Resource Management in Aurora Serverless

Bradley Barnhart
Amazon Web Services
brabarnh@amazon.com

Marc Brooker
Amazon Web Services
mbrooker@amazon.com

Daniil Chinenkov
Amazon Web Services
daniichi@amazon.com

Tony Hooper
Amazon Web Services
thoop@amazon.com

Jihoun Im
Amazon Web Services
ijihoun@amazon.com

Prakash Chandra Jha
Amazon Web Services
jhapraks@amazon.com

Tim Kraska
Amazon Web Services and MIT
timkrask@amazon.com

Ashok Kurakula
Amazon Web Services
ashkur@amazon.com

Alexey Kuznetsov
Amazon Web Services
alexey@amazon.com

Grant McAlister
Amazon Web Services
grant@amazon.com

Arjun Muthukrishnan
Amazon Web Services
mutarjun@amazon.com

Aravinthan Narayanan
Amazon Web Services
aravnar@amazon.com

Douglas Terry
Amazon Web Services
terdoug@amazon.com

Bhuvan Urgaonkar
Amazon Web Services and Penn State
urgaonkb@amazon.com

Jiaming Yan
Amazon Web Services
jiamingy@amazon.com

ABSTRACT

Amazon Aurora Serverless is an on-demand, autoscaling configuration for Amazon Aurora with full MySQL and PostgreSQL compatibility. It automatically offers capacity scale-up/down (i.e., vertical scaling) based on a customer database application's needs. For customers with time-varying workloads, it offers cost savings compared to provisioned Aurora or other alternatives due to its agile and granular scaling and its usage-based charging model. This paper describes the key ideas underlying Aurora Serverless's resource management. To help meet its goals, Aurora Serverless adapts and fine tunes well-established ideas related to resource over-subscription; reactive control informed by recent measurements; distributed & hierarchical decision-making; and innovations in the DB engine, OS, and hypervisor for efficiency. Perhaps the most challenging goal is to offer a consistent resource elasticity experience while operating hosts at high degrees of utilization. Aurora Serverless implements several novel ideas for striking a balance between these opposing needs. Its technique for mapping workloads to hosts ensures that, in the common case, there is adequate spare capacity within a host to support fast scale-up for a workload. In the rare event this is not so, it live migrates workloads to ensure seamless scale-up. Its load distribution strategy is characterized by "unbalancing" of load across hosts to enable agile live migrations. Finally, it employs a token bucket-based rate regulation mechanism to prevent a growing workload from saturating its host faster than live migration-based remedial actions.

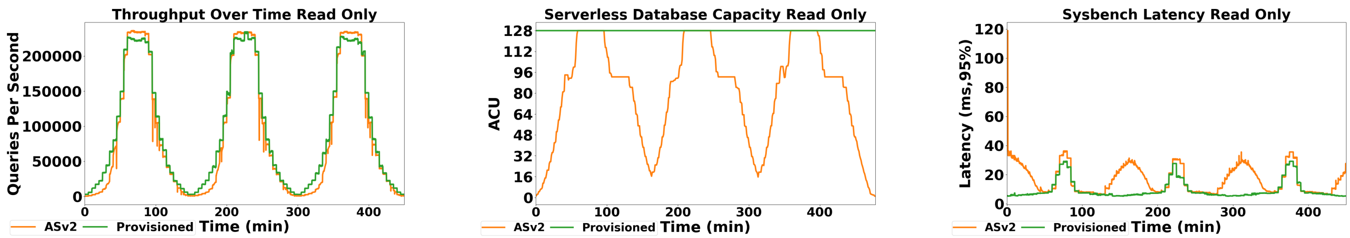
PVLDB Reference Format:
Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. Resource Management in Aurora Serverless. PVLDB, 17(12): 4038 - 4050, 2024.
doi:10.14778/3685800.3685825

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685825

1 INTRODUCTION AND MOTIVATION

Amazon Aurora [2] is a modern relational database service that offers performance and high availability guarantees at scale for fully open-source MySQL- and PostgreSQL-compatible editions [37]. The original *provisioned* Aurora offering, which came out in 2014, allows the customers to choose on-demand instances (virtual machines) and pay for their database (DB) by the hour with no long-term commitments or upfront fees, or choose reserved instances for additional savings [3]. More recently, Amazon has started offering Aurora Serverless [4], an on-demand, autoscaling configuration for Amazon Aurora. The autoscaling capability offered by Aurora is scale-up/down (i.e., "vertical" scaling of the resources allocated to a single DB instance) as opposed to scale-out ("horizontal" scaling) offered by some other systems. Aurora Serverless aims to to scale DB workloads fast, from hundreds to hundreds-of-thousands of transactions per second. The first Aurora Serverless offering (ASv1) came out in Aug. 2018 while the latest offering (ASv2) was released in Apr. 2022.

Why Aurora Serverless? The key selling point of Aurora Serverless is that it largely relieves the customers of having to manage how the resource capacity of their DBs varies in response to dynamic workloads. Instead of choosing a particular instance size or configuration up front, customers only specify DB capacity in units called *Aurora Capacity Units (ACUs)* [5]. Each ACU is a combination of 2 GB of memory, corresponding CPU, networking, and block device I/O throughput. Aurora Serverless scales each writer or reader in the customer's DB cluster within the customer-specified (minimum,



(a) Throughput (QPS) **(b) ACU allocation** **(c) P95 query latency**

Figure 1: A comparison of Aurora provisioned and ASv2 for a dynamic workload.

maximum) ACU range continually and automatically based on application demand.[1] Aurora Serverless is appealing to a diversity of customers vs. provisioned Aurora or other alternatives due to the cost savings and convenience resulting from the following reasons:

(1) It reduces the effort for planning DB instance sizes and resizing DB instances as the workload changes.
(2) It helps customers avoid over-provisioning DB instances. It adds resources in granular increments when DB instances scale up. The customer pays only for used DB resources.
(3) It scales compute and memory capacity as needed, with no disruption to client transactions or overall workload.
(4) It relieves the customers from having to manage virtual machine types, which is necessary due to deprecations, capacity constraints, lack of support for older types in new AWS regions, etc. It also removes the need to commit to a certain instance type, which is often necessary for discounted pricing.

Generally speaking, customers that find Aurora Serverless appealing include: (i) workloads with high temporal variability (either predictable with patterns like time-of-day effects [11, 12] or unpredictable); and (ii) new applications with as-yet-unknown needs.

To illustrate the capabilities of Aurora Serverless, in Figure 1, we compare a dynamic workload using provisioned vs. Aurora Serverless v2 (ASv2). Our workload is based on a read-only configuration of the sysbench benchmark [9] with maximum and minimum capacities of 128 and 0.5 ACUs for the serverless scenarios and 128 ACU for provisioned. Figure 1(a) shows the highly dynamic nature of our workload with the throughput in queries per sec (QPS) varying in the range of close to 0 to more than 200,000 with multiple scale-ups and downs. In Figure 1(b), we show how ASv2 varies its ACU allocations (strictly speaking a quantity called the "reserved ACU" which we will describe later in the paper) in response to workload dynamism. Finally, in Figure 1(c), we compare the query latency (specifically P95 of latency measured over 10 sec windows) across the 2 configurations.

We find the following observations noteworthy. With provisioned our workload incurs the cost of 128 ACUs throughout - even outside of peak times. ASv2 tracks the workload closely and is able to allocate enough to handle spikes, gradually grow in lockstep with the workload and then similarly track workload on its way down. Finally, ASv2 is able to match the latency offered by provisioned to a large extent (the biggest deviations are during periods when the ACU allocations drop to their smallest values) while only needing 54.9% of the total ACU-hours allocated in the provisioned scenario. Generally, ASv2's latency gap vs. provisioned is the highest when our instance's ACU allocation has been scaled down to less than about 50% of its maximum - these are operating regimes where resource contention with other co-located instances has a relatively larger effect on the performance of our instance.

