect size, seed win counts, and confidence intervals rather than on a confirmatory significance threshold. On the external side, the workload consists of one urban-mobility heatmap service over simulated Seville movement trajectories, and the five workload profiles cover selected operating regimes without exhausting the contract space. The controlled comparison covers both thread and process runtimes, whereas the live stage uses the thread runtime, so its deployability evidence does not cover every runtime or application setting.

Scalability beyond the evaluated testbed remains an open question. The controlled and live stages use three heterogeneous nodes, which is sufficient to exercise heterogeneous placement, resource contention, admission, and per-request adaptation, but does not characterize how ARGOS behaves as the number of nodes and concurrent requests grows. Larger deployments may change placement granularity, telemetry and coordination overhead, and the interaction between admission and per-request control. The distributed leader-failover mechanism is also outside the present evaluation, which keeps one active orchestrator to isolate the quality-control behavior. Evaluating these aspects on larger Continuum topologies is therefore an important direction for future work.

On the construct side, the reward function uses fixed coefficients of 1.0 for quality, 1.2 for resource pressure, and 0.5 for cost, with resource and cost normalized into a shared penalty budget. These coefficients are fixed design parameters rather than the result of per-deployment optimization. The measurement conventions introduce additional limits. Quality and cost reflect the selected configuration, while resource telemetry comes from the preceding poll and may not yet reflect that configuration’s resource effects. This limits causal interpretation of an individual action’s immediate resource effect. Spatial fidelity measures sampling distortion within the evaluated partitions, not city-wide coverage or result freshness. The 20-epoch violation count is also a bounded-history summary. It captures recent violation frequency but not the exact order or age of individual events, so the decision model approximates Markov observability from the

telemetry available to the controller. Finally, the action space is discrete, which improves auditability but leaves continuous control unexplored.

These limitations should be considered together with the evaluation protocol. ARGOS is exercised on a deployed heterogeneous cluster using the resource telemetry produced under actual contention. Reproducibility studies have shown that single-seed evaluations, uncontrolled hyperparameter choices, and insufficient separation between training and evaluation can make deep reinforcement-learning results fragile [12, 14]. The orchestration studies discussed in Section 3 vary in how fully they address these concerns. Evaluating the controllers under capacity pressure and scoring frozen policies with held-out seeds and checked policy fingerprints therefore strengthens the deployment relevance and auditability of the reported comparison [12, 16].

7. Conclusions and future work

Computing Continuum analytics need more than resource scaling when heterogeneous nodes become capacity-limited. Static quality policies cannot react to cluster pressure, while placement or replica controllers do not act on the analytics requirements that define each contract. ARGOS addresses this problem by operationalizing multidimensional elasticity as a per-request MDP that adjusts coverage, sample, and freshness within the ranges accepted by each client and combines those decisions with capacity-aware admission. Every controller, learned or not, shares the same orchestration loop and persistence path. Each learned policy is frozen and fingerprinted before evaluation, so the reported gains are derived from auditable artifacts evaluated over multiple held-out seeds.

Across two runtimes, five workload profiles, and five held-out evaluation seeds, the frozen DQN and PPO controllers improve on both untuned deployable baselines in every seed of every controlled comparison. They recover roughly two thirds to three quarters of the reward of the independently tuned *best-fixed* reference while delivering near-reference fidelity at a lower mean service duty in every case but one. Tabular Q-learning does not improve on the static midpoint on either runtime, so the evidence supports the deep controllers rather than reinforcement learning in general. With only five paired seed means, the analysis emphasizes effect direction, effect size, seed win counts, and confidence intervals rather than confirmatory significance testing. The live stage of Section 5.4 complements the controlled comparison under time-varying arrivals. Both frozen deep controllers improve mean reward and delivered fidelity over the static midpoint, while PPO improves in every paired trial of both arrival regimes. No CPU or memory violations are recorded, although upper-bound coverage violations remain because of the three-node placement granularity. The controlled and live stages therefore support distinct claims about comparative performance and deployability.

