Granularity-Aware Capacity Probing for Overload Recovery and Capacity Rediscovery in Distributed Systems with Thread-Pool Backends

Faisal Bahadur, Faqir Gul, Arooba Akhtar, Abdul Qadir, Nadir Hussain
Department of CS&IT, Hazara University, Mansehra, Pakistan.
***Correspondence:** faisal@hu.edu.pk

Citation | Bahadur. F, Faqir. G, Arooba. A, Qadir. A, Hussain. N, "Granularity-Aware Capacity Probing for Overload Recovery and Capacity Rediscovery in Distributed Systems with Thread-Pool Backends", IJIST, Vol. 8 Issue.6 pp 2695-2721, September 2026
DOI: https://doi.org/10.33411/IJIST/2027
Received | Aug 20, 2026 **Revised |** Sep 05, 2026 **Accepted |** Sep 05, 2026 **Published |** Sep 11, 2026

Distributed systems with thread-pool backends operate under changing workloads and require mechanisms for workload distribution, runtime adaptation, and overload control. Existing approaches address these problems through load balancing, concurrency adaptation, admission control, rate limiting, load shedding, and resource allocation. However, existing approaches do not explicitly reclaim additional post-overload capacity from a recovered backend without risking a return to a previously unsafe operating point. This paper presents Granularity-Aware Capacity Probing (GACP), an overload-recovery and capacity-rediscovery scheme for distributed thread-pool systems. After initial overload recovery, GACP progressively probes higher admitted request rates using finer probing granularity. Newly observed overload points are retained as unsafe boundaries to guide further probing. GACP was evaluated using PoolRunner on Oracle Cloud Infrastructure and compared with DTP, Hazelcast-DES, Sora, TopFull, and Rajomon. Each scheme was executed in five independent runs using different random seeds. The schemes were evaluated using pool size, throughput, response time, CPU usage, and heap usage. GACP achieved a low mean response time across all workload phases and the lowest mean response time among the compared schemes in the high-burst phase. In the high-burst phase, it reduced response time by 4.95% compared with DTP, 40.79% compared with Sora, 59.48% compared with Hazelcast-DES, 69.62% compared with TopFull, and 70.25% compared with Rajomon. In the same phase, GACP increased constrained-backend throughput by approximately 4.0% over DTP, while the paired statistical analysis confirmed significant improvements in both throughput and response time. Specifically, the paired analysis showed a mean throughput improvement of 52.20 RPS over DTP (95% CI: 51.16–53.24, $p < 0.001$) and a mean response-time reduction of 7.45 ms (95% CI: 6.94–7.96 ms, $p < 0.001$). These results show that GACP can reclaim additional post-overload capacity while maintaining low-latency operating conditions.

Keywords: Granularity-Aware Capacity Probing; overload recovery; capacity rediscovery; distributed systems; thread-pool backends; admitted-rate control; response time; overload control.

Introduction:

The growth of Internet, cloud, and distributed service platforms has increased the need for backend systems that can handle large, fluctuating, and unpredictable workloads [1][2][3]. Thread pools are widely used in backend server-side distributed applications to process many concurrent requests [4][5] without creating a new thread for each task. The size of a thread pool directly affects system performance [6][7][8]. If too many requests are admitted, excessive concurrency can increase waiting time, resource contention, and response latency, eventually causing overload. On the other hand, keeping the workload too low may leave useful processing capacity unused.

This problem becomes more difficult in distributed systems with thread-pool backends, where requests are shared among several backend servers with different capacities [9][10]. During workload bursts, some backends may become overloaded while others remain stable [11]. Therefore, a distributed system needs more than simple request distribution. It must identify overloaded servers, reduce their workload, and make effective use of their available capacity after overload recovery.

DTP [12] directly addresses overload recovery in distributed thread-pool systems. When a backend reaches an overload point, its admitted request rate is reduced so that the server can recover, while the remaining workload can be handled by other backends. However, after recovery, the backend may remain at the reduced rate even if it can safely process more requests. As a result, part of its available processing capacity may remain unused.

Recent Studies:

Recent work has explored related problems in distributed services from different angles. Sora [13] adapts microservice concurrency according to runtime measurements and response-time requirements. TopFull [14] controls overload through coordinated admission control, while Rajomon [15] uses decentralized rate limiting and load shedding for latency-sensitive microservices. Other recent studies consider overload-aware load balancing [16], distributed admission control [11], load balancing in containerized microservices [17], resource allocation for cloud-native applications [18], and distributed rate limiting [19]. These approaches provide different mechanisms for controlling workload and resources in distributed systems. However, they do not specifically focus on how additional post-overload capacity can be reclaimed from a thread-pool backend node after overload recovery.

The practical importance of this gap can be seen from the recovery behavior of DTP [12], which is the closest existing scheme to the present work. For example, if overload is detected at 10,000 requests/s, DTP's number-placement recovery reduces the admitted rate to 9,000 requests/s. However, if the backend can later sustain 9,500 requests/s, then 500 requests/s of usable capacity remains unutilized. This represents 5% of the original overload rate and approximately 5.6% above the recovered operating point. Existing approaches such as Sora [13], TopFull [14], and Rajomon [15] control concurrency or admitted workload during overload, but they do not explicitly measure or explore this post-recovery capacity gap. Therefore, a mechanism is needed that can gradually determine how much of the reduced capacity can be safely reclaimed after recovery.

Based on these observations, the following research question arises:

Research Question:

After an overloaded backend in a distributed system with thread-pool backends has recovered, how can its admitted workload be safely increased to reclaim additional post-overload capacity while maintaining low response time under the evaluated workload conditions?

Keeping the recovered server at a fixed lower limit may waste processing capacity, whereas increasing the workload too quickly may push it back into overload. A gradual

mechanism is therefore needed to explore the available capacity beyond the initial recovery point while avoiding aggressive return to previously observed overload levels.

Contribution:

To address this problem, this paper proposes the Granularity-Aware Capacity Probing (GACP) scheme. GACP uses the recovered operating point as the starting point and gradually increases the admitted request rate. If the backend remains stable, further capacity is explored. If overload occurs again, the new overload point is treated as an unsafe boundary and the probing step is refined. In this way, GACP combines overload recovery with controlled post-overload capacity rediscovery.

The main contributions of this work are as follows:

Post-overload capacity rediscovery: GACP extends overload recovery by exploring additional backend capacity beyond the initial recovery point.

Granularity-aware probing: The admitted request rate is gradually increased beyond the recovery point, while the probing step is refined when new overload conditions are observed.

Unsafe-boundary-based adaptation: Newly observed overload points are retained as unsafe boundaries to guide subsequent probing and reduce repeated operation near previously overloaded rates.

The proposed scheme is evaluated under normal, burst, and high-burst workloads and compared with prevalent schemes. Five independent runs are conducted using different random seeds, and the results are reported using mean, standard deviation, and 95% confidence intervals. A paired statistical analysis is also performed for the high-burst phase, where GACP becomes active after overload recovery. The evaluation shows that GACP maintains low response time across the workload phases and achieves the lowest mean response time among the compared schemes in the high-burst phase, while recovering additional constrained-backend throughput beyond the DTP [12] recovery point.

Flow of Research:

The research follows a sequence from problem identification to final research findings. First, the research objectives are defined based on the identified post-overload capacity-rediscovery problem. GACP is then designed and developed, followed by preparation of the OCI experimental setup, selection and configuration of comparative baselines, and workload definition. Each scheme is executed through five seeded runs with VM-B logging, after which the collected results are statistically analyzed to derive the final research findings.

Research Objectives:

The objectives of this research are to develop a mechanism that can reclaim additional backend capacity after overload recovery; progressively probe above the recovered admitted-rate limit without immediately returning to previously observed overload points; refine the probing granularity when new overload conditions occur; and evaluate the proposed mechanism against relevant distributed execution and overload-control schemes under different workload intensities.

Novelty of the Study:

GACP addresses the post-overload capacity-rediscovery gap by treating recovery as the starting point rather than the end of the control process. Instead of keeping the recovered backend at its reduced limit, GACP gradually probes higher admitted request rates.

Stable operation indicates that additional capacity may be available. If overload appears again, that point is treated as an unsafe boundary and the probing granularity is reduced. Thus, GACP combines overload recovery with controlled post-overload capacity rediscovery.

The main difference between GACP and existing distributed approaches is its focus on post-overload capacity probing for distributed systems with thread-pool backends. It is not an admission-control, load-balancing, rate-limiting, or resource-provisioning mechanism. Its

specific purpose is to determine how much additional backend capacity can be reclaimed after recovery while maintaining controlled operation.

Related Work:

Recent work on distributed systems has addressed workload management, overload control, admission control, and runtime resource adaptation. The following studies are particularly relevant to the problem considered in this work.

Our previous work, DTP [12], directly addresses overload recovery in distributed thread-pool systems. When a backend reaches an overload point, its admitted request rate is reduced to help it recover, while the excess workload can be redirected to stable servers. However, the recovered backend may remain at a conservative request limit, even when additional capacity may still be available.

Hazelcast Distributed Executor Service [20] provides distributed task execution across cluster members. It supports the distribution and execution of tasks on remote servers but does not provide a mechanism for recovering from overload and then probing the recovered backend for additional capacity.

Sora [13] addresses latency-sensitive resource adaptation in microservices. It uses runtime observations to select an appropriate concurrency level while considering response-time constraints. Unlike the present work, its focus is general runtime concurrency adaptation rather than capacity exploration after an overload event.

TopFull [14] controls overload in SLO-oriented microservices through coordinated admission control. Rajomon [15] takes a decentralized approach and combines coordinated rate limiting and load shedding to protect latency-sensitive services during traffic surges. Both approaches regulate workload during overload, but neither specifically investigates how much additional capacity can be reclaimed from a recovered thread-pool backend.

Author [16] studies dynamic load balancing together with overload control for microservices. DRACO [11] uses distributed, resource-aware admission control to manage overload in large-scale multi-tier systems. These approaches focus on preventing or controlling overload across distributed services rather than exploring additional backend capacity after recovery.

Other recent work considers related aspects of distributed workload and resource management. Author [17] use Docker Swarm for load balancing and service discovery in microservice-based applications. Authors [18] investigate resource allocation for cloud-native microservices while considering application performance requirements. [19] study distributed rate limiting in decentralized cloud networks. These studies regulate workload, traffic, or resource allocation at different levels, but they do not focus on post-overload capacity rediscovery in distributed systems with thread-pool backends.

Recent studies have also explored predictive and SLO-aware resource adaptation in cloud and microservice environments. GRAF uses graph neural networks to predict the effect of traffic changes and identify resource configurations that satisfy latency SLOs [21]. MS-RA uses service-level requirements to support self-adaptive resource allocation in microservices [22], while predictive autoscaling approaches combine workload forecasting with dynamic resource adjustment under changing arrival patterns [23]. These methods improve proactive resource management. However, their main control variables are resource allocation or replica scaling rather than rediscovery of the admitted-rate capacity of a thread-pool backend after overload recovery.