Contributions: Our goal in this paper is to describe the current state of Aurora Serverless's resource management strategies that allow it to offer its customers the resource elasticity illustrated above. We also wish to highlight salient lessons learnt during our progress from ASv1 to ASv2. This journey has involved carefully combining existing best practices with new ideas. We describe how our strategies were informed by our evolving understanding of the needs and behavior of our customers. Finally, our experience so far suggests some promising future directions which the paper also discusses. Our paper touches upon the following aspects of resource management in Aurora Serverless:

- *Capacity Bounds:* Aurora Serverless allows its customers to specify minimum and maximum bounds on their needs in terms of ACUs. Aurora Serverless guarantees that the resource allocation for a customer varies within these bounds in response to its dynamically evolving needs and that the customer experiences a predictable scale-up experience and usage-based pricing within these bounds.
- *Resource Over-subscription and Over-provisioning:* Aurora Serverless employs both over subscription and over provisioning of the resources on its hosts for different reasons. The number of vCPUs on a host may exceed the capacity of the physical CPUs they are mapped to for statistical multiplexing benefits. Similarly, memory capacity is oversubscribed in terms of the customer-specified max ACUs of the instances on a host. Doing so involves a complex trade-off between its own cost/utilization, on the one hand, and customer experience (resource contention among co-located instances and limits on scaling up), on the other hand. At the same time, the ACU allocations for instances are typically kept slightly higher than their current usage - this "scaling band" enables quick detection and meeting of scaling up needs while the allocations are being adjusted.

[1] With Aurora Serverless, as with provisioned clusters, storage capacity and compute capacity are separate. When we refer to Aurora Serverless capacity and scaling, it's always compute capacity that's increasing or decreasing.

- *Dynamic instance packing via live migration:* Aurora Serverless uses *live migration* of instances to ensure that hosts have enough spare capacity to accommodate the scale up events of their instances - if a host starts to become highly utilized ("hot"), instance(s) are moved away from it to create more room. Determining the criteria for initiating such migrations, which instance to migrate and where are highly complex decisions and Aurora Serverless makes novel contributions in this area. Notably, its load distribution strategy is characterized by a limited form of "unbalancing" of load across hosts - a departure from conventional load balancing-oriented techniques - to leave enough lightly loaded hosts that can accommodate migrations. Its instance packing strategy ensures that, in the common case, instance scale up can be realized completely locally within a host ("in-place scaling") without resorting to live migrations.
- *Scale up rate regulation:* There is a fundamental tension between allowing fast scale-up and operating hosts at high utilization levels. Aurora Serverless employs token bucket based regulation of instance growth to complement its packing and migration strategies. Such regulation helps prevent situations wherein a fast-growing instance causes its host's utilization to saturate leading to poor performance for itself and other co-located instances; the bounded growth rate (itself carefully tuned along with other system characteristics such as live migration times) allows Aurora Serverless to live migrate suitable instance(s) out of the affected host in time to avoid undesirably high utilization levels with high probability.[2] An important novel contribution is configuring target host utilization levels jointly with token bucket parameters based on extensive characterization of instance growth scenarios and migration times.
- *Distributed reactive control:* Aurora Serverless resource management spans two spatio-temporal scales: across a cluster ("fleet") and within a host. The fleet-wide control procures/releases hosts and determines instance-to-host mappings (including via migrations) while the intra-host control uses OS mechanisms for ensuring instances resource needs are adequately met. It uses a reactive style of control for its simplicity and ease of implementation. It also keeps the two control levels largely independent (i.e., loosely coupled). These choices allow its control to be scalable. Even though its intra-host scaling is reactive, the reaction starts well before the last bit of RAM is used up - this is achieved by maintaining a small band of capacity beyond the immediate needs to detect and accommodate short-term growth.
- *Systems mechanisms for efficiency:* Aurora Serverless implements a number of innovative systems mechanisms across the software stack in support of its operational goals: the Nitro hypervisor (security and isolation comparable to provisioned Aurora); an enhanced Linux kernel (frugality in instances' use of memory); and Aurora DB engines (relevant metrics that help Aurora Serverless with its host-level resource allocation and fleet-wide packing decisions.) In particular, it introduces a metric in the engine to estimate the size of the working set in the buffer cache. This metric allows an estimation of how much memory can be released back to the service without impacting customer experience.

[2] Despite careful tuning, occasionally live migration based remediation is outpaced by the growth in instance needs on a hot host. On such occasions, instances are prevented from scaling up for a duration of time required to created adequate spare capacity on the host.

Outline: The rest of this paper is organized as follows. In Section 2, we provide a background including the journey from ASv1 to ASv2. In Section 3, we provide an overview of ASv2 resource management with details of its inter- and intra-host levels in Sections 4 and 5, respectively. We present some empirical observations in Section 6. Finally, we discuss related work in Section 7, describe key lessons learnt in Section 8, and conclude in Section 9.

2 BACKGROUND

2.1 Challenges and a Key Design Principle

To offer customers the resource elasticity described above at high levels of efficiency, Aurora Serverless needed to address a number of challenges. These included policy issues such as: (i) how to define "heat" (i.e., resource usage features on which to base decision-making)? when to deem an instance "hot" (i.e., needing remedial actions)? when to deem a host hot? when to deem heat as having been remediated? (ii) on which host to place a new instance? how to carry out dynamic mapping of existing instances to hosts using live migration? and (iii) how to strike the right trade-off between utilization and scaling up rates? Questions of mechanism included: (i) what is the right virtualization solution? and (ii) what is needed within and outside the VM to seamlessly scale the DB engine?

Given the pioneering nature of Aurora Serverless, at the outset we had to make these choices without the benefit of large-scale field data. We wanted to avoid the risk of optimizing for a small exotic user-base which may have precluded other types of users from adopting the product. Therefore, a general design principle throughout Aurora Serverless's evolution has been to start off with minimal assumptions about our workloads and incorporate specificity only once we have seen enough evidence for it in our datasets. Some examples of this approach may be seen in the capacity bounds we allow customers to specify; the transition from ASv1 to ASv2; our initial choice of conservative utilization targets and scale-up rates; and our choice of reactive as opposed to predictive mechanisms. In some cases (e.g., capacity bounds), the initial choice has stood the test of time; in some others (e.g., ASv1 to ASv2, increase in our utilization levels) we have been able to incorporate lessons from our operation to refine our solution; and in yet others (e.g., exploiting predictability in workloads), we have ongoing work on incorporating such lessons into our approach.

2.2 The Aurora Serverless Capacity Bounds

Aurora Serverless offers its customers feature parity with the Aurora provisioned product while ensuring resource elasticity. The unit of resource capacity/ measure for Aurora Serverless is the *Aurora Capacity Unit (ACU)*. Each ACU is a combination of 2 GB of memory, corresponding CPU (currently 0.25 vCPU[3]), networking, and block device IO throughput. For a cluster using single-master replication,

[3] Subject to change based on new generation of hardware offering CPUs with improved characteristics.

the customer can create up to 15 read-only Aurora Replicas ("reader instances"). The customer defines a capacity range: the minimum and maximum capacity values (c_i^{max} and c_i^{max}, resp., for instance i) that each writer or reader can scale between. The capacity range is the same for each writer or reader in a DB cluster. The largest allowed value for $c_i max$ is 128 while the smallest allowed value for c_i^{min} is 0.5. The charges for Aurora Serverless capacity are measured in terms of ACU-hours accounted at 1-second granularity [3].

While crafting our capacity bounds, besides ease-of-use, we had to consider the following factors: (i) how close to a fully pay-as-you-go experience can we offer the customer? (ii) how efficiently and quickly can we resume a customer that returns after a period of inactivity? and (iii) at how high a utilization level can we operate our infrastructure? There is an inherent tension between these concerns. For example, if we let min ACU be 0 and actually remove a customer's resources after some idleness, we risk doing poorly on (ii). To do well on (ii), we would need to be able to predict well periods of idless so that paused DBs can be restored ahead of their next periods of activity. Setting the minimum capacity to a small number (as low as 0.5 ACU) lets lightly loaded DB clusters consume minimal compute resources. At the same time, they stay ready to accept connections immediately and scale up when they become busy. Aurora Serverless recommends setting the minimum to a value that allows each DB writer or reader to hold the working set of the application in the buffer pool. That way, the contents of the buffer pool aren't discarded during idle periods.