Future work should extend ARGOS in three directions. First, continuous-action approaches should be evaluated to determine whether finer control can improve on the fixed coverage, sample, and freshness step sizes used here. Second, evaluation should include more seeds and additional deployment configurations to increase statistical power, particularly for differences within the deep-controller tier and under live traffic. Third, larger Continuum topologies and additional analytics services should be studied to determine how multidimensional quality control interacts with placement and whether the observed behavior generalizes beyond the three-node urban-mobility setting.

Acknowledgment

This work has been partially funded by the grant PTQ2024-013825 funded by MICIU/AEI/10.13039/501100011033, and co-financed 85% by the European Union, the European Regional Development Fund and

by the Department of Education, Science and Professional Training of the Regional Government of Extremadura. Managing Authority: Ministry of Finance. Grant File Number: GR24099.

References

- [1] F. Bonomi, R. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: N. Bessis, C. Dobre (Eds.), *Big Data and Internet of Things: A Roadmap for Smart Environments*, Vol. 546 of *Studies in Computational Intelligence*, Springer, 2014, pp. 169–186. doi:10.1007/978-3-319-05029-4_7.
- [2] M. Satyanarayanan, The emergence of edge computing, *Computer* 50 (1) (2017) 30–39. doi:10.1109/MC.2017.9.
- [3] V. C. Pujol, P. K. Donta, A. Morichetta, I. Murturi, S. Dustdar, Edge intelligence: Research opportunities for distributed computing continuum systems, *IEEE Internet Computing* 27 (4) (2023) 53–74. doi:10.1109/MIC.2023.3284693.
- [4] S. Dustdar, Y. Guo, B. Satzger, H.-L. Truong, Principles of elastic processes, *IEEE Internet Computing* 15 (5) (2011) 66–71. doi:10.1109/MIC.2011.121.
- [5] S. Laso, I. Murturi, P. Frangoudis, J. L. Herrera, J. M. Murillo, S. Dustdar, A multidimensional elasticity framework for adaptive data analytics management in the computing continuum, in: *Proceedings of the 2025 IEEE International Conference on Communications (ICC)*, IEEE, 2025, pp. 1133–1138. doi:10.1109/ICC52391.2025.11161512.
- [6] S. Laso, J. Berrocal, P. Fernández, J. M. García, J. García-Alonso, J. M. Murillo, A. Ruiz-Cortés, S. Dustdar, Elastic data analytics for the cloud-to-things continuum, *IEEE Internet Computing* 26 (6) (2022) 42–49. doi:10.1109/MIC.2021.3138153.
- [7] S. Laso, J. Berrocal, P. Fernández, A. Ruiz-Cortés, J. M. Murillo, S. Dustdar, Evaluating data analytics architectures performance: a comparative study on the continuum, *Journal of Cloud Computing* (2026). doi:10.1186/s13677-026-00960-z.
- [8] T. Llorido-Botran, J. Miguel-Alonso, J. A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, *Journal of Grid Computing* 12 (4) (2014) 559–592. doi:10.1007/s10723-014-9314-7.
- [9] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource management with deep reinforcement learning, in: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*, 2016, pp. 50–56. doi:10.1145/3005745.3005750.
- [10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, M. Alizadeh, Learning scheduling algorithms for data processing clusters, in: *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019, pp. 270–288. doi:10.1145/3341302.3342080.
- [11] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, C. G. Garino, Reinforcement learning-based application autoscaling in the cloud: A survey, *Engineering Applications of Artificial Intelligence* 102 (2021) 104288. doi:10.1016/j.engappai.2021.104288.

