Proactive Microservice Scaling with Storage Consistency in Geo-Distributed Cloud

Xin-Bao Wu, Li-Hsing Yen, and Yan-Wei Chen
Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu, Taiwan.

***Abstract*—As user demand becomes spatially dispersed, horizontal service scaling across geo-distributed servers is essential to reduce latency and meet SLAs. Reactive approaches often fail under uncertain demand and non-negligible rollout delays. We present the first joint formulation of proactive microservice placement, storage replica allocation, and VM provisioning that simultaneously accounts for VM rollout delays, write-broadcast consistency overhead, and CPU interference. These aspects were not jointly addressed in prior work. The resulting problem couples binary placement variables with prioritized feasibility requirements, making standard SAC inapplicable. Hence, we develop Hybrid Policy Gradient (HPG), an SAC extension with a Bernoulli relaxation for binary variables and a staged constraint-aware reward. Experiments show HPG improves SLA satisfaction over Reactive GA by 51%-165%, at a cost increase of at most 60.2% across all evaluated settings.**

***Keywords*—Microservice placement, Proactive resource scaling, Geo-distributed cloud, Storage replication**

I. Introduction

Service-Oriented Architecture (SOA) decomposes software systems into microservices (MSs), each responsible for a specific function [1]. An MS can invoke additional MSs, forming a chain of interactions to provide a specific service to users. MSs may also need storage space for general-purpose data operations, including querying, updating, and persisting domain entities.

A major task in building SOA is *MS placement*, i.e., placing MS instances on geo-distributed servers to serve geo-dispersed users. Placing too few instances conserves resources but increases service delivery time, while placing too many wastes resources and raises costs. Finding an optimal balance is challenging even for static placements.

When MS placement is dynamic, the issue is also about *when* to change placements. *Reactive* strategies change placements only after observing system states, which often cause late placements [2] during sudden demand surges and break Service Level Agreements (SLAs). *Proactive* approaches anticipate future demand and make decisions beforehand, accounting for service rollout delay, i.e., the lead time to deliver MS instances.

To efficiently support geo-distributed MS instances, *storage placement* (i.e., determining where and how many storage replicas to place) is crucial but has been overlooked. Read requests can be routed to the nearest replica, but write operations must be broadcast to all replicas to maintain consistency [3]. Unlike MS placement, storage placement is typically static due to the high overhead of migration.

Table I: Comparison of related works on service scaling and data management

Work	MS Placement	Service Delay	Service Cost	Request Dispat.	CPU Intf.	Data Consist.
[6]	Static	Yes	No	Deter.	No	No
[7]	Reactive	Yes	Payment	RR	When full	No
[4]	Reactive	Yes	No	Stochastic	No	No
[9]	Reactive	Yes	No	RR	No	No
[8]	Reactive	No	Payment	RR	No	No
[5]	Reactive	Yes	Energy	Stochastic	No	No
Ours	Proactive	Yes	Payment	Stochastic	Yes	Yes

An essential complementary task is request dispatching: routing MS invocation and storage read requests to appropriate targets, balancing queuing delay against propagation delay.

Existing studies have explored MS placement alongside stochastic request dispatching [4], [5], rather than deterministic [6] or round-robin [7], [8], [9] rules. However, most overlooked storage placement, limiting their capacity to manage data consistency. Some minimized resource usage [7], [8] but assumed instantaneous provisioning, ignoring rollout delays and making their approaches inherently reactive. To address these gaps, our main contributions are as follows:

- *Joint problem formulation.* To the best of our knowledge, we present the first joint formulation of proactive MS placement, storage replica allocation, and VM provisioning that simultaneously accounts for VM rollout delays, write-broadcast consistency overhead, and CPU interference among co-located MSs (see Table I).
- *HPG algorithm.* We develop HPG, a SAC-based RL algorithm adapted to handle the specific structure of our problem. Standard SAC cannot be directly applied because: (i) our placement variables are binary, ruling out Gaussian reparameterization while the action space is too large to enumerate; and (ii) a scalar reward carries no notion of constraint priority. HPG addresses these with a Bernoulli relaxation and a staged constraint-aware reward.
- *Empirical validation.* HPG outperforms the Reactive GA baseline by up to 165% in SLA satisfaction while limiting cost increases to 41%.