2.3 From ASv1 to ASv2

The journey from Aurora Serverless v1 to v2 exemplifies our approach of starting simpler and then adding more specificity based on operational experience. ASv1 was launched in August 2018. The most important difference was that, unlike using live migration as its resource management building block, it used a much simpler session transfer functionality. To scale up a database, the database would be relaunched. For this, ASv1 implemented support within database engines to allow interruption of service and session transfer from one backend to the other. ASv1 implemented a multi-tenant proxy frontend securely accessible from customers' VPCs via VPC Endpoints, which allowed the service to identify target database destination without relying on unique naming of the databases, user names or credentials.

Our team thought that a combination of these capabilities would allow database backends to scale in accordance with the workload and be completely released when idle. However, a number of limitations of ASv1's design soon became apparent. The scale-up approach required finding quiet points when session transfers would not disrupt customer performance; however, we found that this was not always possible for many of our customer workloads. Not all types of session state (e.g., temporary tables) were transferrable to a different backend. It became apparent that the burden of porting session transfer code into new versions of database engines was high since the service did not have full control of features added to both database products. ASv1's reliance on its session transfer protocol dictated other architectural decisions of ASv1 which would only be justifiable in a context of database instances rapidly swapped for instances of another capacity. However, having to swap instances of different sizes led to other customer experience issues: scaling was only possible in large increments (up or down by factor of 2). This also led to our scaling policy triggering scaling up or scaling down too late to be a cost-efficient solution. Recall these limitations of ASv1 that were illustrated in Figure 1(b). It was not possible to offer more precise workload tracking due to this scaling "see-saw problem" stemming from coarse-grained capacity increments. Despite these shortcomings, the popular adoption of ASv1 offered our team a number of useful operational and business insights. These allowed the service to reconsider the approach given a lot of customer traction and demand for a cost-effective and scalable serverless DB solution.

Based on our experience with ASv1, Aurora Serverless V2 was defined from the outset as a DB product which could scale in-place. This in itself would solve the problems created by coarse-grained scaling by factor of 2, making ASv2 more cost-effective than ASv1. Additionally, in-place scaling is faster in most cases than scaling across instances, which allowed for quicker response to increasing workloads. Such scaling can happen while SQL statements are running and transactions are open, without the need to wait for a quiet point. ASv2 can scale up and down faster. Scaling can change capacity by as little as 0.5 ACUs, instead of doubling or halving the number of ACUs. Scaling typically happens with no pause in processing at all. Scaling does not involve an event that the customer has to be aware of, as with ASv1. Scaling can happen while SQL statements are running and transactions are open, without the need to wait for a quiet point.

The ASv2 scaling would use two mechanisms: memory and CPU hot (un)plug and live migration of instances across hosts. Additionally, ASv2 would remove the need for a frontend proxy layer which compounded latency and noisy neighbor problems in ASv1. Finally, ASv2 would offer close to 100% feature parity with Aurora Provisioned and will be managed by the same systems which would simplify maintaining feature parity in future and prevent fracturing of the customer experience. This was made possible by introduction of a new VM type which offered a memory size scalability, an abstraction which our service has learned to deal with in years of production experience. The approach outlined above was made practical once operational, market and customer insights were accumulated. In the absence of data and customer traction with ASv1, justifying creating new building blocks necessary for ASv2 was believed to be impossible.

In the rest of the paper, we will focus on v2 and refer to it simply as Aurora Serverless without making the v2 explicit.

3 OVERVIEW

We will focus on resource management within a "fleet," a pool of hosts (with accompanying storage from Aurora storage service) that Aurora Serverless manages within an availability zone (AZ) [6].

3.1 Policies

Aurora Serverless resource management's overall decision-making is divided into two types of temporal/spatial granularity: (i) fleet-wide (details in Section 4) and (ii) within a host (details in Section 5). Figure 2 provides a high-level overview of this decision-making.

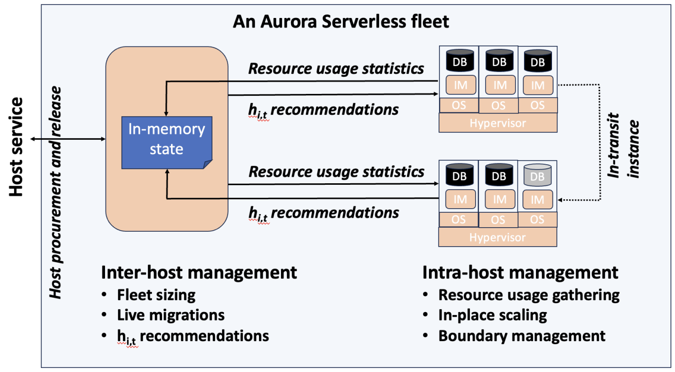


Figure 2: An overview of the Aurora Serverless data plane and control plane (including the resource management spanning fleet-wide and within-host decision-making). The control plane comprises a fleet manager and instance managers (IMs), one per instance. The fleet manager dynamically provisions adequate hosts for the fleet's needs. It obtains resource usage statistics obtained from the IMs and stores these along with other relevant state. It uses this information for its decision-making which comprises recommendations about live migrations and ACU limits (denoted $h_{i,t}$ for instance i at time t) that help alleviate heat from hosts that are found to be experiencing (or trending towards) saturation. The figure shows a live migration currently underway from one host to another. The IMs collaborate with the local DB processes and the operating systems to realize in-place scaling and boundary management.

3.1.1 *Within a host.* The resources within a host are managed collectively by the "instance managers," one per instance. The instance manager is responsible for resource usage gathering/inference, "in-place scaling" and "boundary management" for the instance. In-place scaling dynamically reckons the resource needs of the instance in the form of the instance's "reserved ACU"; let us denote this as $r_{i,t}$ for instance i at time t. An instance i is able to scale up instantaneously in the small band (if any) between its current usage $u_{i,t}$ and its reserved ACU $r_{i,t}$. See illustrations of such instantaneous scaling bands for instance i in Figure 3. Further scale up beyond the reserved ACU requires "boundary adjustment," i.e., increasing the instance's reserved ACU. Boundary management ensures that the instance's reserved ACU is not wastefully overprovisioned relative to its needs and that it does not grow at a pace that may lead to undesirable resource contention with its co-located instances. The instance manager polls an engine-specific agent for resource usage information and uses this to determine how to scale the DB. It also interacts with the guest OS to enforce resource limits based on its observations of recent usage. In Figure 3, the instance's usage is $u_{i,t1}$ and will be allowed to grow up to $r_{i,t1}$ instantly (in-place scaling). If the instance's needs grow further, its reserved ACU will also be increased but in a rate regulated manner. Keeping reserved ACU slightly above the usage allows us to quickly detect a growing trend in the instance's usage.

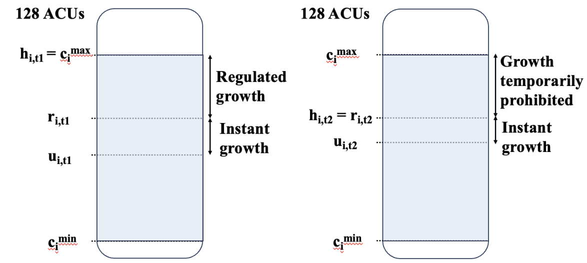


Figure 3: An illustration of various instance-specific ACU limits relevant to resource management. On the left, at time $t1$, we depict an instance on a host that is not in danger of becoming hot. The fleet manager sets $h_{i,t1} = c_i^{max}$. The scaling band between $r_{i,t1}$ and $u_{i,t1}$ allows quick detection of and reaction to short-term growth in resource needs. On the right, at time $t2$, the host has been deemed hot, and the fleet manager temporarily limits the maximum capacity the instance may grow to $h_{i,t2} = r_{i,t2} < c_i^{max}$.

3.1.2 *Fleet-wide.* A service called the fleet manager modulates the fleet size at a coarse timescale (currently weeks/months) based on desired utilization levels and predicted demand. The fleet manager employs live migration of instances to ensure that a host doesn't operate in a regime where its capacity cannot accommodate the needs of the instances placed on it: if a subset of instances A on a host starts exhibiting a growing trend in its collective resource usage that may lead to undesirable levels of resource contention on that host, the fleet manager must be able to migrate a suitable subset of instances B (not necessarily same as or even intersecting with A) away from the host in a timely manner. There are two aspects of this problem:

(1) How to ensure that there is enough spare capacity on the host to continue to serve any growing instance needs while the live migration based remedial actions are being carried out? Similarly, the host an instance is migrated to must be able to accommodate its needs in addition to the instances it already houses.
(2) How to ensure that growth in resource needs of instances does not outpace live migration based remedial actions?