Research Gap:

The reviewed approaches address distributed workload management from different perspectives. DTP [12] focuses on overload recovery in distributed thread-pool systems, whereas Hazelcast [20] provides distributed task execution. Sora [13] adapts runtime concurrency, while TopFull [14], Rajomon [15], DRACO [11], and distributed rate limiting

[19] regulate workload through admission control, rate limiting, or load shedding. BLOC [16] and the Docker Swarm-based approach [17] focus on overload-aware workload distribution, whereas the approach in [18] addresses resource allocation for cloud-native microservices.

Thus, the existing schemes mainly concentrate on overload mitigation, workload distribution, runtime concurrency adaptation, admission control, rate limiting, or resource provisioning. These mechanisms help prevent overload or improve resource utilization, but they do not specifically address the post-overload stage in which a thread-pool backend has already recovered and may still be operating below its available processing capacity.

This leaves a specific gap: how can the admitted workload of a recovered thread-pool backend be safely increased to rediscover and reclaim additional post-overload capacity without repeatedly driving the backend into a new overload condition?

A fixed recovery limit may leave useful capacity unused, whereas an uncontrolled increase in workload may trigger another overload. Therefore, a mechanism is needed to progressively explore capacity beyond the recovery point while using newly observed overload conditions to guide further adaptation.

Material and Methods:

Methodological Overview:

This study follows a sequence from problem identification to final research findings as shown in Figure 1. First, the post-overload capacity-rediscovery problem is identified, where a recovered backend may remain below its sustainable processing capacity. Based on this problem, the research objectives are defined to develop a mechanism that can safely reclaim additional backend capacity after overload recovery.

GACP is then designed and developed using controlled capacity probing, unsafe-boundary management, and adaptive probing granularity. After development, the distributed experimental environment is prepared on Oracle Cloud Infrastructure using PoolRunner with one load-balancer VM and two heterogeneous backend VMs. Relevant comparative schemes are then selected for inclusion, configured according to their native control mechanisms, and deployed sequentially under the same experimental conditions.

The workload is configured using normal, burst, and high-burst phases. Each scheme is executed through five independent runs using random seeds 0–4, while performance logs are collected from the resource-constrained VM-B. The collected results include pool size, throughput, response time, CPU usage, and heap usage. These results are then summarized and statistically analyzed using mean, standard deviation, 95% confidence intervals, and paired statistical testing for the high-burst phase. The final research findings are derived from this comparative and statistical analysis.

Figure 1 shows the complete workflow of the study, where each stage corresponds to a specific part of the manuscript. The problem and research objectives are presented in the Introduction and Research Gap sections. The design of GACP, including the baseline DTP recovery model, granularity-aware probing, unsafe-boundary control, and the proposed algorithm, is described in the "GACP Architecture and Design" section. The OCI deployment is explained in "Experimental Infrastructure and Backend Deployment." The comparative schemes and their configurations are presented in "Evaluation Methodology," while the three workload phases and five seeded runs are defined in "Experimental Workload Intensity and Burst Patterns." The collected results are analyzed in the "Result and Discussion" section using mean, standard deviation, and 95% confidence intervals, followed by paired statistical testing in the "Statistical Significance Analysis" subsection. Finally, the main findings are summarized in the Discussion and Conclusion sections.

Figure 1. Research process from problem identification to statistical evaluation of GACP.

Experimental Infrastructure and Backend Deployment:

This section evaluates GACP and the comparative schemes using PoolRunner [24], whose client tier generates workload requests and the server tier receives requests and forwards them to the embedded load balancer, which distributes them among the available distributed thread pool backends. The experiments were carried out on Oracle Cloud Infrastructure (OCI) [25] using a distributed load-balancer-backend deployment. The PoolRunner client ran on a local Intel Core i7 machine with 16 GB RAM, Windows 11, and Java/OpenJDK 17. The cloud side used three OCI Ampere A1 VM.Standard.A1.Flex virtual machines with Ubuntu 22.04 LTS and Java/OpenJDK 17 [26],[27]. VM-A hosted the load balancer with 16 OCPUs and 16 GB RAM. VM-B was the resource-constrained backend with 2 OCPUs and 4 GB RAM, while VM-C was the higher-capacity backend with 16 OCPUs and 16 GB RAM. Each backend VM ran a separate Thread Pool System (TPS) instance.

The local PoolRunner client sent workload requests to VM-A through VM-A's public IP address on the configured PoolRunner application port. VM-A then forwarded requests to VM-B and VM-C through private IP addresses inside the same OCI region and virtual cloud network [28]. The same request path, VM configuration, monitoring interval, and logging mechanism were used for all schemes. Only the required application ports for PoolRunner, load-balancer communication, backend TPS execution, and Hazelcast-DES were enabled. No artificial delay, packet loss, bandwidth limit, VPN, or WAN emulation were added.

For the reported comparative results, CSV logs were collected from VM-B because the study focuses on overload recovery and capacity rediscovery on the constrained backend. The logs recorded a per-second admitted request rate, controller-level pool-size setting, throughput, average response time, queue length, failed or rejected submissions, CPU usage, memory usage, and overload-related decisions. These CSV files were used for plotting and analysis. Thus, the reported differences mainly reflect the behavior of the compared workload-control and overload-recovery schemes, not differences in workload generation, deployment path, or logging. Each scheme was executed under the same controlled configuration in five independent runs using different random seeds to obtain the mean, standard deviation, and 95% confidence interval of each metric.

The same random seeds were used across schemes to support paired statistical comparison in the high-burst phase.

GACP Architecture and Design:

GACP is designed as an extension to the overload-recovery stage of our previous work, DTP [12]. Figure 2 shows where GACP is integrated into the distributed TPS architecture. The client tier sends dynamic workload requests to the load balancer, which distributes them among backend TPS instances (distributed in a cloud environment) through the collaborative round-robin scheduler. Each backend TPS monitors its queue-average-wait and reports an overload point when overload is detected. The load balancer first applies the number-placement-based recovery mechanism and then activates the GACP controller. The GACP controller maintains the admitted-rate limit, unsafe boundary, and probing granularity for each overloaded backend server. Based on these values, it refines the admitted request rate and searches for a safer stable operating point while keeping the thread pool size parallel to the admitted load.

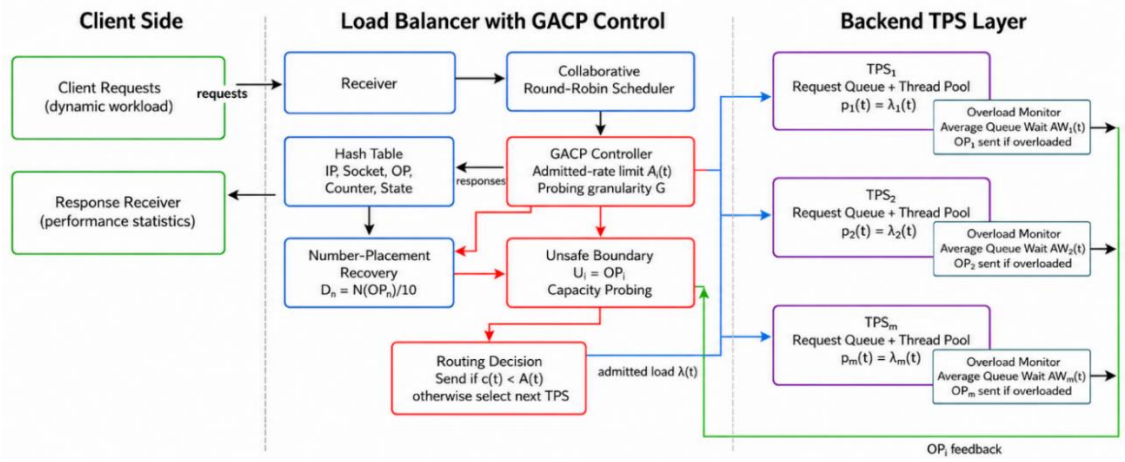

Figure 2. Integration of proposed Granularity-Aware Capacity Probing scheme with DTP.

Before presenting the proposed Granularity-Aware Capacity Probing (GACP) scheme, it is necessary to briefly review the baseline DTP, because GACP is integrated into its overload-recovery mechanism. The baseline DTP controls each backend Thread Pool System (TPS) through the admitted request rate, while the thread pool size remains parallel to that rate. Therefore, the baseline model, overload detection rule, and initial overload-recovery mechanism are first described in the following subsection. After establishing this baseline, the proposed GACP scheme is introduced as an extension that refines the recovered admitted-rate limit through controlled capacity probing.

Baseline DTP Model and Initial Overload Recovery:

Since GACP extends the overload-recovery stage of the baseline DTP model [4], this subsection briefly summarizes the baseline model, overload detection rule, and initial recovery step. Let the distributed system contain backend TPS nodes B={TPS_1,TPS_2,…,TPS_m}, where m is the number of backend servers connected to the load balancer. The load balancer receives incoming client requests and distributes them among these backend TPS nodes.

For a backend server TPS_i, let λ_i (t) denote the admitted request rate assigned during monitoring interval t. Since the backend TPS is modeled as an unbounded request-rate-driven TPS, its controller-level pool-size setting follows the admitted request rate:

$$p_i(t) = \lambda_i(t) \qquad (1)$$

where $p_i(t)$ denotes the controller-level pool-size setting of TPS_i during interval t. Thus, controlling the admitted request rate also controls the pool-size setting used by the backend TPS controller.

Each backend TPS detects overload using the average waiting time of requests in its queue. Let $q_i(t)$ be the number of queued requests at TPS_i during interval t. For each queued request r_j, let AT_j be its arrival time and $CT(t)$ be the current time. The average queue wait is computed as:

$$AW_i(t) = \frac{1}{q_i(t)} \sum_{j=1}^{q_i(t)} (CT(t) - AT_j) \qquad (2)$$

where $AW_i(t)$ is the average waiting time of queued requests currently queued at backend TPS_i, $q_i(t)$ is the number of queued requests, AT_j is the arrival time of queued request r_j, and $CT(t)$ is the current time at monitoring interval t.

The percentage increase in average wait is computed as:

$$\Delta AW_i(t) = \frac{AW_i(t) - AW_i(t-1)}{\max(AW_i(t-1), \epsilon)} \times 100 \qquad (3)$$

where ϵ is a very small positive value used to avoid division by zero. The overload state of TPS_i is defined as:

$$O_i(t) = \begin{cases} 1, & \text{if } \Delta AW_i(t) > \theta_{AW} \\ 0, & \text{otherwise} \end{cases} \qquad (4)$$

where θ_{AW} is the average-wait growth threshold. If $O_i(t) = 1$, then TPS_i is considered overloaded. When overload is detected, the current admitted request rate is treated as the overload point:

$$OP_i^k = \lambda_i(t), \text{if } O_i(t) = 1 \qquad (5)$$

where OP_i^k is the kth overload point observed for backend server TPS_i.