II. Background and Related Work

MS placement strategies can be static, reactive, and proactive. Static strategies deploy microservices based on long-term average demand or predefined policies [6], but lack flexibility to adapt to traffic variation. Reactive strategies

allocate resources only after observing heavy load or deteriorated performance, which can lead to under-provisioning and SLA violations during sudden surges. Some studies enhance reactive mechanisms by considering availability constraints or cost optimization [8], [7], [4]. Proactive approaches anticipate future demand and prepare resources in advance [10], but commonly neglect service rollout delays [2], which can still cause under-provisioning during the lead time.

The role of storage placement in SOA has been largely overlooked. Only Ferdaus et al. [11] distinguished between compute and data units, but did not consider data replication essential in geo-distributed systems. Without storage replicas, service scaling suffers from remote data access delays. With replicas, consistency overhead is typically ignored, limiting applicability to real-world deployments.

Table I summarizes the comparison of existing approaches, highlighting the gaps our work addresses.

III. System Model

We assume a service provider which rents a underlying computation and storage infrastructure from a cloud vendor to provide service to users. The cloud system consists of a set of geo-distributed servers denoted by N that are connected by network links. The service provider uses a *controller* for 1) VM (or equivalent container runtime) setup on servers, 2) MS instance placements on VMs, 3) storage replication on servers, and 4) request dispatching. MS instances need an isolated execution environment provided by VM. Thus, a controller cannot place any MS instance on a server unless and until it has set up a VM on that server. This is a property of the deployment pipeline rather than a policy decision (in Kubernetes, a pod specification naming a node whose runtime is not yet ready is simply not scheduled there). Setting up a VM takes considerable time (typically ranging from several seconds to a few minutes). Therefore, there is a time gap between setting up a VM and placing an MS instance, referred to as the *service rollout delay*. The controller allocates at most one VM in a server to host all MS instances that are part of the same service and lets them share the VM's computation resource. For this reason, we use VM and server interchangeably hereafter. We use virtualized storage (such as persistent volumes or software-defined storage) to place the storage spaces supporting microservices.

We assume a set of microservices M and a set of storage spaces S. Geo-dispersed users who want to access a service send their service requests to an arbitrary (possibly the nearest) server where a frontend MS is running. The frontend MS then calls other MS instances and, if needed, performs access operations on storage to deliver the service. Fig. 1 shows a service-storage interaction graph that depicts the architecture of a typical online shopping service. We divide the operation time into discrete time slots such that both user-generated and subsequent MS-generated requests can be completed within a single time slot.

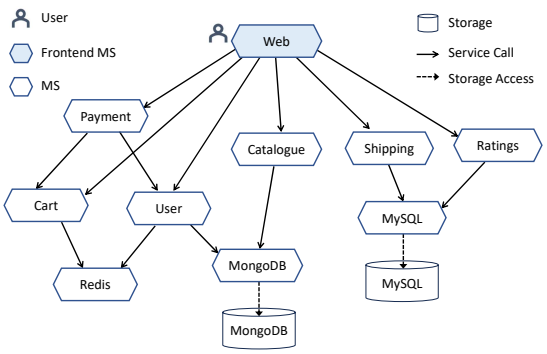

Figure 1: A service-storage interaction graph for the Robot Shop Application at https://github.com/instana/robot-shop

A. Request Dispatching and Aggregation

Each server is equipped with a *dispatcher* that determines where to forward an outgoing request. When a server receives an MS invocation or storage access request, its dispatcher can forward the request to any server (including itself) that hosts the target instance or storage space. The dispatcher follows a stochastic dispatching rule set up by the controller. We use $Y^m_{t,i,j}$ to denote the probability by which server i's dispatcher forwards the request calling MS m to a candidate server j which hosts an instance of m.

The request arrival rate at MS instance m on server i is an aggregation of the arrival rates of all upstream request sources targeting that instance. Let $R^m_{t,i}$ be the arrival rate at server i for MS m, $\eta^{m',m}$ be the probability that m' calls m, and $Y^m_{t,j,i}$ be the forwarding probability from j to i. We have

$$R^m_{t,i} = \begin{cases} \sum_{j \in N} \Lambda_{t,j} Y^m_{t,j,i}, & \text{if } m \text{ is a frontend} \\ \sum_{j \in N} \sum_{m' \in M} R^{m'}_{t,j} \eta^{m',m} Y^m_{t,j,i}, & \text{otherwise,} \end{cases} \tag{1}$$

where $\Lambda_{t,j}$ is user's request arrival rate (for the service) at server j in time slot t.