Aurora Serverless's fleet manager addresses (1) by controlling the mapping of instances to hosts which in turn determines the distribution of spare capacity on the hosts. (2) is addressed by a combination of fleet-wide heat management and host-level mechanisms. For hosts undergoing live migrations to remediate resource pressure ("heat"), the fleet manager may impose short-term limits on the maximum ACUs for its instances. Denoting this fleet manager prescribed max ACU limit for instance i at time t as $h_{i,t}$ during a period of heat remediation, we may have $h_{i,t} < c_i^{max}$. These limits are lifted as soon as heat has been remediated, so in the common case we have $h_{i,t} = c_i^{max}$. In Figure 3, we illustrate these scenarios for instance i. At time $t1$ when the host is not deemed hot, $h_{i,t1} = c_i^{max}$. At time $t2$ the fleet manager has deemed the host hot causing it to temporarily freeze $h_{i,t2}$ at $r_{i,t2}$.

3.2 Mechanisms

Aurora Serverless continues to use Aurora's disaggregated storage architecture [37] and, therefore, inherits the latter's reliability and durability characteristics. Aurora data is stored in the cluster volume, which is a single, virtual volume that uses solid state drives (SSDs). The storage for each Aurora DB cluster consists of six copies of all customer data, spread across three AZs. This built-in data replication applies regardless of whether the DB cluster includes any readers in addition to the writer. When data is written to the primary DB instance, Aurora synchronously replicates the data across AZes to the six storage nodes associated with the cluster volume. Aurora cluster volumes automatically grow as the amount of data increases. The maximum size for an Aurora cluster volume is 128 or 64 TB, depending on the DB engine version. For its compute layer, Aurora Serverless procures hosts comprising 256 ACUs.

Aurora Serverless leverages innovations in systems software that we describe in Section 5.1. Briefly, it uses a new instance type based on the Nitro system [7] that provides Nitro's low IO latency (based on hardware IO virtualization, exposed to the instance through SRIOV), along with flexible CPU and memory provisioning. It relies upon enhancements to the Linux operating system to allow the RAM capacity of an instance to be adjusted dynamically. Aurora Serverless relies upon mechanisms within our DB engines to provide estimates of working sets and take measures such as trading memory from buffer cache for user-queries or selectively shed load during transient high resource pressure scenarios. Finally, Aurora Serverless leverages a live migration facility that allows a running instance to be transparently moved from one host to another with minimal disruption.

4 FLEET-WIDE RESOURCE MANAGEMENT

The fleet-wide resource management is responsible for decision-making over the timescale of minutes to hours and larger. It relies upon three key control knobs: (i) live migration; (ii) modulation of a per-instance fleet manager prescribed ACU limit ($h_{i,t}$) that has an impact on the boundary management within an instance; and (iii) fleet size adjustment. A key concern is how/when we deem a host as requiring remedial actions to relieve resource pressure on it; we refer to such a host as having become "hot." The approach we settled on is based on defining critical levels of utilization along one or more of the following capacity dimensions: (i) CPU bandwidth; (ii) allocated RAM; (iii) total network throughput; or (iv) local block device I/O throughput (used by indexing process, sorting results, etc.) Besides being simple, this approach also has had the advantage that, as our mechanisms improve in their efficiency, we have been able to raise these critical utilization thresholds.

4.1 Live Migration-Based Dynamic Instance Re-Packing

The fleet manager periodically polls individual instance managers to retrieve (highly granular, second-level) resource usage metrics that it uses for its decision-making. Once every second, it carries out the following three-step procedure to determine if any live migrations are needed for re-packing instances.

Step 1: Which hosts need out-migrations? For every host, the fleet manager runs a host heat aggregation task to assess if this host needs some of its instances moved elsewhere. This assessment is based on observing if the summation of the reserved ACUs for the host's instances over the last few minutes exceeds a threshold θ_{acu}^{mig}.[4] Hosts where this threshold has been breached are considered as having become hot. The fleet manager then checks if a hot host has enough network bandwidth to sustain an out-migration. For hot hosts that can sustain out-migrations, the fleet manager proceeds to determining which instances to out-migrate in Step 2 below. Also, for all hot hosts, if their usage crosses a threshold $\theta_{acu}^{hi} > \theta_{acu}^{mig}$, the fleet manager prescribed max ACUs of all their instances (recall $h_{i,t}$ for instance i at time t from Figure 3, Section 3) are held at their current reserved ACU values; this has the effect of freezing the resource allocations of the instances to their current values, i.e., disabling further growth beyond their current allocations. The fleet manager prescribed max ACUs for instances on these hosts are reset to their respective customer max ACUs once their heat has been dissipated via migrations and/or a drop in the workload intensity of some instances.

Step 2: Which instance to migrate out of a hot host? Identifying the right instance to migrate is a non-trivial decision because there are multiple criteria that have a bearing on the quality of the decision. E.g., consider that: (i) for the same memory size, an instance with higher CPU or network usage may be a better candidate to be out-migrated; (ii) migration times are affected both by the current memory image size and the dirtying rate during the migration making certain instances faster to migrate than others; and (iii) it may be desirable to have a notion of "fairness" in how many times an instance gets migrated over a period of time. Theoretically, this problem is a form of online bin-packing with the added complexity of migrations and is, therefore, NP-hard [17]. The key design challenge for Aurora Serverless was to come up with a heuristic that was effective yet extremely fast and scalable (this decision-making needs to occur within a few hundreds of milliseconds for pools of hundreds of hosts each potentially containing tens-hundreds of instances) and could keep DB performance stable despite migrations. The fleet manager employs a 3-stage heuristic that we have empirically found to offer a good trade-off between these requirements.

– Stage 2a: Certain filters for migration eligibility are applied to reduce the number of instances to consider for out-migration (e.g., an instance that was recently migrated is dropped).

– Stage 2b: A preference ranking of the filtered instances is created. Each preference score is binary and these are combined via a weighted sum. As one example, one such score captures if an instance was migrated recently. Another prefers instances that have been heard from recently (and, so, are less likely to be unavailable).

– Stage 2c: Two rankers are used to create numeric scores, one per instance: (i) the first ranker returns a score proportional to the reserved ACU for the instance with the idea being to choose an instance whose migration will relieve a larger amount of heat (and in turn help keep the number of migrations low); and (ii) the second ranker returns a linear combination of (roughly) the unused fractions of the network Rx/Tx throughput, EBS throughout, and EBS IOPS capacities available to the instance. For both these

[4]Specifically, high percentiles of the aggregate reserved ACU empirical distribution are compared against the thresholds.

rankers, an instance that is more desirable gets assigned a higher score. These two scores are combined as a weighted-sum into a single score.

The instance with the highest preference score is selected with ties broken using the numeric score.

Step 3: Where to migrate? The fleet manager follows a 3-stage process similar to that in Step 2.

– Stage 3a: Apply filters to ensure that the host is active, would not exceed the heat threshold with the new instance, has bandwidth to support the migration, and the number of instances on the host is less than a threshold.

– Stage 3b: Create a preference ranking of hosts based on: prefer to place instances of the same cluster on separate hosts for fault tolerance reasons; prefer hosts that don't have recent failures or were involved in failed migrations.

– Stage 3c: Compute two numeric scores and then combine them via a product (host with higher score is deemed a more desirable destination). The first amounts to a "best-fit" like heuristic for bin-packing and computes the score as the ratio of the heat on the host with the new instance over θ_{acu}^{mig}. Roughly, the idea is to ensure that the load is *unevely* distributed among the hosts such that some hosts have enough headroom for serving as live migration destinations. The second computes the score as 1− ratio of ACUs corresponding to the most utilized resource over θ_{acu}^{mig}; this score has the effect of reducing the load imbalance across resources (i.e., CPU vs. memory vs. network). In our evaluation in Section 6.3, we provide empirical justification for our heuristic by comparing it with alternatives.