- [12] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, D. Meger, Deep reinforcement learning that matters, in: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018, pp. 3207–3214. doi:10.1609/aaai.v32i1.11694.
- [13] F. Rossi, V. Cardellini, F. L. Presti, M. Nardelli, Dynamic multi-metric thresholds for scaling applications using reinforcement learning, *IEEE Transactions on Cloud Computing* 11 (2) (2023) 1807–1821. doi:10.1109/TCC.2022.3163357.
- [14] C. Colas, O. Sigaud, P.-Y. Oudeyer, How many random seeds? statistical power analysis in deep reinforcement learning experiments (2018). arXiv:1806.08295, doi:10.48550/arXiv.1806.08295.
- [15] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, M. G. Bellemare, Deep reinforcement learning at the edge of the statistical precipice, in: *Advances in Neural Information Processing Systems 34 (NeurIPS)*, 2021, pp. 29304–29320.
URL <https://proceedings.neurips.cc/paper/2021/hash/f514cec81cb148559cf475e7426eed5e-Abstract.html>
- [16] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d’Alché Buc, E. Fox, H. Larochelle, Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program), *Journal of Machine Learning Research* 22 (164) (2021) 1–20.
URL <https://www.jmlr.org/papers/v22/20-303.html>
- [17] M. E. López-Pérez, M. E. Reyes-García, M. E. López-Sanz, Smart mobility and smart climate: An illustrative case in Seville, Spain, *International Journal of Environmental Research and Public Health* 20 (2) (2023) 1404. doi:10.3390/ijerph20021404.
- [18] C. J. C. H. Watkins, P. Dayan, Technical note: Q-learning, *Machine Learning* 8 (3–4) (1992) 279–292. doi:10.1007/BF00992698.
- [19] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, MIT Press, Cambridge, MA, 2018.
- [20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, *Nature* 518 (7540) (2015) 529–533. doi:10.1038/nature14236.
- [21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms (2017). arXiv:1707.06347, doi:10.48550/arXiv.1707.06347.
- [22] E. Casalicchio, A study on performance measures for auto-scaling CPU-intensive containerized applications, *Cluster Computing* 22 (3) (2019) 995–1006. doi:10.1007/s10586-018-02890-1.
- [23] F. Rossi, M. Nardelli, V. Cardellini, Horizontal and vertical scaling of container-based applications using reinforcement learning, in: *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338. doi:10.1109/CLOUD.2019.00061.
- [24] D. Trihinas, G. Pallis, M. D. Dikaiakos, AdaM: An adaptive monitoring framework for sampling and filtering on IoT devices, in: *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 717–726. doi:10.1109/BigData.2015.7363816.

- [25] A. A. Khaleq, I. Ra, Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud, in: Proceedings of the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), 2021, pp. 13–19. doi:10.1109/ACSOS-C52956.2021.00025.
- [26] I. Murturi, S. Dustdar, DECENT: A decentralized configurator for controlling elasticity in dynamic edge networks, *ACM Transactions on Internet Technology* 22 (3) (2022) 78:1–78:21. doi:10.1145/3530692.
- [27] G. Tesaro, N. K. Jong, R. Das, M. N. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, in: Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC), 2006, pp. 65–73. doi:10.1109/ICAC.2006.1662383.
- [28] Y. Garí, D. A. Monge, C. Mateos, A Q-learning approach for the autoscaling of scientific workflows in the cloud, *Future Generation Computer Systems* 127 (2022) 168–180. doi:10.1016/j.future.2021.09.007.
- [29] M. Goudarzi, M. Palaniswami, R. Buyya, A distributed deep reinforcement learning technique for application placement in edge and fog computing environments, *IEEE Transactions on Mobile Computing* 22 (5) (2023) 2491–2505. doi:10.1109/TMC.2021.3123165.
- [30] Z. Wang, M. Goudarzi, M. Gong, R. Buyya, Deep reinforcement learning-based scheduling for optimizing system load and response time in edge and fog computing environments, *Future Generation Computer Systems* 152 (2024) 55–69. doi:10.1016/j.future.2023.10.012.
- [31] A. Mampage, S. Karunasekera, R. Buyya, Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments, *Future Generation Computer Systems* 143 (2023) 277–292. doi:10.1016/j.future.2023.02.006.
- [32] J. Mateos-Bravo, S. Laso, J. L. Herrera, I. Murturi, P. Frangoudis, S. Dustdar, Argos: Reinforcement learning-driven multidimensional elasticity for service orchestration in the computing continuum (2026). doi:10.5281/zenodo.22933898.
URL <https://doi.org/10.5281/zenodo.22933898>
- [33] A. Keränen, J. Ott, T. Kärkkäinen, The ONE simulator for DTN protocol evaluation, in: Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools), 2009, pp. 1–10. doi:10.4108/ICST.SIMUTOOLS2009.5674.