B. Storage Access Operations

Dispatcher is responsible for forwarding storage access requests but treats read and write requests differently. Dispatching a read request for storage s is similar to dispatching an MS call. The dispatcher can forward the operation to any replica of s with a probability determined by the controller. For an MS instance m running on server j, let $\eta^{m,s}_{\text{read}}$ be the probability that m issues a read request toward storage s in a call to m and $Y^{\text{read},s}_{t,m,j,i}$ be the probability that j's dispatcher forwards that request to server i which hosts a replica of s. The read request arrival rate at server i for storage s in time slot t is

$$R^{\text{read},s}_{t,i} = \sum_{m \in M} \sum_{j \in N} R^m_{t,j} \eta^{m,s}_{\text{read}} Y^{\text{read},s}_{t,m,j,i}. \tag{2}$$

By contrast, the dispatcher forwards a write request to all replicas of the target storage to maintain data consistency. Let

$\eta^{m,s}_{\text{write}}$ be the probability that MS m issues a write request to storage s in a call to m. The write request arrival rate at server i for storage s in time slot t is

$$R^{\text{write},s}_{t,i} = \sum_{m\in M}\sum_{j\in N} R^{m}_{t,j}\eta^{m,s}_{\text{write}}. \tag{3}$$

C. Queuing and Processing Time

Each server processes all MS invocation and storage access requests arriving at it in a first-come-first-served (FCFS) manner and uses a waiting queue to keep all requests that cannot be processed immediately. Therefore, the sojourn time of a request comprises the request's waiting time in the queue (if any) and its processing time. We assume that instances are all single-threaded, so request process on a server can be modeled as an M/G/1 queue. We assume Poisson arrivals at rate $\lambda^m_i = R^m_{t,i}$ and exponentially distributed processing times with MS-specific rate μ_m for each $m \in M$. With these assumptions, the mean sojourn time for a request to MS m on server i is [12]

$$W^m_{t,i} = \frac{\sum_{m'\in M}\frac{\lambda^{m'}_i}{\mu_{m'}{}^2}}{1-\sum_{m'\in M}\frac{\lambda^{m'}_i}{\mu_{m'}}} + \frac{1}{\mu_m}. \tag{4}$$

The total access request arrival rate at server i for storage s is $\lambda^s_i = R^{\text{read},s}_{t,i} + R^{\text{write},s}_{t,i}$, modeled as Poisson. With exponentially distributed operation times at rate μ_s, the M/M/1 expected sojourn time is

$$W^s_{t,i} = \frac{1}{\mu_s-\lambda^s_i} = \frac{1}{\mu_s-(R^{\text{read},s}_{t,i}+R^{\text{write},s}_{t,i})}. \tag{5}$$

D. Request Turnaround Time

A request calling an MS instance m on server i in time slot t incurs a turnaround time $D^m_{t,i}$ that includes its sojourn time $W^m_{t,i}$ plus delays from downstream MS calls and storage accesses. Each downstream call or storage access incurs a round-trip propagation delay $2\Delta_{i,j}$ plus the sojourn time at the target. Since a write request is broadcast to all replicas of storage s, its delay is determined by the slowest replica: $\max_{j\in N}\{Z^s_j(2\Delta_{i,j}+W^s_{t,j})\}$. Therefore,

$$\begin{aligned} D^m_{t,i} = {} & \underbrace{W^m_{t,i}}_{\text{(i) microservice sojourn time}} \\ & + \underbrace{\sum_{m'\in M}\eta^{m,m'}\sum_{j\in N} Y^{m'}_{t,i,j}(2\Delta_{i,j}+D^{m'}_{t,j})}_{\text{(ii) downstream microservice calls}} \\ & + \underbrace{\sum_{s\in S}\eta^{m,s}_{\text{read}}\sum_{j\in N} Y^{\text{read},s}_{t,m,i,j}(2\Delta_{i,j}+W^s_{t,j})}_{\text{(iii) downstream storage reads}} \\ & + \underbrace{\sum_{s\in S}\eta^{m,s}_{\text{write}}\cdot\max_{j\in N}\left\{Z^s_j(2\Delta_{i,j}+W^s_{t,j})\right\}}_{\text{(iv) downstream storage writes}}. \end{aligned}$$