An illustrative example: In Figure 4, we illustrate the fleet manager's live migration decision-making by showing relevant aspects of the state of 111 hosts within a fleet. Here, host *s* has been deemed hot based on the criteria described in Step 1 above. In this specific case, this occurred due to instance *i*'s scale-up which caused the host's aggregate ACUs to exceed the θ_{acu}^{mig} threshold. Following this, the fleet manager chose instance *i* itself for out-migration and chose host *d* as its destination. It is worth highlighting a couple of observations. First, notice how the fleet is fairly well-balanced in terms of the aggregate ACU load on the hosts, yet some hosts (e.g., host *d*) have been left relatively unoccupied to facilitate large-sized migrations should there be a need. Second, notice that the destination chosen for instance *i* is not the host with the most spare capacity (e.g., compare host *d* with host d'). Both of these result from our strategy of keeping the fleet slightly unbalanced to facilitate faster/fewer live migrations than would occur in a more balanced system.

4.2 New instance placement

The difficulty here is the lack of knowledge about the resource needs of the new instance (in the general case). The only hints available to the system are the customer min/max ACU limits. Aurora Serverless uses customer min ACU as the new instance's resource demand. The reason is related to the much faster scale up that Aurora Serverless supports than its scale down: in case the customer min ACU is an underestimate, Aurora Serverless's in-place scaling can adapt quickly while if the customer max ACU proves an overestimate it would lead to wasted capacity for a relatively long scale down period. Following this, the remaining problem of determining the host that can best accommodate this new instance is precisely the same as that addressed by Step 3 above.

The fact that the customer does not pay more for higher max ACU means that we encourage customers to choose peace of mind when it comes to maximum capacity and promise to use it only if we need to. Under these conditions it is not reasonable to place a new instance based off a worst-case scenario where the instance will need to serve workload at max ACU right away.

4.3 Fleet size adjustment

Given the current growing phase of Aurora Serverless the focus of fleet sizing is on ensuring that a fleet always has adequate hosts and additional hosts are requested with enough lead time. Generally, as we have learned more about our customers' workloads and as the efficiency of our resource management techniques has improved, we have been able to increase our desired utilization levels. The fleet manager employs a combination of fleet-level demand prediction and triggering additional procurement upon a fleet utilization exceeding a predetermined threshold. Employing more sophisticated prediction and integrating fleet sizing more tightly with the rest of the resource management is an area of future work.

A key consideration in fleet sizing are the computation and high-frequency data gathering that the resource management system has to do as these overheads grows with fleet size. We maintain the fleet size below a number which allows for a locally computed fleet health using one heat management server.

5 RESOURCE MANAGEMENT WITHIN A HOST

Resource management within a host is carried out by instance managers, one per instance on the host. The instance manager is a library that encapsulates Aurora Serverless's functionality within an instance. Serverless capabilities are a plug-in the Aurora instance manager whereas the rest of the functionality unrelated to resource management is the same as regular Aurora, thereby enabling feature parity. Figure 5 summarizes the instance manager and its interactions with various entities. In the following, we first describe various enabling mechanisms for the Aurora Serverless instance manager followed by its resource management policies.

5.1 Mechanisms

5.1.1 Data collection. The instance manager relies upon an engine-specific agent for functionality idiosyncratic to the engine which includes: (i) how the engine scales based on a provided amount of buffer pool; and (ii) collecting the engine's usage and estimate of its desired buffer pool size. The engine employs its own internal algorithms for estimating these quantities and writes them into a shared memory segment from which the agent reads them.

For usage statistics of all resources other than the buffer pool, the instance manager relies upon the guest OS. It employs a number of "metrics fetchers" that it runs once every second to gather resource usage information from the OS. These metrics are collected into an in-memory scaling data report. The choice to look at per-second data was made based on ensuring responsiveness to resource usage spikes at a fine timescale. In particular, while deficit of CPU cycles usually causes gradual degradation of experience, the lack

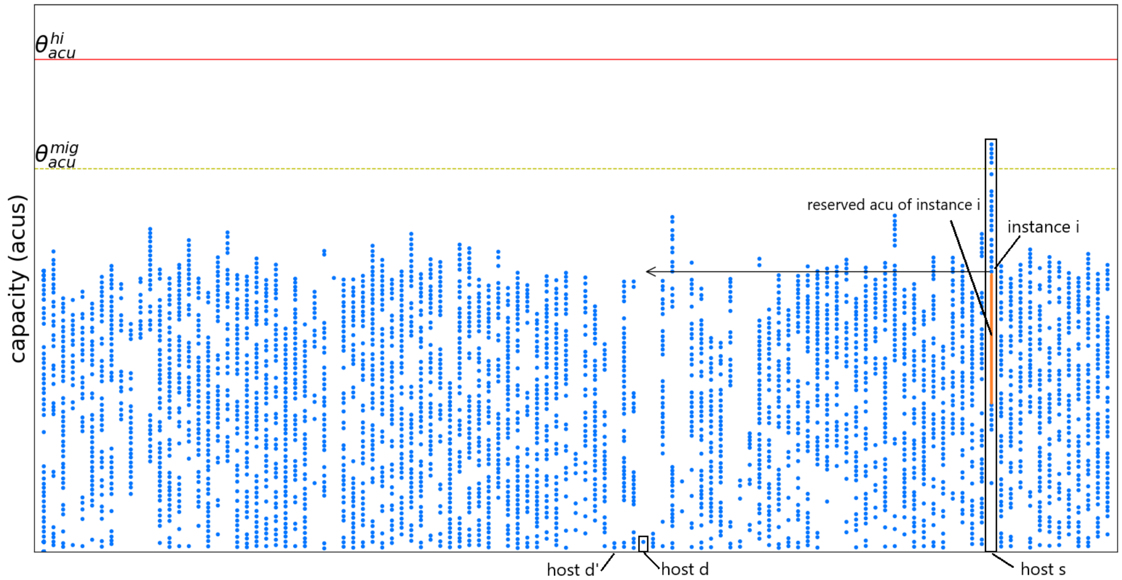


Figure 4: An illustration of the fleet manager's live migration related decision-making. We show the state of 111 hosts within a fleet at a point in time when host s has been deemed hot because the summation of reserved ACUs of its instances was found to exceed the threshold θ_{acu}^{mig}. The y-axis represents ACUs. For each host, the reserved ACUs of its instances are depicted as the vertical gaps between successive blue dots. Instance i is chosen for migration and host d is chosen as its destination.

of memory will eventually cause a crash. Therefore, we need to approve memory increase as soon as we know more memory is necessary if we can at all.

5.1.2 Virtualization solution. DB engines, like the MySQL and PostgreSQL engines supported by Aurora, are complex, written in languages that don't provide memory safety, handle arbitrary code and data, and even run native customer-provided code in the form of extensions. The complexity makes DB engines vulnerable to classic security threats like buffer and stack overflows, mostly caused by memory safety bugs. DB engines are not suitable as a security boundary in a multi-tenant environment like AWS, and need to be wrapped in a stronger isolation primitive. Since launch, Aurora has run each database instance in its own virtual machine (VM) secured by hardware virtualization. This provides strong protection between DBs, and customers, against remote code execution, side channels, and other security vulnerabilities. Aurora Serverless needed to retain this security posture, but also needed a solution that allowed it to dynamically scale the amount of memory, CPU, and other resources available to each DB engine. It initially evaluated the Firecracker VMM [10], which provides secure, lightweight, and flexible virtualization. Firecracker was a great fit in several ways: it meets the systems' security needs by using hardware virtualization for isolation, allows dynamic scaling of memory and CPU, and has per-VM overhead as little as 5MB. Unfortunately, it was found that the latency and CPU overhead of userspace IO virtualization (as used in Firecracker), combined with Aurora's disaggregated storage design [37], lead to unacceptable loss of performance for workloads with poor cache locality (which are particularly sensitive to increased latency to Aurora's storage layer). Our team developed a new instance type based on the Nitro system [7] that provided Nitro's low IO latency (based on hardware IO virtualization, exposed to the instance through SRIOV), along with flexible CPU and memory provisioning.