The baseline DTP uses the number placement of the overload point to reduce the admitted load. Let $\mathcal{N}(x)$ denote the number placement of x, defined as:

$$\mathcal{N}(x) = 10^{\lfloor \log_{10}(x) \rfloor}, x \geq 1 \qquad (6)$$

For example, $\mathcal{N}(10000) = 10000$, $\mathcal{N}(9500) = 1000$, and $\mathcal{N}(945) = 100$. When TPS_i becomes overloaded at OP_i^k, the recovery amount is computed as:

$$D_i^k = \frac{\mathcal{N}(OP_i^k)}{10} \qquad (7)$$

where D_i^k is the deceleration amount applied after the kth overload point. The recovered admitted-rate limit is then computed as:

$$A_i(t+1) = OP_i^k - D_i^k \qquad (8)$$

where $A_i(t+1)$ is the admitted-rate limit assigned to TPS_i for the next monitoring interval. For example, if $OP_i^1 = 10000$, then $\mathcal{N}(10000) = 10000$, $D_i^1 = 10000/10 = 1000$, and $A_i(t+1) = 10000 - 1000 = 9000$. Thus, the backend server initially recovered from 10,000 requests/s to 9,000 requests/s. GACP uses this recovered value as the starting point for controlled capacity probing.

Proposed Granularity-Aware Capacity Probing:

The main limitation of stopping at Equation (8) is that the backend server may still be able to process more than the reduced admitted rate. For example, after recovering from 10,000 requests/s to 9,000 requests/s, the server may still safely sustain 9,500 requests/s. In this case, fixed recovery protects the backend from repeated overload but may also leave useful processing capacity unused.

GACP addresses this limitation by adding a controlled probing phase after the initial number-placement-based recovery. The overload point is stored as an unsafe boundary, and the recovered admitted-rate limit is gradually increased using a probing granularity. If the

backend remains stable, GACP continues probing toward a safer stable rate. If overload appears again, the current admitted rate becomes a new overload point, the unsafe boundary is updated, and the probing step is reduced.

Figure 3 summarizes the control flow of the proposed GACP scheme. The process starts from overload-point detection, applies number-placement recovery, registers the unsafe boundary, performs capacity probing, updates the admitted-rate control, and then uses backend feedback to decide whether probing should continue or be refined.

Let G_i^k denote the probing granularity used after the k^{th} overload point. The initial probing granularity is derived from the recovery amount:

$$G_i^k = \begin{cases} \max\left(1, \left\lfloor \frac{D_i^k}{10} \right\rfloor\right), & \text{if } k = 1 \\ \max\left(1, \min\left(\left\lfloor \frac{D_i^k}{10} \right\rfloor, \left\lfloor \frac{G_i^{k-1}}{10} \right\rfloor\right)\right), & \text{if } k > 1 \end{cases} \quad (9)$$

Equation (9) ensures that the probing granularity becomes finer after repeated overload events. The minimum probing granularity is one request per second. Therefore, the probing process never uses fractional increments.

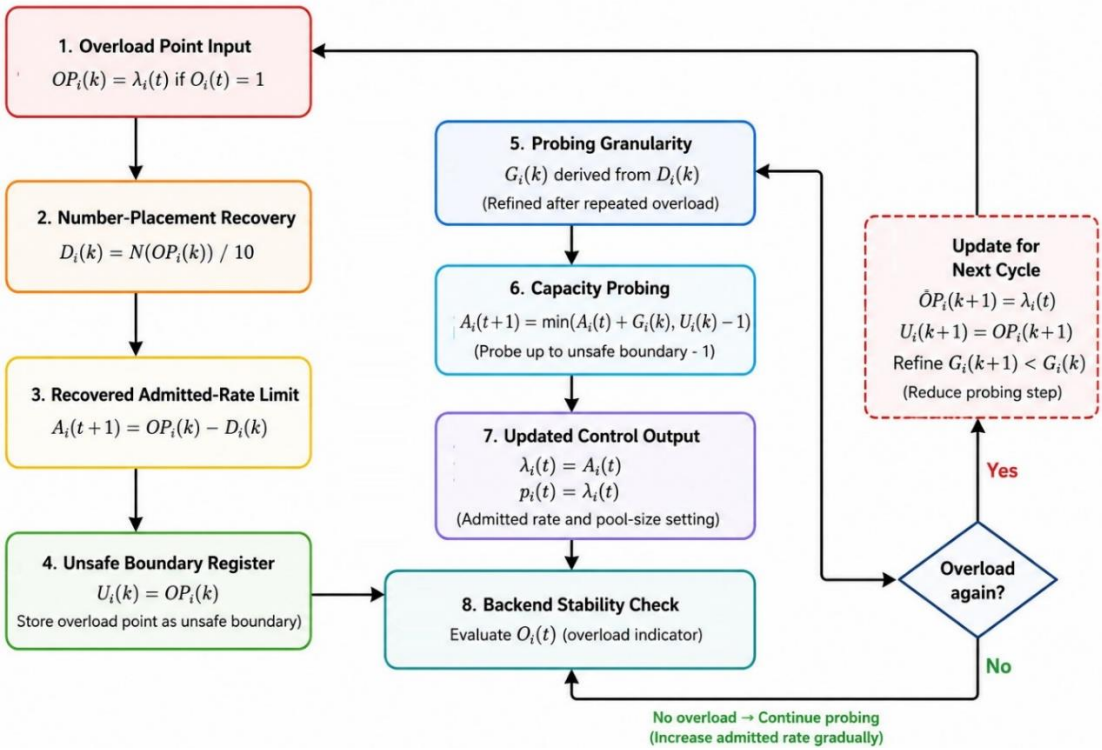

Figure 3. Control flow of the proposed Granularity-Aware Capacity Probing (GACP) scheme.

After overload recovery, if the backend server does not become overloaded in the next interval, the candidate admitted rate is increased as:

$$A_i(t+1) = A_i(t) + G_i^k \quad (10)$$

where G_i^k is the current probing granularity. This is a candidate probing update. The final admitted rate is later bounded by the unsafe boundary (as defined in Equation (13).

If overload occurs again during probing, the current admitted rate becomes the new overload point:

$$OP_i^{k+1} = A_i(t), \text{if } O_i(t) = 1 \quad (11)$$

The system then computes a new recovery amount using Equation (7), reduces the admitted rate using Equation (8), and updates the probing granularity using Equation (9).

Unsafe Boundary and Probing Control:

Each overload point represents an unsafe operating boundary for the backend server under the current runtime condition. Therefore, when TPS_i becomes overloaded at OP_i^k, the unsafe boundary is stored as:

$$U_i^k = OP_i^k \qquad (12)$$

where U_i^k is the current unsafe boundary of TPS_i.

During probing, the admitted rate should not immediately jump back to the unsafe boundary. Therefore, the next admitted rate is bounded as:

$$A_i(t+1) = \min\left(A_i(t) + G_i^k, U_i^k - 1\right) \qquad (13)$$

Equation (13) allows the load balancer to move closer to the unsafe boundary, but it avoids directly assigning the previously overloaded rate again.

If the admitted rate approaches the unsafe boundary and no overload has occurred, the probing granularity is reduced further:

$$G_i^k = \max\left(1, \left\lfloor \frac{G_i^k}{10} \right\rfloor\right) \qquad (14)$$

Equation (14) allows the system to search more precisely near the boundary between stable and unstable operation.

Numerical Example:

Suppose the first overload is observed at $10{,}000$ requests/s. The number-placement recovery gives $D_i^1 = 1{,}000$, so the recovered admitted rate becomes $A_i = 9{,}000$ requests/s and the unsafe boundary is stored as $U_i^1 = 10{,}000$. From Equation (9), the initial probing granularity is $G_i^1 = 100$ requests/s. If the backend remains stable, Equation (13) increases the admitted rate successively to 9,100, 9,200, 9,300 requests/s, and so on. Suppose a new overload is observed at 9,500 requests/s. This rate becomes the new unsafe boundary, $U_i^2 = 9{,}500$. The new recovery amount is 100 requests/s, so the admitted rate is reduced to 9,400 requests/s. Equation (9) then refines the probing granularity from 100 to 10 requests/s. Subsequent stable probes therefore become 9,410, 9,420, 9,430 requests/s, and so forth, while Equation (13) prevents the admitted rate from returning directly to the unsafe 9,500 requests/s boundary. If probing reaches 9,499 requests/s without another overload, Equation (14) further reduces the granularity from 10 to 1 request/s. This example illustrates how GACP progressively narrows the search around the boundary between stable and overloaded operation.

Final Admitted Rate and Pool Size:

The final admitted rate assigned to TPS_i during interval t is:

$$\lambda_i(t) = \begin{cases} RR_i(t), & \text{if } TPS_i \text{ is in normal state} \\ A_i(t), & \text{if } TPS_i \text{ is in probing state} \end{cases} \qquad (15)$$

where $RR_i(t)$ is the request rate assigned by the normal round-robin scheduler.

Since the backend TPS is unbounded and the pool size follows the request rate, the final pool size becomes:

$$p_i(t) = \lambda_i(t) \qquad (16)$$

Thus, GACP controls the backend server in two ways. First, it limits the admitted request rate after overload. Second, it indirectly controls the thread pool size by controlling the request rate sent to the backend TPS.

Algorithm of the Proposed GACP Scheme:

Algorithm 1 presents GACP after overload feedback has been obtained from the baseline DTP monitor. The baseline monitor computes the overload state using Equations

(2)–(4), while GACP starts from the reported overload state and the current admitted rate. When overload is reported, GACP stores the current admitted rate as the overload point, records it as an unsafe boundary, applies number-placement recovery, and initializes or refines the probing granularity. During later stable intervals, GACP probes higher admitted-rate limits while avoiding an immediate return to the previously unsafe boundary.