A user request arriving at any server i in time slot t is forwarded to an instance of frontend MS m running on some server j based on $Y^m_{t,i,j}$. Therefore, the expected turnaround time of a user request that is generated in time slot t can be estimated as

$$D_t = \frac{\sum_{i\in N}\Lambda_{t,i}\sum_{j\in N}Y^m_{t,i,j}(2\Delta_{i,j}+D^m_{t,j})}{\sum_{i\in N}\Lambda_{t,i}}. \tag{6}$$

IV. Problem Formulation

The controller aims to minimize overall cost subject to SLA, queue stability, and physical constraints. Let κ_c and κ_s be the costs of allocating (i.e., booting or hosting) a VM and hosting a storage space in a server for one time slot, respectively. The total cost in time slot t is

$$C_t = \sum_{i\in N}\left(\kappa_c x_{t,i} + \sum_{s\in S}\kappa_s Z^s_{t,i}\right), \tag{7}$$

where $x_{t,i}$ indicates whether the controller is setting up or has set up a VM in server i, and $Z^s_{t,i}$ indicates whether i hosts a replica of storage s in time slot t. The controller aims to minimize C_t over time:

$$\min_{\mathbf{Z},\mathbf{Y},\mathbf{X}}\sum_{t\in T} C_t, \tag{8}$$

where $\mathbf{Z} = (Z^s_{t,i})_{t\in T,i\in N,s\in S}$ denotes storage replica placements, $\mathbf{Y} = (Y^m_{t,i,j}, Y^{\text{read},s}_{t,m,i,j})_{t\in T,m\in M,i,j\in N,s\in S}$ are the dispatching rules for MS invocation and storage read requests, and $\mathbf{X} = (x_{t,i}, X^m_{t,i})_{t\in T,i\in N}$ denotes VM setup and instance placement decisions. $X^m_{t,i}$ indicates whether an instance of MS m is placed in the VM on server i in time slot t. It is highly coupled with $x_{t,i}$ since the controller cannot place m if it did not set up the VM τ_{roll} time slots before t, where τ_{roll} is the service rollout delay in time slots.

The objective is subject to the following constraints. The SLA constraint requires

$$D_t < \delta_{\text{SLA}}, \quad \forall t\in T, \tag{9}$$

and the rollout-delay constraint reflects the τ_{roll}-slot VM setup period that must go through, as the deployment property defined in Sec. III, before any MS instance can be activated:

$$X^m_{t,i} \le \prod_{k=0}^{\tau_{\text{roll}}} x_{t-k,i}, \quad \forall t, \forall i\in N, \forall m\in M. \tag{10}$$

The remaining constraints require queue stability ($\sum_m \lambda^m_i < \mu_m$, $R^s_{t,i} < \mu^s$), validity of dispatching (requests routed only to active instances or replicas), coverage (at least one instance and replica per type), dispatching probabilities summing to one, and binary domains for all placement variables.

V. Proposed Approach

A. Heuristic Request Dispatching Rules

We adopt softmax-proportional dispatching for MS invocation requests, $Y^m_{t,i,j} = X^m_{t,j}\exp(\omega^c_{i,j})/\sum_{k\in N}\exp(\omega^c_{i,k})$, where

$$\omega^c_{i,j} = -\left(\beta_{\text{PG}}\frac{\Delta_{i,j}}{\bar{\Delta}_i} + \beta_{\text{CG}}\frac{1}{|M|}\sum_{m'\in M}\left(X^{m'}_{t,j}\cdot\frac{\bar{\mu}_M}{\mu_{m'}}\right)\right). \tag{11}$$

The first term of $\omega^c_{i,j}$ (with weight β_{PG}) is the propagation delay to j normalized by the network mean $\bar{\Delta}_i = \sum_{k\in N} \Delta_{i,k}/|N|$, and the second (with weight β_{CG}) is a slowness-weighted placement density on server j, where $\bar{\mu}_M = \sum_{m'\in M} \mu_{m'}/|M|$ is the mean MS service rate and $\bar{\mu}_M/\mu_{m'}$ penalizes slower MS types more heavily.

For storage reads, we use

$$Y^{\text{read},s}_{t,m,i,j} = Z^s_j \frac{\exp(\omega^s_{i,j})}{\sum_{k\in N}\exp(\omega^s_{i,k})}, \text{where } \omega^s_{i,j} = -\beta_{\text{PG}}\frac{\Delta_{i,j}}{\bar{\Delta}_i}, \tag{12}$$

since storage operations are less affected by server-side congestion.