5.1.3 Support for resource overbooking. The hosts have the ability to over-subscribe (i.e., overbook) CPU and memory. Concretely, CPU over-subscription means that the sum of the vCPUs across all the instances within a host can exceed the total number of physical CPUs on the host; memory over-subscription means that the total amount of memory corresponding the customer max ACUs of the instances can exceed the host's physical memory.

5.1.4 Support for efficient memory scale-up. Any Aurora Serverless instance, regardless of its actual resource needs, is of the same instance type called `db.serverless` with the ability to grow up to a size of 128 ACUs. In this sense, Aurora Serverless instances are over-provisioned. Because of this over-provisioning, there is a particular memory usage efficiency problem that arises in the Aurora Serverless guest kernel. By default, the Linux kernel views any unused memory as "wasted" and is designed to use as much memory as it can. This is undesirable in the Aurora Serverless context because the kernel will tend to keep growing its used memory potentially all the way up to the Customer Max ACU. Aurora Serverless implements a few features within its guest kernel to make it frugal in its memory usage.

– *Memory offlining:* The kernel implements memory offlining - the ability to dynamically release portions of its memory to the host. Memory offlining also helps reduce some meta-data memory overhead - Linux maintains a 64B `page struct` for every 4KB page which amounts to 2GB for a `db.serverless` instance.

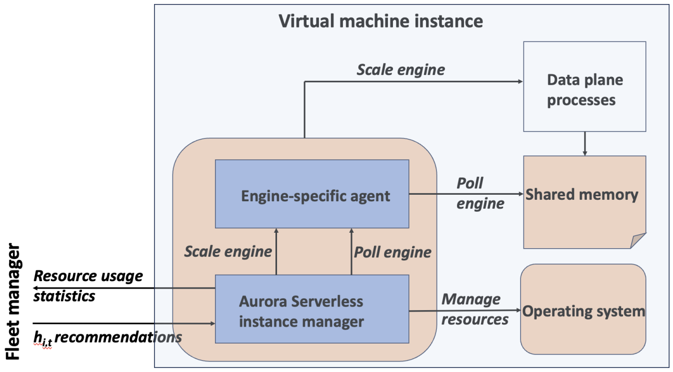


Figure 5: The Aurora Serverless instance manager and its key interactions with entities within and outside the instance.

– *Cold page identification:* A kernel process called DARC [1] continuously monitors pages and identifies cold pages. It marks cold file-based pages as free and swaps out cold anonymous pages.
– *Free page reporting:* Aurora Serverless introduces an explicit free page reporting mechanism. A daemon runs within the guest looking for free pages. If it finds a 2MB free block it reports it to the hypervisor which can then reclaim it.
– *Compaction:* Nitro operates only at 2MB granularity to keep page fault overheads resulting from its reclamation of free pages low. So, Aurora Serverless guest OS employs a compaction activity to coalesce 4KB free pages into 2MB blocks whenever possible to increase the likelihood of them being reclaimed by the hypervisor.

5.1.5 Boundary enforcement. This involves ensuring that the instance is allocated resources based on the "boundary" determined by the scaling policies in Section 5.2. The instance manager employs two types of mechanisms to control its instance's CPU and memory allocations (similar ideas apply to other resources):
– *cgroup:* The more fine-grained mechanism uses `cgroups` [8]. Broadly, the engine and instance manager processes are made part of a group and resource allocations corresponding to local max ACU are specified for this group. For CPU, this mechanism is especially useful for controlling allocation at the sub-ACU granularity. Resource isolation is offered to instance manager from the data plane processes using CPU shares and a guaranteed size for the instance manager JVM.
– *CPU and memory on/offlining:* Complementing the above, vCPUs or memory (at the granularity of 2MB) may be added to or removed from the guest kernel to control resource allocations for the instance. Bringing on additional vCPUs can help mitigate the "noisy neighbor" problem and is especially useful for ensuring good performance when multiple small ACU instances are consolidated together. Memory offlining offers a way to limit the memory usage of processes that `cgroups` is unable to (since the latter can only limit user-space processes).

5.2 Policies

There are two key important policy issues for the instance manager: (i) boundary management and (ii) in-place scaling.

5.2.1 Boundary management. This aspect of the instance manager's decision-making is concerned with dynamically adjusting the reserved ACU (i.e., the resource allocation boundary) of the instance based on its recent usage patterns. There were two main considerations that went into the Aurora Serverless boundary management strategy.
Agile and efficient detection of growth trends: The reserved ACU should be chosen so it is likely to remain slightly higher than the resource consumption in the near future. Recall from Figure reffig:acus how this gap (i.e., the scaling band) is crucial for quickly detecting growing resource needs; such detection allows the instance manager to correspondingly reassess and increase the instance's allocation. At the same time, the scaling band should not result in wasteful over-provisioning of resources.

The approach we found to work well was as follows. The instance manager monitors the data plane memory footprint every second and maintains a history of the instance's estimated dynamic memory usage. It uses the maximum memory usage over a the last minute and converts it into ACUs to determine the instance's reserved ACU It goes beyond just memory. We also look at other parameters like CPU usage and network throughput and block device IO. When any of these parameters exceed the current allowed maximum for a given amount of ACU, the service deems this VM as using more ACU even if other parameters are not exceeding the quota. This leads to more resources allocated to the instance if current ACU usage is below the customer-configured maximum.
Regulated growth: The reserved ACU should be allowed to grow only so fast that does not create the risk of a fast-growing instance overwhelming the fleet manager's live migration based remedial actions. For this, the instance manager limits the scale up rate to give the fleet-wide resource management enough time to remediate high heat on a host via live migrations. Specifically, the instance manager implements a token bucket based regulator for the instance's reserved ACU (recall $r_{i,t}$ from Section 3); therefore, as long as an instance's usage is below its reserved ACU it enjoys nearly instantaneous scale up. The classical token bucket, originating in networking [15], is characterized by two parameters: (i) a sustainable (or committed) rate (R bytes/s) and (ii) a bucket size (B bytes). Conceptually, tokens arrive at a fixed rate R into the bucket. The policing is based on ensuring that a resource allocation request is only satisfied when tokens equal to its size are available; the contents of the bucket are adjusted to reflect this discharge of tokens.

The instance manager adapts the classical token bucket for the ACU abstraction with the following enhancement: the token bucket parameters are defined to be *increasing* functions of the current local max ACU c, i.e., $R = R(c)$ ACUs/s and $B = B(c)$ ACUs. Aurora Serverless has made this choice based on its own observations as well as customer demands that their workloads tend to require higher rates of growth at higher intensities. The instance manager token bucket parameters are informed by empirically observed distributions of migration duration and workload resource growth. At a high level, with a high probability, the heat that a single well-chosen migration is able to dissipate should be able to comfortably accommodate the growth allowed by the token bucket during the migration.

Finally, Aurora Serverless also limits the scale down rate. Limiting the scale down rate is an inevitable consequence of the limited scale up rate - since scale up only occurs at a deliberately controlled rate, instance manager chooses to not scale down aggressively lest high workload intensity return after a brief lull. The key idea is to allow enough time of low activity elapse before commencing a scale down.

5.2.2 In-place scaling. The core element of in-place scaling is concerned with determining the ideal scaling target (i.e., resource allocation level) for the engine. The instance manager takes a more cautious approach towards scaling down than for scaling up. The following two-step process used to determine the ideal scaling target for the engine. This decision-making runs once every second and considers up to a few minutes worth of per-second historical usage data. The bulk of the decision-making is concerned with resource-specific "deciders" each for estimating the instance's needs for a particular resource type.

Step 1: Deciders: Each decider converts its projection of its specific resource's needs into the common currency of the engine's buffer pool needs allowing for comparison across various deciders. We describe below the decider based directly on engine buffer pool followed by a representative non-buffer pool decider:

– Estimate the engine's buffer pool needs. For this, it takes the maximum buffer pool usage by the engine over the last minute as its projected buffer pool need. The specifics of how this is decided vary across engines and, therefore, it is obtained by the engine-specific agent from the memory segment it shares with the engine.