Algorithm 1: Granularity-Aware Capacity Probing

Require:
B = {TPS1, TPS2, ..., TPSm}: set of backend thread pool systems
Oi(t): overload state of TPSi obtained from baseline overload monitor using Equations (2)–(4)
λi(t): admitted request rate of TPSi during interval t
N(.): number-placement function
ci(t): number of requests already sent to TPSi during interval t
Ensure:
Updated admitted-rate limit Ai(t)
Updated unsafe boundary Ui
Updated probing granularity Gi
Routing decision under GACP control
Controller-level pool-size setting pi(t)
1: for each backend server TPSi ∈ B do
2: Si ← NORMAL // TPSi initially receives normal round-robin traffic
3: OPi ← NULL // no overload point has been observed yet
4: Ui ← NULL // no unsafe boundary is known initially
5: Ai ← ∞ // no admitted-rate limit is applied in normal state
6: Gi ← NULL // probing granularity is initialized after first overload
7: ki ← 0 // overload counter for TPSi
8: ci ← 0 // request counter for the current monitoring interval
9: end for
10: for each monitoring interval t do
11: for each backend server TPSi ∈ B do
12: if Oi(t) = 1 then // baseline monitor reports overload
13: ki ← ki + 1 // record a new overload event
14: OPi ← λi(t) // current admitted rate becomes overload point
15: Ui ← OPi // overload point is stored as unsafe boundary
16: Di ← N(OPi) / 10 // compute number-placement recovery amount
17: Ai ← OPi − Di // apply initial recovery below overload point
18: if Gi = NULL then
19: Gi ← max(1, floor(Di / 10)) // initialize probing step after first overload
20: else
21: Gi ← max(1, min(floor(Di / 10), floor(Gi / 10))) // refine probing step after repeated overload
22: end if
23: Si ← PROBING // move TPSi from normal state to GACP probing state
24: else if Si = PROBING then // backend is stable during probing
25: Acandidate ← Ai + Gi // try a higher admitted-rate limit
26: Ai ← min(Acandidate, Ui − 1) // do not return directly to unsafe boundary
27: if Ai = Ui − 1 and Gi > 1 then
28: Gi ← max(1, floor(Gi / 10)) // reduce step size near unsafe boundary
29: end if

30: else
31: Ai ← ∞ // no rate limit is needed in normal state
32: end if
33: ci ← 0 // reset interval request counter
34: end for
35: while client requests arrive during interval t do
36: Select next backend server TPSi using round-robin order
37: if Si = NORMAL then
38: Forward request to TPSi
39: ci ← ci + 1
40: else if Si = PROBING and ci < Ai then // enforce GACP admitted-rate limit
41: Forward request to TPSi
42: ci ← ci + 1
43: else
44: Skip TPSi temporarily and select the next backend server whose admitted-rate limit has not been reached // avoid sending extra load to restricted backend
45: end if
46: end while
47: for each backend server TPSi ∈ B do
48: λi(t) ← number of requests forwarded to TPSi during interval t
49: pi(t) ← λi(t) // update controller-level pool-size setting
50: end for
51: end for

Evaluation Methodology:

The experimental infrastructure and backend deployment have already been discussed in the previous section. Therefore, this section focuses on the comparative evaluation, including the selection of baseline schemes, their configuration, and the workload used during evaluation.

Inclusion and Exclusion of Comparative Schemes:

The selection of schemes for evaluation and comparison is based mainly on their relevance to the distributed thread-pool environment, overload control, and the feasibility of applying them under the same experimental setup.

DTP [12] is included because it directly addresses overload recovery in distributed thread-pool systems and is the closest baseline to the proposed work. Hazelcast-DES [20] is included as a practical distributed executor service baseline. In this setup, VM-A coordinated task submission, while VM-B and VM-C acted as Hazelcast backend members. Each backend member was configured with a fixed executor pool of 150 threads, estimated from a peak per-backend load of 1500 requests/s × 0.1 s. Sora [13] is included through its SCG component, which can be used as a runtime controller for determining backend concurrency. TopFull [14] and Rajomon [15] are included as recent distributed overload-control approaches that regulate workload under dynamic and overloaded conditions.

The remaining schemes, including BLOC [16], DRACO [11], and the approaches in [17][18][19], are excluded from experimental comparison. Some require a substantially different deployment architecture, such as multi-tier, container-orchestrated, or decentralized network environments [11], [17], [19]. Others focus mainly on resource provisioning, network-level traffic regulation, or system-level workload management [16], [18], [19]. Reproducing these schemes would require major changes to the current experimental setup and would not provide a direct comparison under the same distributed thread-pool environment. Therefore, they are retained in the related work discussion but are not used as experimental baselines.

Configuration of Comparative Baseline Schemes:

Each comparative scheme is evaluated using its native overload-control mechanism. The DTP/GACP request-forwarding mechanism is therefore disabled for schemes that do not originally include it, because enabling it would create a hybrid implementation and would not represent the original baseline.

For the Sora baseline [13], VM-A distributes requests to VM-B and VM-C using round-robin scheduling, while the DTP request-migration mechanism is disabled. Sora is deployed on backends as a runtime concurrency controller. Its initial concurrency is configured according to the expected workload and task service time; with an approximately 100 ms task duration, 50 concurrent worker threads are used as the initial setting for a load of about 500 RPS (see next section). Sora then monitors response-time/SLO behavior and adjusts the allowed concurrency level at runtime according to the configured 500 ms SLO. Thus, Sora handles overload through runtime concurrency adaptation.

For the TopFull [14] baseline, VM-A distributes requests to VM-B and VM-C using the common round-robin scheduler. VM-B and VM-C are configured with a fixed thread pool of 150 threads throughout the experiment. This value of 150 was selected from the peak expected backend workload of approximately 1500 RPS (see next section) and the 100 ms task service time, giving an estimated concurrency requirement of 1500 x 0.1 = 150 threads. TopFull does not resize this pool; instead, it monitors application performance against the configured response-time SLO of 500 ms and dynamically controls the request-admission rate. When overload or SLO degradation is observed, TopFull reduces the rate admitted to VM-B and rejects or throttles excess requests. As performance recovers, the admitted rate can be increased again according to TopFull's native control policy. Thus, TopFull handles overload through adaptive admission control while the underlying thread-pool size remains fixed.

For the Rajomon [15] baseline, VM-A distributes requests to VM-B and VM-C using the common round-robin scheduler, while both VMs use a fixed thread pool of 150 threads. Rajomon does not resize the thread pool; instead, it regulates workload through its decentralized token- and price-based rate-control mechanism. When response-time/SLO pressure indicates overload, Rajomon reduces the effective admitted request rate and sheds unsupported requests. As the service recovers, the permitted workload can increase again according to Rajomon's native control policy. No DTP/GACP request migration is applied.

Experimental Workload Intensity and Burst Patterns:

Following the workload intensities used in recent cloud-native microservice research [29], we employ synthetic workloads of 1000 RPS as the normal load, 2,000 RPS as the burst load, and 3000 RPS as the high-burst load. This load is generated from the local PoolRunner client machine to stress the distributed TPS and drive the resource-constrained backend VM-B into overload. The workload pattern is shown in Figure 4, where the x-axis represents time in seconds and the y-axis represents the request rate per second. As shown in Figure 4, the workload consists of three phases.

These workload levels were selected relative to the capacity of the constrained VM-B rather than as universal workload thresholds. Because VM-A distributes requests approximately evenly between the two backends, total loads of 1000, 2000, and 3000 RPS correspond to approximately 500, 1000, and 1500 RPS at VM-B, respectively. In the selected 2-OCPU, 4-GB VM-B configuration, the first two levels remained within stable operation and did not activate the overload-recovery mechanism, whereas the approximately 1500 RPS backend load in the third phase consistently pushed VM-B into the overload region. Therefore, the three workload levels were used operationally to represent normal, burst, and high-burst conditions for this experimental configuration.

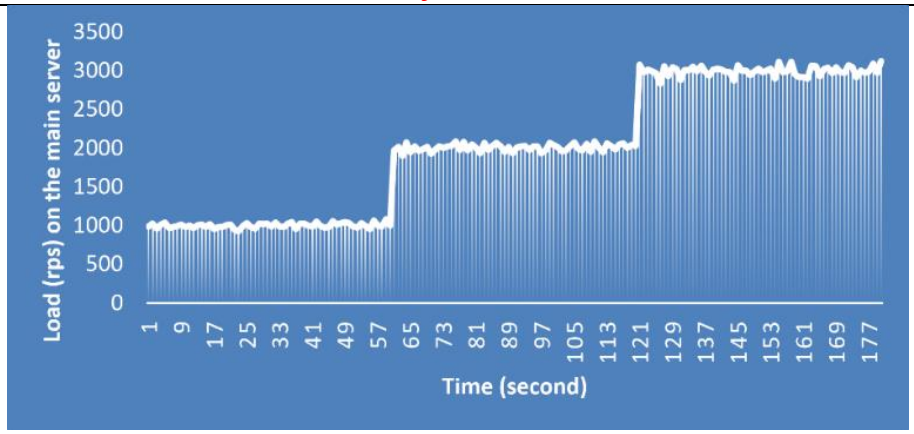

Figure 4. Three-phase workload applied during the 180-second experimental run

The 180-second experiment is divided into three consecutive 60-second phases: 0–60 s at $\lambda = 1000$ RPS, 61–120 s at $\lambda = 2000$ RPS, and 121–180 s at $\lambda = 3000$ RPS.

At the beginning of the experiment (first phase), the request rate is generated using a Poisson distribution with λ=1000 for one minute. After one minute, the workload is increased to λ = 2000 for the second phase and then to λ = 3000 for the third phase, with each phase lasting one minute. The third phase creates the high-burst condition in which the resource-constrained VM-B reaches overload. The workload is simulated using the Task object available in PoolRunner, which represents the processing of a 1 KB file by sleeping for approximately 100 milliseconds. For each scheme, the same workload was executed separately using the same OCI deployment, VM configurations, request path, monitoring interval, and logging mechanism. The PoolRunner client generated the workload from the local machine, VM-A distributed requests to the backend TPS instances, and VM-B and VM-C processed the requests according to the selected scheme.

Result and Discussion:

To reduce the effect of workload randomness on the reported results, each scheme was executed independently five times using random seeds 0–4 under the same experimental configuration. For each run, measurements were collected from the constrained backend VM-B for the three workload phases described in the section "Experimental Workload Intensity and Burst Patterns". The values from the five runs were then summarized using the mean, standard deviation (SD), and 95% confidence interval (CI). Tables 1–5 report these statistics for thread-pool size, throughput, response time, CPU usage, and heap usage, respectively. Figure 5 provides the corresponding visual comparison using the mean values only. Therefore, the bars in Figure 5 should be interpreted together with the SD and 95% CI values reported in the respective tables. The discussion below follows the three workload phases. The first minute represents the normal workload, the second minute represents the burst workload, and the third minute represents the high-burst condition in which VM-B reaches the overload region. GACP activates its post-overload probing only after overload recovery. Its main behavior becomes visible after overload recovery in the third phase.

Thread-Pool Size and Adaptation Behavior:

Table 1 and Figure 5(a) compare the concurrency-control quantities used by the evaluated schemes. For GACP and DTP, the reported values are controller-level request-rate-driven pool-size settings and should not be interpreted as physical worker-thread counts directly equivalent to the fixed 150-thread pools used by Hazelcast, TopFull, and Rajomon. Sora, in contrast, reports its dynamically selected concurrency level.

In the second phase, the controller-level pool-size settings of both GACP and DTP increased to 1000.00, again with SD = 0.00. These values are expected because both schemes follow the request-rate-driven TPS model during normal operation, where the pool-size

setting remains parallel to the admitted request rate. No overload had yet occurred; therefore, the additional probing mechanism of GACP remained inactive in these two phases. Hazelcast, TopFull, and Rajomon maintained a fixed pool of 150 threads throughout all three phases. Consequently, their SD values and confidence-interval widths are zero in Table 1. The pool-size values should be interpreted according to each scheme's control model: for GACP and DTP they represent the controller-level request-rate-driven pool-size setting, whereas the fixed-pool baselines use configured worker-thread counts. The zero SD and zero-width confidence intervals for these three schemes result from their fixed 150-thread configuration across all runs.