B. MDP Formulation

We model storage replication, VM setup, and instance placement as an MDP. The action at time t is $a_t = \langle \mathbf{Z}_t, \mathbf{x}_t, \mathbf{X}_t\rangle$, where $\mathbf{Z}_t$, $\mathbf{x}_t$, $\mathbf{X}_t$ denote static storage, dynamic VM decisions, and dynamic MS placements, respectively. The observation $o_t = \langle \mathbf{\Lambda}_{t-1}, \mathbf{x}_{t-1}, \tilde{\mathbf{x}}_{t-1}\rangle$ stacks recent traffic, VM, and placement states into an extended state s_t to mitigate partial observability and rollout delays.

C. Hybrid Policy Gradient (HPG)

Since storage migration is prohibitively expensive, $\mathbf{Z}$ is committed in the beginning of each episode and adjusted only across episodes. $\mathbf{Z}$ is therefore a state-independent policy and the action reduces to $a_t = \langle \mathbf{Z}, \mathbf{x}_t, \mathbf{X}_t\rangle$ accordingly. However, we cannot apply standard SAC [13] directly for two fundamental reasons. First, SAC's Gaussian reparameterization is not applicable for our binary placement variables, while its discrete counterpart requires an exact expectation over the action space that is of cardinality $2^{|N|(|M|+|S|+1)}$. Second, SAC optimizes a scalar reward with no notion of constraint priority, whereas our problem places feasibility ahead of cost. HPG addresses each of these through a *Bernoulli relaxation for binary variables* and a *constraint-aware reward*.

a) Relaxed binary actions. To enable gradient-based learning for binary placement variables, we introduce a smoothed Bernoulli relaxation:

$$\tilde{b}_\phi = \rho\,\pi_\phi(b=1) + (1-\rho)\mathbb{I}_{\pi_\phi(b=1) > \text{Uniform}(0,1)}, \tag{13}$$

which not only provides a differentiable surrogate but also preserves stochastic exploration for discrete actions.

b) Constraint-aware reward. To encourage system constraints, we design a hierarchical staged reward:

$$r(s_t,a_t) = \sum_{k=1}^{4}\prod_{j=1}^{k} I^j_t - \frac{C_t}{(\kappa_c|N| + \kappa_s|N||S|)}, \tag{14}$$

where I^1_t–I^4_t indicate VM availability, service completeness, queue stability, and delay satisfaction, respectively. Hence, each level contributes only when all preceding ones hold. This staged structure departs from the scalar reward typically used in SAC and orders the feasibility constraint ahead of the cost within the reward for policy learning. To elaborate further, the structure integrates the feasibility requirements into cost minimization in the learning signal, rather than imposing them as hard constraints.

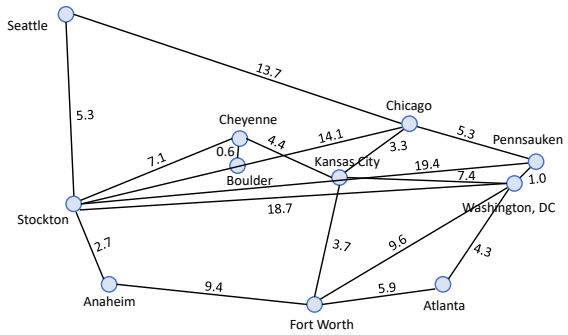

Figure 2: Network topology Sprint [14]. Each link is labeled with propagation delay (ms).

Within this structure, we adopt the SAC objective with entropy regularization, $\max_\pi \mathbb{E}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))]$, with automatic temperature tuning. Standard off-policy updates with a replay buffer are used; details are omitted for brevity since they follow SAC.

At each training step, static decisions $\mathbf{Z}$ and dynamic decisions $(\mathbf{x}_t, \mathbf{X}_t)$ are sampled from the hybrid policy, combined into a_t, and corrected to satisfy activation constraints. After training, online execution fixes $\mathbf{Z}$ per episode and greedily selects $(\mathbf{x}_t, \mathbf{X}_t)$ at each time slot.

VI. Performance Evaluation

A. Experiment Settings

We used network topology Sprint [14], as shown in Fig. 2, where all physical links were assumed optical fibers with refractive index $n = 1.47$. The propagation delay of a link with length l is nl/c, where c is the speed of light in vacuum.

We derived traffic traces from the 'production_power_util' field of the Google Cluster dataset [15] (May 14, 2019), which reflects data-center computational intensity and serves as a proxy for traffic demand. Each trace was smoothed with a one-hour moving average and then min-max normalized to $[0,1]$. Peak traffic at each server was then scaled by local population [16], and phase shifts were introduced to reflect time zone differences, yielding realistic spatio-temporal variations.