– Respond to a substantial change in the recent CPU usage of the instance. First identify the following two quantities: (i) P50 CPU usage over the last 30 seconds; and (ii) P70 CPU usage over the last 60 seconds. For each of these percentiles, it determines the corresponding memory sizes (call them mem_{30s} and mem_{60s}, resp.) based on the shape of an ACU. If mem_{30s} exceeds the current engine fixed memory the decider's target engine fixed memory size is set to mem_{30s}. If, however, both mem_{30s} and mem_{60s} are lower than the current engine fixed memory, the decider's target engine fixed memory size is set to mem_{60s}. Note how a potential scale down is considered more cautiously than a scale up.

Four additional resource-based deciders are used for network (received and transmitted) and storage (bandwidth and IOPS) that are conceptually similar to the one for CPU described above.

Step 2: Combining various deciders: The engine buffer pool needs emerging from the various deciders are converted into a single projection by taking their *maximum.* This ensures that a scale up occurs if *any* of the deciders prescribes a capacity increase whereas a scale down occurs only if *all* of them do so. The minimum scaling granularity is 0.5 ACUs.

6 EMPIRICAL OBSERVATIONS AND EVALUATION

In this section, we use a combination of observations from Aurora Serverless fleets and simulations to understand the efficacy of its resource management.

6.1 Datasets and metrics of interest

We report metrics from 2 different Postgresql fleets: (i) *Fleet 1:* AWS region us-east-1 over the 17-day period 01/14/2024 - 02/01/2024; (ii) *Fleet 2:* AWS region us-west-2 over the 31-day period 01/01/2024-02/01/2024. We also use instance-level reserved ACU measurements from these fleets for our simulations.

Generally, two types of metrics are of interest in evaluating the efficacy of Aurora Serverless resource management. The first type of metrics relate to a fleet's operational efficiency, e.g., host utilization. The second type of metrics relate to customer experience, e.g., the resource elasticity offered. While we are unable to share absolute numbers related to our fleet sizes, host utilization levels, and settings for operational parameters such as utilization thresholds or token bucket due to their proprietary nature, we will present the following metrics to help understand customer experience:

- What percentage of scale up events were satisfied in-place vs. via live migration? A smaller percentage is indicative of the efficacy of our re-packing and placement strategies.
- What percentage of scale up events resulted in a host being deemed hot? Recall that, for a host deemed hot, its instances max ACU is temporarily limited to their current reserved ACUs while remedial live migrations are being undertaken. What is the impact on customer workloads of these remedial actions?

6.2 Customer experience observations

In Fleet 1, there were a total of 33,792 instances during our observation period. Collectively, these instances exhibited 16,440,024 scale-up events. Of these, only 2,923 scale-up events needed one or more live migrations (i.e., the θ_{acu}^{mig} threshold on a host was breached) while the vast majority (99.98%) were satisfied completely via our in-place scaling mechanism. Of the scale-up events that couldn't be satisfied in-place, a majority (52%) needed only 1 live migration; the average number of live migrations for such scale-up events was 1.68. Lastly, the number of occasions when a host breached its θ_{acu}^{hi} threshold was 198, i.e., for 6.77% of the scale-up events that couldn't be satisfied in-place, instance reserved ACUs were temporarily held at their current allocations.

In Fleet 2, there were a total of 12,467 instances during our observation period. Collectively, these instances incurred 8,151,229 scale-up events. Of these, only 1,214 scale-up events needed one or more live migrations while the rest were satisfied completely via our in-place scaling mechanism. Of the scale-up events that couldn't be satisfied in-place, a majority (55%) needed only 1 live migration; the average number of live migrations for such scale-up events was 1.56. Lastly, the number of occasions when a host breached its θ_{acu}^{hi} threshold was a mere 48, i.e., for 3.95% of the scale-up events that couldn't be satisfied in-place, instance reserved ACUs were temporarily held at their current allocations.

These observations indicate that the Aurora Serverless repacking and placement strategies are effective at ensuring that a vast majority of scale-up requests are satisfied completely locally within the current host giving the customer a seamless resource elasticity experience. Further, even when re-packing is needed, generally a single migration suffices to remediate which limit any adverse restrictions on customer resource elasticity and performance.

6.3 Comparison against an alternative re-packing strategy

To illustrate how we chose specific aspects of our overall approach vs. other natural alternatives, we compare against a baseline in simulation that modifies Step 3 ("Where to migrate?") of our technique in Section 4.1 as follows: it picks a purely best-fit approach instead of our approach that combines best-fit with CPU/memory balancing. Generally, we find that our the baseline tends to concentrate instances onto a smaller set of hosts than our technique causing these host to have higher utilization than occupied hosts with our technique. Our technique maintains a few more empty or lightly loaded hosts than the baseline which make it need fewer migrations, on average, to alleviate heat. For Fleet 1, we find that the total number of live migrations with baseline is 82% higher than with our technique while the average utilization on a busy host is about 10% higher. As an indicator of the impact of customer experience, the total time instance ACU allocations were "frozen" while heat was being remediated went up by about 55%. Similarly, for Fleet 2, we find that the total number of live migrations with baseline is about 57% higher with the average utilization of hosts being about 12% higher. The total time instance ACU allocations were frozen went up by 57%.

6.4 A close look at a migration-assisted scale up

Finally, we take a close look at an instance whose scaling up was satisfied in-place due to capacity created by carrying out a live migration of a different instance. Figure 6 shows the key events of interest. We focus on a time-period (normalized to a (0, 100) time units range) during which the host (labeled "source host") housing our scaling instance ("scaling instance") became hot due to a rapid growth in the resource needs of the scaling instance. We depict how the aggregate reserved ACUs (normalized in a (0, 100) range) for source host evolved over this time period. We also show the heat contributed by scaling instance which starts to scale up around 35 time units. At around 41 time units, the θ_{acu}^{mig} threshold was found to have been breached by the fleet manager following which it identified an instance "migrated instance" to out-migrate. The chosen destination was "destination host" which was much less loaded than source host. We also show a timeline of the heat on destination host. The migration commenced at 41 time units and concluded at 50 time units. We see that the migration was effective in allowing the scaling instance to meet its needs. The aggregate heat on source host stayed below θ_{acu}^{mig} after the migration concluded mainly because the scaling instance did not scale up any further (it began to scale down steadily soon after peaking and is back to its nearly 0 usage at about 66 time units). If it had needed to scale further, the fleet manager would have initiated additional migrations to allow such growth.

7 RELATED WORK

There is extensive literature on resource management in clusters and datacenter-scale systems going back to the 1990s. Aurora Serverless adapts several ideas from this body of work and fine tunes them for its needs. Multiples forms of complementarity in resource needs of workloads have been exploited. Colocating CPU- vs. memory-

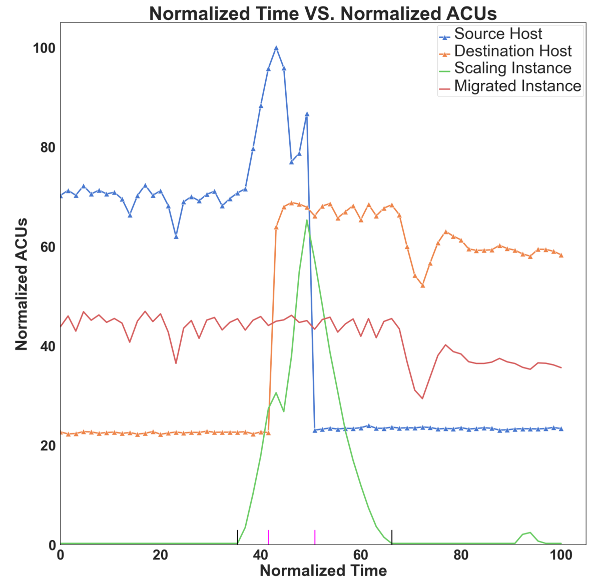


Figure 6: A close look at how in-place scaling was achieved for an instance whose resource needs grew from close to 0 ACUs to a significantly larger value (we are reporting ACUs on the y-axis normalized in a (0, 100) range since we are unable to reveal the actual θ_{acu}^{mig} used in our system.) We show the reserved ACUs for the instance that scaled up and a different instance that was migrated to allow the first instance to continue scaling up when its needs could not be accommodated locally. We also show the sum of reserved ACUs on the source and destination hosts. The pink vertical lines show the span of the live migration (again, we are only able to report times normalized in the (0, 100) range since information about our migration times is proprietary.) The black vertical lines depict the period over which the scaling instance scaled up and then scaled down.

vs. network-intensive workloads can help improve packing density [20, 24, 34, 39]. An individual workload's peak needs may occur rarely allowing for *under-provisioning* resources for it [27, 36]. The aggregate needs of a group of workloads may exhibit a much lower peak demand than the sum of individual peaks (because of individual peaks occurring at different times) allowing for resources to be *overbooked* [13, 16, 32, 36]. While Aurora Serverless's approach is heavily influenced by these ideas (especially those relating to overbooking), a distinguishing feature is that it bases its decision-making on recently observed workload features rather than long-term profiling and characterization. This approach is motivated, in part, by its simplicity and by the availability of a highly agile corrective live migration facility.