Sora shows a different pattern because its concurrency level is dynamically adapted. Its mean pool size increased from 82.40 ± 5.13 threads in the first phase to 117.60 ± 10.95 in the second phase and 145.00 ± 4.64 in the third phase. The corresponding confidence intervals also show run-to-run variation in the selected concurrency levels. This is consistent with Sora's runtime concurrency-control behavior because the number of active worker threads is selected according to the observed latency and operating condition rather than being fixed before execution.

The most important difference between GACP and DTP appears in the third phase. DTP settled at an average pool-size setting of 1301.80 ± 2.39, whereas GACP reached 1352.40 ± 2.97. The 95% CI of GACP was [1348.72, 1356.08], while that of DTP was [1298.84, 1304.76]. The non-overlapping intervals show that the difference is considerably larger than the observed run-to-run variation. Both schemes first recover from overload using the same baseline DTP recovery mechanism, but DTP remains near its reduced operating point while GACP continues to probe the recovered backend. The larger third-phase pool-size setting of GACP therefore provides the first direct indication that the proposed probing mechanism is reclaiming additional backend capacity rather than leaving VM-B permanently restricted at the initial recovery level.

Table 1. Mean, standard deviation, and 95% confidence intervals of the controller-level pool-size setting for GACP/DTP and worker-thread pool size for the comparison schemes across the three workload phases.

Phase		GACP	DTP	Hazelcast	SORA	TopFull	Rajomon
1st minute	Mean ± SD	500.00 ± 0.00	500.00 ± 0.00	150.00 ± 0.00	82.40 ± 5.13	150.00 ± 0.00	150.00 ± 0.00
	95% CI (Lower)	500.00	500.00	150.00	76.03	150.00	150.00
	95% CI (Upper)	500.00	500.00	150.00	88.77	150.00	150.00
2nd minute	Mean ± SD	1000.00 ± 0.00	1000.00 ± 0.00	150.00 ± 0.00	117.60 ± 10.95	150.00 ± 0.00	150.00 ± 0.00
	95% CI (Lower)	1000.00	1000.00	150.00	104.01	150.00	150.00
	95% CI (Upper)	1000.00	1000.00	150.00	131.19	150.00	150.00
3rd minute	Mean ± SD	1352.40 ± 2.97	1301.80 ± 2.39	150.00 ± 0.00	145.00 ± 4.64	150.00 ± 0.00	150.00 ± 0.00
	95% CI (Lower)	1348.72	1298.84	150.00	139.24	150.00	150.00
	95% CI (Upper)	1356.08	1304.76	150.00	150.76	150.00	150.00

Note:

For GACP and DTP, the reported values represent the controller-level request-rate-driven pool-size setting, where $p_i(t) = \lambda_i(t)$; for Hazelcast, TopFull, and Rajomon, the values represent configured worker-thread counts. Sora reports its dynamically selected concurrency level.

Table 2. Mean, standard deviation, and 95% confidence intervals of throughput across the three workload phases.

Phase		GACP	DTP	Hazelcast	SORA	TopFull	Rajomon
1st minute	Mean ± SD	499.20 ± 0.84	498.00 ± 1.00	497.87 ± 1.39	498.00 ± 1.58	490.60 ± 1.14	489.80 ± 1.92
	95% CI (Lower)	498.16	496.76	496.14	496.04	489.18	487.41
	95% CI (Upper)	500.24	499.24	499.59	499.96	492.02	492.19
2nd minute	Mean ± SD	999.20 ± 0.84	998.60 ± 1.52	997.00 ± 2.12	995.00 ± 1.87	980.00 ± 1.58	969.80 ± 1.92
	95% CI (Lower)	998.16	996.72	994.37	992.68	978.04	967.41
	95% CI (Upper)	1000.24	1000.48	999.63	997.32	981.96	972.19
3rd minute	Mean ± SD	1352.40 ± 2.97	1300.20 ± 3.11	1491.60 ± 3.36	1451.60 ± 2.41	1409.00 ± 5.52	1400.20 ± 1.92
	95% CI (Lower)	1348.72	1296.33	1487.43	1448.61	1402.14	1397.81
	95% CI (Upper)	1356.08	1304.07	1495.77	1454.59	1415.86	1402.59

Table 3. Mean, standard deviation, and 95% confidence intervals of response time across the three workload phases.

Phase		GACP	DTP	Hazelcast	SORA	TopFull	Rajomon
1st minute	Mean ± SD	126.35 ± 1.23	125.35 ± 1.29	151.28 ± 1.22	181.08 ± 1.03	201.62 ± 0.95	211.13 ± 1.00
	95% CI (Lower)	124.82	123.32	149.77	179.80	200.43	209.89
	95% CI (Upper)	127.88	127.38	152.79	182.36	202.80	212.37
2nd minute	Mean ± SD	136.48 ± 1.35	137.42 ± 1.45	250.42 ± 1.76	211.66 ± 1.48	300.22 ± 1.09	302.41 ± 1.25
	95% CI (Lower)	134.80	135.13	248.24	209.83	298.87	300.85
	95% CI (Upper)	138.15	139.71	252.60	213.50	301.57	303.96
3rd minute	Mean ± SD	143.14 ± 1.05	150.59 ± 1.46	353.23 ± 2.08	241.74 ± 1.57	471.24 ± 1.24	481.22 ± 1.22
	95% CI (Lower)	141.84	148.78	350.65	239.79	469.70	479.70
	95% CI (Upper)	144.45	152.40	355.81	243.69	472.78	482.74

Table 4. Mean, standard deviation, and 95% confidence intervals of CPU usage across the three workload phases.

Phase		GACP	DTP	Hazelcast	SORA	TopFull	Rajomon
1st minute	Mean ± SD	71.87 ± 1.46	73.17 ± 1.16	45.77 ± 0.95	66.30 ± 1.23	45.86 ± 1.17	31.21 ± 1.61
	95% CI (Lower)	70.05	71.73	44.60	64.78	44.41	29.22
	95% CI (Upper)	73.68	74.61	46.95	67.82	47.31	33.21
2nd minute	Mean ± SD	74.72 ± 1.11	74.67 ± 0.92	53.54 ± 1.44	71.09 ± 1.10	56.49 ± 3.27	40.67 ± 1.17
	95% CI (Lower)	73.35	73.52	51.76	69.73	52.43	39.22
	95% CI (Upper)	76.10	75.82	55.33	72.45	60.54	42.13
3rd minute	Mean ± SD	79.89 ± 1.36	79.59 ± 1.95	60.07 ± 1.00	75.19 ± 1.13	54.85 ± 1.14	52.30 ± 1.69
	95% CI (Lower)	78.20	77.16	58.83	73.78	53.43	50.21
	95% CI (Upper)	81.58	82.02	61.32	76.60	56.27	54.40

Table 5. Mean, standard deviation, and 95% confidence intervals of heap usage across the three workload phases.

Phase		GACP	DTP	Hazelcast	SORA	TopFull	Rajomon
1st minute	Mean ± SD	45.46 ± 0.47	43.09 ± 0.69	25.51 ± 0.85	29.86 ± 1.12	32.39 ± 1.01	30.84 ± 0.48
	95% CI (Lower)	44.88	42.23	24.45	28.46	31.13	30.25
	95% CI (Upper)	46.04	43.94	26.57	31.25	33.64	31.43
2nd minute	Mean ± SD	48.95 ± 0.55	49.93 ± 0.55	43.87 ± 1.13	40.78 ± 0.77	41.10 ± 0.71	40.74 ± 1.00
	95% CI (Lower)	48.27	49.24	42.47	39.82	40.22	39.49
	95% CI (Upper)	49.63	50.61	45.27	41.73	41.99	41.99
3rd minute	Mean ± SD	53.90 ± 0.65	54.97 ± 0.60	52.56 ± 1.00	51.27 ± 0.86	51.04 ± 1.25	51.13 ± 0.49
	95% CI (Lower)	53.10	54.22	51.32	50.21	49.49	50.53
	95% CI (Upper)	54.71	55.73	53.80	52.34	52.59	51.74

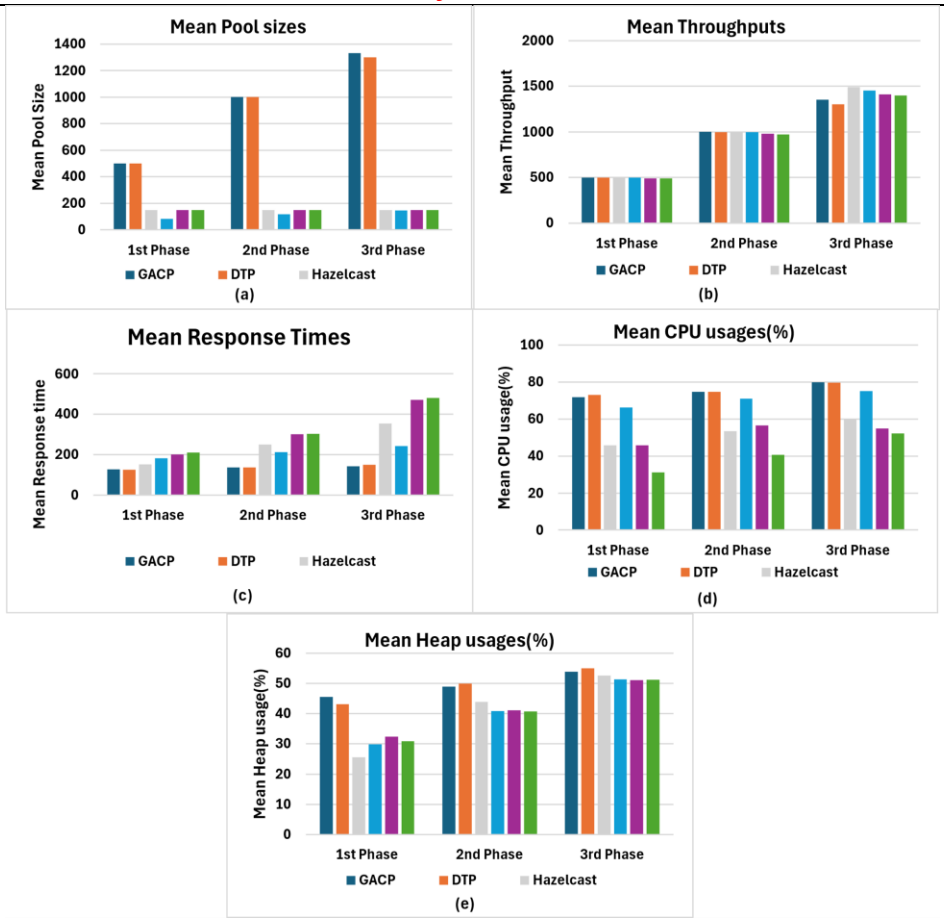

Figure 5. Phase-wise comparison of GACP and baseline schemes on VM-B: (a) mean thread-pool size that represents mean concurrency-control setting, (b) mean throughput, (c) mean response time, (d) mean CPU usage, and (e) mean heap usage across the three workload phases

Throughput Performance:

The phase-wise throughput results are reported in Table 2 and visually summarized in Figure 5(b). During the first phase, the schemes operated close to the approximately 500 RPS workload assigned to VM-B through round-robin distribution. GACP achieved 499.20 ± 0.84 RPS, DTP 498.00 ± 1.00 RPS, Hazelcast 497.87 ± 1.39 RPS, and Sora 498.00 ± 1.58 RPS. TopFull and Rajomon showed slightly lower means of 490.60 ± 1.14 and 489.80 ± 1.92 RPS, respectively. The small SD values indicate that these throughput patterns were reproduced consistently across the five runs.