We used the Robot Shop Application with the service-storage interaction graph shown in Fig. 1. Table II lists the key parameters used in our experiments.

We compared HPG (with $\beta_{\text{PG}} = \beta_{\text{CG}} = 0.1$) against several representative baseline methods listed below.

- *HPG PG*: A variant of HPG whose request dispatching rule disregards service time (i.e., β_{CG} in (11) is 0).
- *HPG RR*: A variant of HPG that uses Round Robin (RR) instead of the proposed rules for request dispatching.
- *Static GA*: It uses the same request dispatching rules as HPG, but uses a Genetic Algorithm (GA) for storage

Table II: System model parameters Note: RPS denotes requests per second

Parameter	Default Value	Reference
$\eta^{m,m'}, \eta^{m,s}_{\text{read}} + \eta^{m,s}_{\text{write}}$	$[0.45, 0.55]$	
SLA Delay Limit (δ^{SLA})	20 ms	
Request Service Rate (μ_m)	$[5600, 6000]$	[17]
Storage I/O Capacity (μ_s)	$[900, 1100]$	
Writing Ratio	0.1	
Service Rollout Delay (τ_{roll})	120 sec.	[2]
Time Slot Length (τ)	30 sec.	
Total Time slots ($\lvert T \rvert$)	3000	
VM Cost per sec. (κ_c)	2.888×10^{-6}	[18]
Storage Cost per sec. (κ_s)	3.086×10^{-6}	[19]
Mean Request Arrival Rate	11k RPS per million users	

replication, VM setup, and instance placement. The fitness function is defined to be aligned with (14).

- *Reactive GA*: It runs Static GA initially and every ten time slots after for possible updates of VM setup and instance placement.

We considered the following two performance metrics:

- *SLA satisfaction rate*, which is the ratio of time slots in an episode where the SLA requirement (9) was met.
- *Normalized cost*, which is the mean episode cost normalized to the range $[0, 1]$.

HPG converged stably across 30 independent runs within 10^5 training steps. Regarding decision overhead, HPG requires only 0.7 ms per time slot for inference, compared to 33.3 ms for Reactive GA (which re-runs the GA every ten slots), both well within the 30-second slot duration.

B. Impacts of Request Arrival Rates

Table III shows the SLA satisfaction rates and normalized costs under varying mean request arrival rates. As the arrival rate increased, SLA satisfaction decreased while cost increased slowly, reflecting the need for additional resources to handle the higher queuing delays.

HPG outperformed Static GA in terms of SLA satisfaction, achieving 5.2% to 43.4% improvement in high-traffic scenarios (13k–19k RPS per million users), with only a 3.3% drop at 11k RPS. This demonstrates that Static GA could not adjust VM setup or instance placement during traffic shifts, causing higher queuing delays.

HPG and HPG PG show comparable performance, with relative differences between -4.8% and $+2.1\%$, suggesting server load is not a reliable proxy for queuing delay. Compared to HPG RR, HPG achieved improvements of 0.4% to 30.2% under most traffic conditions, with a slight 7.4% decrease observed at one traffic level, supporting the benefit of propagation-aware dispatching. Compared with Reactive GA, HPG incurred a cost increase of 19.0% to 60.2%, resulting in a substantial improvement in SLA satisfaction of 77.7% to 165.2%.

C. Impacts of Read-Write Ratio

As shown in Table IV, SLA satisfaction fell steadily as the write ratio increased (for HPG, from 0.903 at 0.1 to 0.548 at 0.9) and every method degraded by a comparable margin. This sweep exercises the write-broadcast consistency overhead: writes are forwarded to all replicas with the resulting delay set by the slowest one. The degradation is a property of consistent replication rather than of any scaling policy. Models omitting this overhead would therefore overstate achievable SLA satisfaction. The cost trend first rose, then fell, and rose again. The initial increase resulted from allocating more resources to satisfy SLA, followed by a decrease when SLA violations became unavoidable, and storage deployment was reduced, and finally another increase to preserve SLA for the majority of requests.