A similar behavior was observed in the second phase. GACP produced 999.20 ± 0.84 RPS and DTP 998.60 ± 1.52 RPS, while Hazelcast and Sora produced 997.00 ± 2.12 and 995.00 ± 1.87 RPS, respectively. TopFull achieved 980.00 ± 1.58 RPS and Rajomon 969.80 ± 1.92 RPS. Thus, before VM-B entered the high-burst overload condition, GACP did not unnecessarily restrict normal backend operation. This behavior confirms that GACP remains transparent during stable operation and activates its additional probing behavior only after overload recovery.

The third phase provides the main throughput comparison. Hazelcast achieved the highest mean throughput at 1491.60 ± 3.36 RPS, followed by Sora at 1451.60 ± 2.41 RPS, TopFull at 1409.00 ± 5.52 RPS, Rajomon at 1400.20 ± 1.92 RPS, GACP at 1352.40 ± 2.97 RPS, and DTP at 1300.20 ± 3.11 RPS. Although Hazelcast, Sora, TopFull, and Rajomon achieved higher raw throughput in the third phase, GACP is designed specifically for

controlled post-overload capacity rediscovery rather than maximum throughput. That is, maximum raw throughput is not the objective of the proposed scheme. GACP is specifically designed to improve the post-overload recovery behavior of DTP by reclaiming additional post-overload capacity under the tested workload without immediately returning the backend to a previously unsafe operating point.

DTP provides the closest reference for evaluating the contribution of the proposed probing strategy because GACP extends its overload-recovery mechanism. Compared with DTP, GACP increased the third-phase mean throughput from 1300.20 to 1352.40 RPS, corresponding to an improvement of approximately 4.0%. The 95% CI of GACP was [1348.72, 1356.08], whereas DTP remained within [1296.33, 1304.07]. These intervals are clearly separated, showing that the additional throughput obtained by GACP was consistent across the five independent runs and was not caused by a single random execution.

The throughput differences are better understood together with the response-time and resource results because the compared schemes apply different workload-control mechanisms and operate at different performance points. These schemes use different control objectives and mechanisms. Hazelcast continues execution with a fixed executor, Sora adapts concurrency, TopFull controls request admission, and Rajomon applies decentralized rate control and load shedding. Their higher third-phase throughput represents different operating points. The main question for GACP is whether additional capacity can be reclaimed beyond DTP's conservative recovery point while preserving stable backend behavior. The throughput results show that this objective was achieved.

Response-Time Performance:

Table 3 and Figure 5(c) present the response-time results. Response time provides an important second dimension for interpreting the throughput results because a scheme that processes more requests may simultaneously expose those requests to considerably greater delay.

During the first phase, GACP and DTP showed very similar response-time behavior. GACP recorded 126.35 ± 1.23 ms, while DTP recorded 125.35 ± 1.29 ms. Hazelcast produced 151.28 ± 1.22 ms, while Sora, TopFull, and Rajomon recorded 181.08 ± 1.03, 201.62 ± 0.95, and 211.13 ± 1.00 ms, respectively. Since no overload occurred during this phase, GACP's post-overload probing remained inactive, and the small difference between GACP and DTP reflects normal run-to-run execution variation rather than the effect of the proposed probing mechanism.

A similar behavior was observed during the second phase. GACP recorded 136.48 ± 1.35 ms, while DTP recorded 137.42 ± 1.45 ms. Sora reached 211.66 ± 1.48 ms, Hazelcast 250.42 ± 1.76 ms, TopFull 300.22 ± 1.09 ms, and Rajomon 302.41 ± 1.25 ms. GACP probing was still inactive because VM-B had not yet entered the overload condition. Therefore, the small response-time difference between GACP and DTP in this phase represents normal experimental variation before activation of the proposed mechanism.

The effect of GACP therefore needs to be evaluated primarily in the third phase, where overload recovery occurs and the post-overload probing mechanism becomes active.

The high-burst third phase produces the clearest separation. GACP maintained an average response time of only 143.14 ± 1.05 ms, compared with 150.59 ± 1.46 ms for DTP. Sora reached 241.74 ± 1.57 ms and Hazelcast 353.23 ± 2.08 ms. TopFull and Rajomon operated much closer to the configured 500 ms SLO, with mean response times of 471.24 ± 1.24 and 481.22 ± 1.22 ms, respectively.

The GACP-DTP comparison directly shows the effect of post-overload capacity probing. GACP did not obtain its additional third-phase throughput by accepting a latency penalty. Instead, while increasing throughput by approximately 4.0% relative to DTP, its mean response time decreased from 150.59 to 143.14 ms, an improvement of approximately 4.95%.

Their 95% confidence intervals were also non-overlapping: [141.84, 144.45] for GACP and [148.78, 152.40] for DTP. This result supports the main motivation of the proposed method. The additional capacity recovered through probing was not obtained by aggressively forcing VM-B toward the previous overload point; rather, the backend remained in a comparatively low-latency operating region.

This result also explains that the higher throughput of other schemes came with higher response time. Hazelcast produced approximately 1492 RPS during the third phase, but its mean response time was more than twice that of GACP. Sora also achieved higher throughput but operated at a noticeably higher response time. TopFull and Rajomon achieved around 1400 RPS but their mean response times approached the 500 ms SLO boundary. GACP therefore occupies a different throughput-response-time operating point: it sacrifices some maximum achievable raw throughput in order to retain a safer recovered state with substantially lower request latency.

CPU Utilization:

Table 4 and Figure 5(d) compare the average CPU utilization of the schemes. CPU usage increased with workload for most schemes, although the magnitude of the increase depended on how each mechanism-controlled concurrency and admitted workload.

During the first phase, GACP used 71.87 ± 1.46% CPU and DTP 73.17 ± 1.16%. Sora used 66.30 ± 1.23%, whereas Hazelcast and TopFull remained close to 46%. Rajomon recorded the lowest value at 31.21 ± 1.61%. In the second phase, GACP and DTP increased to 74.72 ± 1.11% and 74.67 ± 0.92%, respectively. Sora reached 71.09 ± 1.10%, Hazelcast 53.54 ± 1.44%, TopFull 56.49 ± 3.27%, and Rajomon 40.67 ± 1.17%.

The highest workload phase increased GACP CPU utilization to 79.89 ± 1.36%. DTP was very close at 79.59 ± 1.95%, followed by Sora at 75.19 ± 1.13%. Hazelcast used 60.07 ± 1.00%, TopFull 54.85 ± 1.14%, and Rajomon 52.30 ± 1.69%.

These results do not mean that lower CPU usage is always better. GACP actively attempts to reclaim processing capacity on the constrained VM-B after recovery. Consequently, greater use of VM-B's available CPU is expected when additional work is successfully returned to the backend. In the third phase, GACP achieved approximately 52 RPS more throughput than DTP while their CPU utilizations remained very close, 79.89% and 79.59%, respectively. This suggests that the additional recovered workload did not require a proportionally large increase in CPU consumption.

The lower CPU utilization of TopFull and Rajomon is also consistent with their workload-regulation behavior. These mechanisms may restrict, throttle, or shed requests rather than attempting to utilize the backend in the same way as GACP. Therefore, their CPU results should be considered together with their throughput and response-time results rather than interpreted as direct evidence of superior resource efficiency.

Another observation from Table 4 is that most CPU measurements show small SD values across the five runs. TopFull exhibits comparatively higher variability during the second phase, with 56.49 ± 3.27%, but its third-phase variation becomes much smaller. Overall, the CPU results indicate repeatable phase-wise behavior across the independent tests.

Heap-Memory Utilization:

Heap-memory results are presented in Table 5 and Figure 5(e). Unlike throughput and response time, the differences among schemes become relatively small under the highest workload phase.

During the first phase, GACP used 45.46 ± 0.47% heap and DTP 43.09 ± 0.69%. Hazelcast recorded the lowest value of 25.51 ± 0.85%, while Sora, TopFull, and Rajomon used 29.86 ± 1.12%, 32.39 ± 1.01%, and 30.84 ± 0.48%, respectively.

In the second phase, the heap usage of GACP increased to 48.95 ± 0.55% and DTP to 49.93 ± 0.55%. Hazelcast reached 43.87 ± 1.13%, while Sora, TopFull, and Rajomon remained around 41%, with means of 40.78, 41.10, and 40.74%, respectively.

In the high-burst third phase, the values converged considerably. GACP used 53.90 ± 0.65%, DTP 54.97 ± 0.60%, Hazelcast 52.56 ± 1.00%, Sora 51.27 ± 0.86%, TopFull 51.04 ± 1.25%, and Rajomon 51.13 ± 0.49%. Thus, although GACP reclaimed additional workload from the constrained backend, its heap usage remained in the same general range as the comparison schemes. This indicates that capacity rediscovery did not introduce disproportionate memory growth. In fact, GACP's third-phase heap usage was slightly lower than DTP despite its higher recovered throughput. The relatively narrow SD and confidence intervals also show that memory behavior remained stable across the five independent executions.

Cross-Metric Analysis of the High-Burst Phase:

The third workload phase is the main condition for evaluating the proposed contribution because GACP is a post-overload recovery and capacity-rediscovery mechanism. During the first two phases, VM-B did not reach the overload condition required to activate the GACP probing procedure. This shows that GACP does not unnecessarily alter the baseline scheduling behavior when the backend remains stable.

The situation changes in the third phase when the total workload reaches approximately 3000 RPS and the constrained VM-B enters the overload region. DTP performs its initial overload recovery by reducing the workload assigned to VM-B. However, after recovery, it remains close to the reduced operating point. This behavior is visible in both Table 1 and Table 2, where DTP settles at a mean pool-size setting of 1301.80 and throughput of 1300.20 RPS.

GACP begins with the same recovery mechanism but treats recovery as the starting point for further capacity exploration. When VM-B remains stable, the admitted-rate limit is gradually increased according to the probing granularity. If another overload point is encountered, that value becomes an unsafe boundary and the probing step is refined. This mechanism results in a third-phase mean pool-size setting of 1352.40 and throughput of 1352.40 RPS. Therefore, compared with its closest baseline, GACP recovers approximately 52 additional RPS and provides about a 4.0% throughput improvement.

More importantly, this additional throughput is achieved without degrading response time. GACP records 143.14 ms compared with 150.59 ms for DTP. CPU usage remains almost identical between them, at 79.89% and 79.59%, while heap usage is also comparable, at 53.90% and 54.97%. Considering these metrics together, the benefit of GACP is not simply that it sends more requests to VM-B. Rather, it uses the recovered backend more effectively without pushing it immediately back toward the unsafe overload condition.

The other schemes use different control mechanisms and therefore show different trade-offs. Hazelcast achieves the highest raw throughput in the third phase, but its response time rises to 353.23 ms. Sora reaches 1451.60 RPS with a response time of 241.74 ms. TopFull and Rajomon achieve 1409.00 and 1400.20 RPS, respectively, but their response times increase to 471.24 and 481.22 ms, close to the configured 500 ms SLO. In contrast, GACP maintains a much lower response time of 143.14 ms. The throughput–latency trade-off is clear from the third-phase results. GACP's throughput was approximately 9.33%, 6.83%, 4.02%, and 3.41% lower than Hazelcast, Sora, TopFull, and Rajomon, respectively. However, GACP reduced response time by 59.48%, 40.79%, 69.62%, and 70.25% compared with the same schemes. Therefore, GACP does not provide the highest raw throughput, but it operates at a different point where a small reduction in throughput provides a much larger reduction in response time. In comparison with its direct DTP baseline, GACP improves both metrics by increasing throughput by approximately 4.0% and reducing response time by approximately 4.95%.

Accordingly, GACP focuses on post-overload capacity rediscovery rather than maximum throughput. Its advantage is more specific: GACP improves the post-overload throughput of its direct DTP baseline under the evaluated workload while maintaining a considerably lower response time than the higher-throughput comparative schemes. In this sense, GACP provides a different throughput-latency trade-off. It avoids simply maximizing the amount of work admitted to VM-B and instead attempts to recover additional capacity in controlled steps while respecting previously observed unsafe operating points. When considering TopFull and Rajomon, the lower CPU usage does not necessarily mean that they use VM-B more effectively. Their control policies intentionally regulate or shed part of the offered workload to protect latency-sensitive execution. Similarly, Sora primarily adapts concurrency rather than searching for post-overload capacity in the same manner as GACP. DTP remains the closest baseline for evaluating GACP.

The results consequently answer the research question posed in the Introduction. Under the evaluated workload conditions, additional post-overload capacity can be reclaimed after backend recovery. GACP achieves this by probing above the recovery point while treating newly observed overload levels as unsafe boundaries. In the current experiments, this strategy allowed GACP to recover additional VM-B throughput beyond DTP's conservative operating point while maintaining low response time and comparable resource consumption.

Statistical Significance Analysis:

The mean, SD, and 95% confidence intervals reported in Tables 1–5 provide the repeated-run descriptive analysis for all three workload phases. In addition to these descriptive statistics, the third phase was selected for targeted inferential analysis because it represents the high-burst condition in which VM-B reaches overload and the proposed GACP recovery and capacity-probing mechanism becomes active. In the first and second phases, GACP remains in normal operation because no overload is observed. Therefore, the third phase provides the most relevant condition for determining whether the differences observed between GACP and the comparison schemes are statistically significant.

Since the same five random seeds (0–4) were used for GACP and every comparative scheme, a two-sided paired t-test was applied. The pairing was performed between runs generated with the same workload seed, so that each GACP run was compared with the corresponding baseline run under the matched workload realization.

The five paired observations come from five independent workload realizations generated using seeds 0–4. Therefore, each GACP run is paired with the corresponding baseline run under the same stochastic workload, while different seed pairs remain independent. The paired t-test assumes continuous paired differences, independence among pairs, and an approximately normal distribution of the differences without severe outliers. Since only five paired runs are available, normality assessment has limited power. Therefore, the inferential results are interpreted together with the mean differences and 95% confidence intervals rather than as standalone evidence. The value n=5 directly represents the five independent seeded runs conducted for each scheme under the same experimental protocol. For each seed, the paired difference was calculated as:

$$d_i = X_{GACP,i} - X_{Baseline,i}$$

The mean paired difference was then calculated as:

$$\bar{d} = (1/n) \Sigma_{i=1}^{n} d_i$$

The corresponding paired t-statistic was calculated as:

$$t = \bar{d} / (s_d / \sqrt{n})$$

where s_d is the sample standard deviation of the paired differences. Since five matched seeded runs were available for each GACP–baseline comparison, $n = 5$ paired observations were used and the test has four degrees of freedom. The null and alternative hypotheses were defined as:

$$H_0: \mu_d = 0$$
$$H_1: \mu_d \neq 0$$

The difference was considered statistically significant when $p < 0.05$. The 95% confidence interval of the paired difference was calculated as:

$$95\%\ CI = \bar{d} \pm t_{0.975,4}\ (s_d / \sqrt{5})$$

A confidence interval that does not include zero provides further evidence that the observed paired difference is statistically significant. The difference is defined throughout Table 6 as GACP minus the corresponding baseline. Therefore, for throughput a positive difference means that GACP achieved higher throughput, whereas for response time, CPU usage, and heap usage a negative difference means that GACP produced a lower value.

The very large t-statistics observed for some response-time comparisons result from large mean paired differences combined with very small variability in those differences across the five matched seeds. Because the sample size is limited to five paired runs, these results are interpreted together with the corresponding effect sizes and 95% confidence intervals rather than from the magnitude of the t-statistic alone.

The inferential results support the main observations already visible in Tables 1–5 and Figure 5. Most importantly, GACP achieved a statistically significant throughput improvement over its direct DTP baseline. The mean paired improvement was 52.20 RPS, with a 95% CI of [51.16, 53.24] RPS and $p < 0.001$.

Table 6. Paired t-test results for GACP against the comparative schemes in the high-burst third phase across five seeded runs.

Metric	Comparison	Mean Difference	t-statistic	p-value	95% CI of Difference	Result
Throughput	GACP vs DTP	+52.20 RPS	139.510	<0.001	[51.16, 53.24]	Significant
Throughput	GACP vs Hazelcast	−139.20 RPS	−57.601	<0.001	[−145.91, −132.49]	Significant
Throughput	GACP vs Sora	−99.20 RPS	−134.994	<0.001	[−101.24, −97.16]	Significant
Throughput	GACP vs TopFull	−56.60 RPS	−15.783	<0.001	[−66.56, −46.64]	Significant
Throughput	GACP vs Rajomon	−47.80 RPS	−22.945	<0.001	[−53.58, −42.02]	Significant
Response time	GACP vs DTP	−7.45 ms	−40.684	<0.001	[−7.96, −6.94]	Significant
Response time	GACP vs Hazelcast	−210.09 ms	−399.604	<0.001	[−211.55, −208.63]	Significant
Response time	GACP vs Sora	−98.60 ms	−419.283	<0.001	[−99.25, −97.95]	Significant
Response time	GACP vs TopFull	−328.10 ms	−2358.053	<0.001	[−328.49, −327.71]	Significant
Response time	GACP vs Rajomon	−338.08 ms	−1979.593	<0.001	[−338.55, −337.60]	Significant
CPU usage	GACP vs DTP	+0.30%	0.996	0.376	[−0.53, 1.13]	Not significant

This corresponds closely to the approximately 4.0% improvement observed from the phase-wise mean values in Table 2. The confidence interval remains entirely above zero, showing that the additional capacity recovered by GACP was reproduced consistently across all five seeded runs.

At the same time, the additional throughput over DTP did not produce a response-time penalty. The paired response-time difference between GACP and DTP was −7.45 ms, with a 95% CI of [−7.96, −6.94] ms and $p < 0.001$. Thus, GACP provided both statistically higher throughput and statistically lower response time than the fixed-recovery behavior of DTP in the phase where the proposed probing mechanism was active. This result directly supports the main objective of GACP: recovering additional post-overload backend capacity under the evaluated workload conditions rather than permanently retaining the conservative DTP recovery limit.

The throughput comparisons with the remaining schemes provide a different but equally notable result. Hazelcast, Sora, TopFull, and Rajomon achieved statistically higher raw throughput than GACP in the third phase, as indicated by the negative GACP-minus-baseline differences in Table 6. However, Table 3, Figure 5(c), and the paired tests show that GACP simultaneously maintained significantly lower response time than every one of these schemes. The mean response-time advantage was approximately 98.60 ms over Sora, 210.09 ms over Hazelcast, 328.10 ms over TopFull, and 338.08 ms over Rajomon, with $p < 0.001$ in every case and with none of the corresponding 95% confidence intervals including zero.

These results statistically support the throughput–latency trade-off identified in the previous subsections. The higher-throughput comparative schemes operate at different points of this trade-off, whereas GACP intentionally avoids aggressively returning VM-B toward a previously unsafe overload region. The statistical results support GACP's post-overload capacity-rediscovery objective.

The resource results further support this interpretation. The paired CPU difference between GACP and DTP was only +0.30 percentage points and was not statistically significant ($p = 0.376$); its 95% CI [−0.53, 1.13] also includes zero. Therefore, the approximately 52 RPS additional throughput recovered by GACP cannot be associated with a statistically significant increase in average CPU usage over DTP in these five runs. In contrast, GACP showed a statistically significant reduction of approximately 1.07 percentage points in heap usage relative to DTP ($p < 0.001$, 95% CI [−1.30, −0.84]).

Taken together, the paired analysis strengthens the answer to the research question. Under the high-burst workload, GACP recovered statistically more throughput than its direct DTP baseline while simultaneously achieving statistically lower response time, without a statistically significant increase in CPU usage. Although several other schemes achieved statistically higher raw throughput, this was accompanied by significantly higher response time. The experimental evidence therefore supports GACP as a post-overload capacity-rediscovery mechanism rather than a maximum-throughput mechanism: it uses controlled probing to reclaim capacity beyond the fixed DTP recovery point while retaining a comparatively low-latency operating condition.

Limitations and Practical Implications:

The present evaluation has some limitations that should be considered when interpreting the results. First, the distributed testbed contains two heterogeneous backend VMs, while the detailed quantitative analysis focuses on the resource-constrained VM-B. This setup is suitable for studying the post-overload capacity-rediscovery behavior of GACP, but it does not represent larger distributed deployments with many heterogeneous backend nodes. Second, each experimental run lasts 180 seconds and contains three consecutive workload phases. This controlled duration allows overload recovery and probing behavior to be observed under comparable conditions, but longer experiments may reveal slower resource effects, background changes, or repeated overload-recovery cycles that are not captured here. Third, the evaluation uses five independently seeded runs. These runs support descriptive and paired statistical analysis, but the small sample size limits broader generalization across a wider range of workload realizations. Finally, the detailed results are obtained from a single

constrained backend configuration with 2 OCPUs and 4 GB RAM. Therefore, the observed overload point and recovered capacity depend on this backend configuration and workload characteristics and should not be considered universal thresholds.

From a practical perspective, GACP is most suitable for distributed thread-pool systems where a backend can report overload, its admitted request rate can be controlled, and previously observed unsafe operating points can be retained. The scheme complements overload recovery by gradually reclaiming capacity instead of immediately restoring the previously overloaded rate. In larger or more heterogeneous deployments, GACP can be maintained separately for each backend because the admitted-rate limit, unsafe boundary, and probing granularity are backend-specific. However, further evaluation is required under larger clusters, longer workloads, different backend capacities, mixed service times, and more frequent workload changes.