Table III: Performance Under Different User Request Rates

(a) SLA Satisfaction Rate

RPS per million users	11k	13k	15k	17k	19k
HPG	0.903	0.927	**0.955**	0.756	**0.78**
HPG PG	**0.949**	**0.949**	0.947	0.784	0.764
HPG RR	0.9	0.884	0.921	**0.817**	0.599
Static GA	0.934	0.881	0.666	0.626	0.628
Reactive GA	0.508	0.459	0.36	0.357	0.327

(b) Normalized Cost

RPS per million users	11k	13k	15k	17k	19k
HPG	0.392	0.408	0.499	0.422	0.432
HPG PG	0.35	0.435	0.496	0.42	0.379
HPG RR	0.395	0.462	0.478	0.507	**0.279**
Static GA	0.447	0.445	0.447	0.478	0.416
Reactive GA	**0.33**	**0.301**	**0.312**	**0.342**	0.28

At a write ratio of 0.5, HPG PG incurred a notably higher cost of 43.7% compared to Static GA, but also attained the highest SLA satisfaction at that setting (0.760 versus 0.720). This is an outlier relative to the other settings. In other cases, HPG achieved 2.3% to 9.6% lower SLA satisfaction and 0.8% to 17.2% cost reduction compared to Static GA. These are distinct operating points, not losses. Recall that (8) minimizes cost subject to the SLA, so a solution with lower cost and slightly lower SLA satisfaction is not inferior by the problem's own criterion. Since the bottleneck in this experiment lay in storage, which both methods deploy statically, the decision space in which HPG's dynamic VM and MS adjustments could help was correspondingly small.

HPG outperformed Reactive GA in SLA satisfaction by 51.0% to 90.7% at an additional cost of 5.3% to 26.7%. This performance gap was much smaller than in previous experiments. The narrower margin here was expected: this sweep stressed storage, which all three methods deployed statically, so it did not exercise the proactive-versus-reactive distinction that Reactive GA is included to isolate.

D. Impacts of Service Rollout Delay

Table V shows that the SLA satisfaction of the HPG series fluctuated without a clear trend as the service rollout delay increased, remaining above 0.9 throughout. All algorithms exhibited variation in cost. Reactive GA, by contrast, degraded monotonically from 0.725 at 0 s to 0.508 at 120 s, since a longer delay widens the gap between the state the GA

Table IV: Performance Under Different Writing Ratios

(a) SLA Satisfaction Rate

Writing Ratio	0.1	0.3	0.5	0.7	0.9
HPG	0.903	0.969	0.72	0.615	0.548
HPG PG	**0.949**	0.917	**0.76**	0.615	0.547
HPG RR	0.9	0.899	0.751	0.569	0.532
Static GA	0.934	**0.992**	0.72	**0.674**	**0.606**
Reactive GA	0.508	0.508	0.429	0.395	0.363

(b) Normalized Cost

Writing Ratio	0.1	0.3	0.5	0.7	0.9
HPG	0.392	0.451	0.37	0.298	0.388
HPG PG	0.35	0.447	0.559	0.301	0.402
HPG RR	0.395	0.432	0.423	**0.277**	0.356
Static GA	0.447	0.509	0.389	0.36	0.391
Reactive GA	**0.33**	**0.391**	**0.292**	0.283	**0.309**

Table V: Performance Under Different Service Rollout Delays

(a) SLA Satisfaction Rate

Service Rollout Delay	0 s	30 s	60 s	90 s	120 s
HPG	**1.0**	0.96	**0.992**	**0.999**	0.903
HPG PG	0.871	0.953	**0.992**	0.991	**0.949**
HPG RR	0.924	**0.969**	0.937	0.911	0.9
Static GA	0.92	0.924	0.946	0.952	0.934
Reactive GA	0.725	0.67	0.616	0.562	0.508

(b) Normalized Cost

Service Rollout Delay	0 s	30 s	60 s	90 s	120 s
HPG	0.337	0.464	0.416	0.364	0.392
HPG PG	0.367	0.428	0.38	0.386	0.35
HPG RR	0.457	0.469	0.4	0.394	0.395
Static GA	0.416	0.418	0.416	0.445	0.447
Reactive GA	**0.299**	**0.33**	**0.299**	**0.299**	**0.33**

references and the state in which the requested VM becomes active. Consequently, HPG's advantage over Reactive GA grew with the rollout delay, from 37.9% at 0 s to 77.7% to 77.8% at 90 to 120 s. This is the regime the proposed method targets: the value of acting in advance increases precisely when reactive scaling is structurally late. Static GA showed the smallest fluctuation in SLA (0.920 to 0.952) because it did not scale the number or placement of VMs. Hence, its performance was largely unaffected by the rollout delay, and the minor variation arose from the inherent instability of the Genetic Algorithm.