Conclusion:

This study presented Granularity-Aware Capacity Probing (GACP) for post-overload capacity rediscovery in distributed systems with thread-pool backends. The main objective was to address the situation in which a backend recovers from overload but remains at a conservative admitted-rate limit even though some additional post-overload capacity may still be available under the current workload conditions. GACP extends the recovery process by gradually probing above the recovered operating point, while newly observed overload points are retained as unsafe boundaries and the probing granularity is refined when required. The scheme was evaluated using PoolRunner on Oracle Cloud Infrastructure with one load-balancer VM and two heterogeneous backend VMs. GACP was compared with DTP, Hazelcast-DES, Sora, TopFull, and Rajomon under normal, burst, and high-burst workload phases. Each scheme was executed through five independent runs using different random seeds, and the results were analyzed using mean, standard deviation, 95% confidence intervals, and paired statistical testing for the high-burst phase. The results show that GACP remained transparent during the first two workload phases and became active after overload recovery in the high-burst phase. In this phase, GACP increased the constrained-backend throughput from 1300.20 RPS with DTP to 1352.40 RPS, giving an improvement of approximately 4.0%. At the same time, its mean response time decreased from 150.59 ms to 143.14 ms. GACP maintained low response time during the first two workload phases and achieved the lowest mean response time among the compared schemes in the high-burst phase. In the high-burst phase, the response-time reduction was 4.95% compared with DTP, 40.79% compared with Sora, 59.48% compared with Hazelcast-DES, 69.62% compared with TopFull, and 70.25% compared with Rajomon. The statistical analysis further confirmed the main high-burst findings. GACP achieved statistically higher throughput and lower response time than DTP, while the increase in CPU usage was not statistically significant. The broader comparison also showed that some schemes achieved higher raw throughput than GACP, but this was accompanied by substantially higher response times. Therefore, the results indicate that GACP provides a favorable throughput–latency trade-off, improving both throughput and response time over its closest DTP baseline while maintaining substantially lower response time than the higher-throughput comparative schemes. Overall, the study shows that overload recovery does not need to end at a fixed reduced operating point. Controlled probing can continue after recovery and rediscover additional post-overload backend capacity under the evaluated workload conditions. GACP provides this capability through gradual admitted-rate probing, unsafe-boundary control, and adaptive probing granularity. Future work can evaluate the scheme under longer workloads, larger distributed deployments, more heterogeneous backend configurations, mixed service times, and adaptive probing policies under changing runtime conditions.

Acknowledgement:
The authors gratefully acknowledge Hazara University, Mansehra, for its academic support and for facilitating access to the cloud-computing resources used in this study.

Author's Contribution:

Faisal Bahadur: Conceptualization, Methodology, Software, Writeup and Supervision. Faqir Gul: Design & Analysis. Arooba Akhtar: Experimental Deployment and Configurations. Nadir Hussain: Data Curation, Investigation and Validation. Abdul Qadir: Testing, Evaluation and Analysis.

Conflict of Interest:

The authors declare no conflict of interest.

References:

[1] M. Nashaat, W. Moussa, R. Rizk, and W. Saber, "Dynamic machine learning approach for workload prediction in cloud environments," *Sci. Reports 2026 161*, vol. 16, no. 1, pp. 10983-, Apr. 2026, doi: 10.1038/s41598-026-40777-z.

[2] Rossi, A., Visentin, A., Carraro, D. et al., "Forecasting workload in cloud computing: towards uncertainty-aware predictions and transfer learning," *Clust. Comput*, 2025, doi: https://doi.org/10.1007/s10586-024-04933-2.

[3] H. Zhang, T. Guo, W. Tian, and H. Ma, "Learning-driven hybrid scaling for multi-type services in cloud," *J. Parallel Distrib. Comput.*, vol. 189, p. 104880, Jul. 2024, doi: 10.1016/J.JPDC.2024.104880.

[4] I. S. Makarov, D. V. Larin, E. G. Vorobeva, D. P. Emelin, and D. A. Kartashov, "The impact of asynchronous and multithreaded query processing models on the performance of server-side web applications," *Программные системы и вычислительные методы*, no. 1, pp. 13–20, Jan. 2025, doi: 10.7256/2454-0714.2025.1.73665.

[5] S. Shanjai Kumar, B. N. Sanjai, S. Etheeswar Kaarthi, V. Sivakumar, and S. Jagadeesan, "Impact of virtual threads and garbage collection on energy efficiency of Java applications for battery powered IoT devices," *Sci. Reports 2026 161*, vol. 16, no. 1, pp. 13507-, Mar. 2026, doi: 10.1038/s41598-026-40112-6.

[6] F. Bahadur, A. I. Umar, I. Ullah, F. Algarni, M. A. Khan, and S. M. Mostafa, "A Perfect Knob to Scale Thread Pool on Runtime," *Comput. Mater. Contin.*, vol. 72, no. 1, pp. 1483–1493, Feb. 2022, doi: 10.32604/CMC.2022.024936.

[7] D. Casini, "A Theoretical Approach to Determine the Optimal Size of a Thread Pool for Real-Time Systems," *Proc. - Real-Time Syst. Symp.*, vol. 2022-December, pp. 66–78, 2022, doi: 10.1109/RTSS55097.2022.00016.

[8] J. Alcaraz *et al.*, "Predicting number of threads using balanced datasets for openMP regions," *Comput. 2022 1055*, vol. 105, no. 5, pp. 999–1017, Apr. 2022, doi: 10.1007/S00607-022-01081-6.

[9] Z. Zhao and D. Mukherjee, "Optimal rate-matrix pruning for large-scale heterogeneous systems," *Queueing Syst. 2026 1101*, vol. 110, no. 1, pp. 16-, Feb. 2026, doi: 10.1007/S11134-026-09971-8.

[10] Z. Zhao, D. Mukherjee, and R. Wu, "Exploiting Data Locality to Improve Performance of Heterogeneous Server Clusters," *Stoch. Syst.*, vol. 14, no. 3, pp. 229–272, Nov. 2022, doi: 10.1287/stsy.2022.0040.

[11] D. Cotroneo, R. Natella, and S. Rosiello, "DRACO: Distributed Resource-aware Admission Control for large-scale, multi-tier systems," *J. Parallel Distrib. Comput.*, vol. 192, p. 104935, Oct. 2024, doi: 10.1016/J.JPDC.2024.104935.

[12] F. Bahadur, A. I. Umar, I. Ullah, F. Algarni, and M. A. Khan, "The Double Edge Sword Based Distributed Executor Service," *Comput. Syst. Sci. Eng.*, vol. 42, no. 2, pp. 589–604, Jan. 2022, doi: 10.32604/CSSE.2022.022319.

[13] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A Latency Sensitive Approach for Microservice Soft Resource Adaptation," *Middlew. 2023 - Proc. 24th ACM/IFIP Int.*

Middlew. Conf., pp. 43–56, Nov. 2023, doi: 10.1145/3590140.3592851.

[14] J. Park, J. Park, Y. Jung, H. Lim, H. Yeo, and D. Han, "TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices," *ACM SIGCOMM 2024 - Proc. 2024 ACM SIGCOMM 2024 Conf.*, pp. 876–890, Aug. 2024, doi: 10.1145/3651890.3672253;

[15] B. Wan *et al.*, *Rajomon: Decentralized and Coordinated Overload Control for {Latency-Sensitive} Microservices*. 2025. Accessed: Aug. 29, 2026. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[16] R. Bhattacharya, Y. Gao, and T. Wood, "Dynamically Balancing Load with Overload Control for Microservices," *ACM Trans. Auton. Adapt. Syst.*, vol. 19, no. 4, Nov. 2024, doi: 10.1145/3676167.

[17] N. Singh *et al.*, "Load balancing and service discovery using Docker Swarm for microservice based big data applications," *J. Cloud Comput. 2023 121*, vol. 12, no. 1, pp. 4-, Jan. 2023, doi: 10.1186/S13677-022-00358-7.

[18] R. Erdei and L. Toka, "Minimizing Resource Allocation for Cloud-Native Microservices," *J. Netw. Syst. Manag. 2023 312*, vol. 31, no. 2, pp. 35-, Feb. 2023, doi: 10.1007/S10922-023-09726-3.

[19] X. Hu *et al.*, "Distributed Rate Limiting Under Decentralized Cloud Networks," *IEEE Trans. Mob. Comput.*, vol. 25, no. 2, pp. 2774–2792, 2026, doi: 10.1109/TMC.2025.3610314.

[20] Hazelcast, "Distributed executor service," *Hazelcast Platf. Doc.*, 2026, [Online]. Available: https://docs.hazelcast.com/hazelcast/latest/computing/executor-service.

[21] J. Park, B. Choi, C. Lee, and D. Han, "Graph Neural Network-Based SLO-Aware Proactive Resource Autoscaling Framework for Microservices," *IEEE/ACM Trans. Netw.*, vol. 32, no. 4, pp. 3331–3346, 2024, doi: 10.1109/TNET.2024.3393427.

[22] J. P. K. S. Nunes, S. Nejati, M. Sabetzadeh, and E. Y. Nakagawa, "Self-adaptive, Requirements-driven Autoscaling of Microservices," *Proc. - 2024 IEEE/ACM 19th Symp. Softw. Eng. Adapt. Self-Managing Syst. SEAMS 2024*, pp. 168–174, Apr. 2024, doi: 10.1145/3643915.3644094.

[23] L. Zanussi, D. Tessera, L. Massari, B. Bermejo, C. Juiz, and M. Calzarossa, "Load-aware predictive auto-scaling framework for cloud environments," *Clust. Comput. 2026 293*, vol. 29, no. 3, pp. 197-, Mar. 2026, doi: 10.1007/S10586-026-05944-X.

[24] F. Bahadur, Z. Ahmad, and A. Algarni, "PoolRunner: An Extensible Performance Testing Simulation Tool for Thread-Pool Middleware," *IEEE Access*, vol. 13, pp. 96660–96680, 2025, doi: 10.1109/ACCESS.2025.3575437.

[25] "Oracle Cloud Infrastructure." Accessed: Aug. 29, 2026. [Online]. Available: https://docs.oracle.com/en-us/iaas/Content/services.htm

[26] "Compute Shapes." Accessed: Aug. 14, 2026. [Online]. Available: https://docs.oracle.com/enus/iaas/Content/Compute/References/computeshapes.htm

[27] ORACLE, "Overview (Java SE 17 & JDK 17)." Accessed: Sep. 07, 2026. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/

[28] "Networking Overview." Accessed: Aug. 14, 2026. [Online]. Available: https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/overview.htm

[29] R. Ahmed, S. Ren, E. Kim, and C. Lee, "Efficient service mesh traffic management for cloud-native applications," *PLoS One*, vol. 21, no. 3, p. e0344516, Mar. 2026, doi: 10.1371/JOURNAL.PONE.0344516.

Copyright © by authors and 50Sea. This work is licensed under Creative Commons Attribution 4.0 International License.