VII. Conclusions

We have formulated a joint optimization of MS placement, storage replication, and VM provisioning under rollout delays and consistency overhead, and developed HPG to solve it. HPG improves SLA satisfaction over Reactive GA by 51.2% to 165.2%, with the margin widening as the rollout delay grows, demonstrating the value of proactive orchestration. Propagation-aware dispatching outperforms round-robin by up to 30.2%, while added congestion-awareness shows mixed results.

Several directions remain open: introduce constrained RL, broader comparisons of learning baselines, ablations of the staged reward and the Bernoulli relaxation, validation on additional traces and larger topologies, and model storage migration so that $\mathbf{Z}$ becomes a dynamic decision.

Acknowledgment

This work was supported in part by the National Science and Technology Council of Taiwan under grant number NSTC 115-2640-E-A49-007 and 115-2221-E-A49-102-MY2.

References

[1] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proc. IEEE*, vol. 112, no. 1, pp. 12–46, 2024.

[2] J. Hao, T. Jiang, W. Wang, and I. K. Kim, "An empirical analysis of VM startup times in public IaaS clouds," in *Proc. IEEE CLOUD*, 2021, pp. 398–403.

[3] S. Ramanathan, G. Gautam, V. Srinivasan, and P. Parag, "Latency-redundancy tradeoff in distributed read-write systems," in *Proc. COMSNETS*, 2022, pp. 172–180.

[4] K. Peng, L. Wang, J. He, C. Cai, and M. Hu, "Joint optimization of service deployment and request routing for microservices in mobile edge computing," *IEEE Trans. Serv. Comput.*, vol. 17, no. 3, pp. 1016–1028, 2024.

[5] L. Wang, X. Liu, H. Ding, Y. Hu, K. Peng, and M. Hu, "Energy-delay-aware joint microservice deployment and request routing with DVFS in edge: A reinforcement learning approach," *IEEE Trans. Comput.*, vol. 74, no. 5, pp. 1589–1604, 2025.

[6] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundant placement for microservice-based applications at the edge," *IEEE T. Serv. Comput.*, vol. 15, no. 3, pp. 1732–1745, 2022.

[7] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Trans. Cloud. Comput.*, vol. 11, no. 2, pp. 1777–1793, 2023.

[8] S. Arora and A. Ksentini, "Dynamic resource allocation and placement of cloud native network services," in *Proc. IEEE ICC*, 2021, pp. 1–6.

[9] Y. Chen, C. Wu, F. Zhang, C. Lu, Y. Huang, and H. Lu, "Topology-aware microservice architecture in edge networks: Deployment optimization and implementation," *IEEE Trans. Mob. Comput.*, vol. 24, no. 7, pp. 6090–6105, 2025.

[10] D. Edirisinghe, K. Rajapakse, P. Abeysinghe, and S. Rathnayake, "SpotKube: Cost-optimal microservices deployment with cluster autoscaling and spot pricing," in *Proc. CloudCom*, 2024, pp. 87–94.

[11] M. H. Ferdaus, M. Murshed, R. N. Calheiros, and R. Buyya, "An algorithm for network and data-aware placement of multi-tier applications in cloud data centers," *J. Netw. Comput. Appl.*, vol. 98, no. C, p. 65–83, Nov. 2017.

[12] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory*, 4th ed. Wiley, 2008.

[13] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2018, *arXiv:1812.05905*.

[14] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet topology Zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, 2011.

[15] Google, "Google cluster data," 2019, accessed: 2026-04-26. [Online]. Available: https://github.com/google/cluster-data

[16] U.S. Census Bureau, "Press kit: Vintage 2024 national and state population estimates," 2024, accessed: 2026-04-26. [Online]. Available: https://www.census.gov/newsroom/press-kits/2024/national-state-population-estimates.html

[17] T. N. Do, "Express Benchmark," 2025, accessed: 2026-04-26. [Online]. Available: https://sharkbench.dev/web/javascript-express

[18] Amazon Web Services, "Amazon EC2 instance types: t3," 2025, accessed: 2025-08-05. [Online]. Available: https://aws.amazon.com/ec2/instance-types/t3/

[19] ——, "Amazon EMR storage and volume types," 2025, accessed: 2026-04-26. [Online]. Available: https://docs.aws.amazon.com/en_us/emr/latest/ManagementGuide/emr-plan-storage-compare-volume-types.html