Numerous works have demonstrated the merits of combining predictive and reactive techniques for management [25, 29, 30, 35]. Earlier work in this space used classical ideas from queueing theory and control [14, 19, 33]. Lately there has been much interest in studying techniques using (deep) ML/RL for such problems [21, 22, 25]. Whereas Aurora Serverless's reactive mechanisms have shown themselves to be effective, incorporating predictive management for fleet-sizing and scheduling of migrations may yield further improvements that we are currently investigating. For example, many Aurora Serverless workloads exhibit periodicity such as time-of-day effects [11, 12] that lend themselves to predictive management. Our preliminary investigation suggests that a particular weakness of reactive mechanisms relates to releasing unused resources - recall Aurora Serverless's conservative scale-down. Predicting a downward trends in resource needs may allow Aurora Serverless to release idle hosts more effectively for cost savings.

While live migration as a knob for resource management has been studied in numerous papers [18, 28, 38, 40], its use in a commercial scale system is relatively uncommon due to the associated complexity. Aurora Serverless makes important contributions in establishing live migration as an effective control knob for a performance-sensitive and challenging class of workloads.

As a load distribution problem, our approach stands in contrast to most clustered systems - whereas most systems attempt to keep their system *load balanced* [23], Aurora Serverless falls into the less common category of systems that deliberately keep their cluster "unbalanced" [26, 31]. Specifically, it ensures that the fleet always has some hosts with enough spare capacity to serve as destinations for an instance migrated from one of the busier hosts.

8 LESSONS LEARNT AND KEY TAKEAWAYS

We highlight some salient lessons learnt during our journey developing and improving Aurora Serverless's resource management.

• As stated at the outset, a central design principle has been to start simpler and avoid the trap of premature optimization. This was especially important in the field of resource management where a vast and rich set of techniques exist. We provided several examples of this approach where more complexity was only added based on observing customer workload features and needs.

• Designing for a predictable resource elasticity experience has been a second central tenet and we have co-designed our SLAs and resource management accordingly. Specifically, any improvements in efficiency must align with the goal of providing a consistent performance experience. A particularly interesting design choice in this context is to bound the growth rate allowed for an instance's resource allocation even if this means that, on occasion, we do not let an instance grow as fast as the available headroom on its host would theoretically allow.

• The decision-making in Aurora Serverless's resource management is predominantly reactive in nature. By this we mean that no explicit predictions of resource needs (or any other workload features) are employed. Resource management components employ recent data (granularity of a minute to a few minutes) as representatives of upcoming behavior (i.e., assume minute-scale temporal locality in workload features), which may be viewed as a form of implicit prediction. Clearly, exploiting predictability in workloads (e.g., time-of-day effects [11, 12]) can help Aurora Serverless improve its costs (e.g., by carrying out migrations well ahead of time and releasing hosts during low workload intensity) and is indeed an important area of ongoing research. The key merit of reactive control over alternatives based on prediction is its simplicity. Furthermore, as migration technology becomes more agile, the gap between reactive and predictive management is likely to shrink.

• We found it effective to have the fleet-wide vs. host-level aspects of resource management operate largely independently of each other. The only interaction between them happens through an instance-specific $h_{i,t}$ that the fleet manager lowers from its default value of customer max ACU on a hot host. This in turn affects the reserved ACU limit used by boundary management. This significantly simplifies our resource management algorithms and allows them to be more scalable than the alternative.

• Finally, Aurora Serverless evolution offers a powerful illustration of being able to evolve hypervisors and OS kernels in ways that make them better suited for DB workloads. This seems to be an under-tapped area of research, and there may be a lot of opportunity in co-designing and co-optimizing these layers.

9 CONCLUDING REMARKS

We described how resource management in the latest version of Aurora Serverless adopts classical techniques while also contributing some novel ideas. It treats predictable resource elasticity as its central goal. This was best exemplified by its use of token bucket regulation for instance scale up even in regimes where faster growth is theoretically allowed by the host's spare capacity. It relies on a largely reactive control based on a loose coupling between cell-wide and host-level management informed by recent minute-level measurements (i.e., assumption of short-term temporal locality in workload features). Its central control knob is a live migration facility. It contributes novel heuristics for identifying instances to migrate and their destinations that carefully balance multiple optimization objectives. Its load distribution is characterized by a form of *unbalancing* of load across hosts to facilitate agile live migrations. Fine tuning these techniques is very much a continually ongoing process and there are several avenues for further exploration, most notably improvement that can result from (i) introducing predictive techniques for live migration; (ii) co-designing predictive and reactive mechanisms so they complement each other well; (iii) more tightly integrating fleet-wide and host-level management; (iv) exploiting statistical multiplexing opportunities stemming from complementary resource needs (e.g., preferentially packing CPU-intensive and memory-intensive instances together) and (v) using sophisticated ML/RL-based techniques for workload prediction and decision-making.

REFERENCES

[1] 2021. DAMON-based Reclamation. https://lore.kernel.org/20211019150731.16699-16-sj@kernel.org/.

[2] 2024. Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[3] 2024. Amazon Aurora Pricing. https://aws.amazon.com/rds/aurora/pricing/.

[4] 2024. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.

[5] 2024. Aurora Serverless v2 Capacity. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.how-it-works.html#aurora-serverless-v2.how-it-works.capacity.

[6] 2024. AWS EC2 Regions and Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[7] 2024. AWS Nitro System: A combination of dedicated hardware and light-weight hypervisor enabling faster innovation and enhanced security. https://aws.amazon.com/ec2/nitro/.

[8] 2024. Control groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html.

[9] 2024. Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[11] Martin F. Arlitt and Tai Jin. 2000. A workload characterization study of the 1998 World Cup Web site. *IEEE Netw.* 14, 3 (2000), 30–37.

[12] Martin F. Arlitt and Carey L. Williamson. 1996. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, Daniel A. Reed and Blaine D. Gaither (Eds.). ACM, 126–137.

[13] Faruk Caglar and Aniruddha S. Gokhale. 2014. iOverbook: Intelligent Resource-Overbooking to Support Soft Real-Time Applications in the Cloud. In *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*. IEEE Computer Society, 538–545.

[14] Peter Desnoyers, Timothy Wood, Prashant J. Shenoy, Rahul Singh, Sangameshwar Patil, and Harrick M. Vin. 2012. Modellus: Automated modeling of complex internet data center applications. *ACM Trans. Web* 6, 2 (2012), 8:1–8:29.

[15] J. Heinanen, T. Finland, and R. Guerin. 1999. RFC 2687: A Single Rate Three Color Marker. https://datatracker.ietf.org/doc/html/rfc2697.

[16] David A. Hoeflin and Paul Reeser. 2012. Quantifying the performance impact of overbooking virtualized resources. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*. IEEE, 5523–5527.

[17] Klaus Jansen and Kim-Manuel Klein. 2013. A Robust AFPTAS for Online Bin Packing with Polynomial Migration. *CoRR* abs/1302.4213 (2013). arXiv:1302.4213 http://arxiv.org/abs/1302.4213

[18] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. 2022. Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2232–2245. https://doi.org/10.1145/3514221.3526047

[19] Sam Lightstone, Maheswaran Surendra, Yixin Diao, Sujay S. Parekh, Joseph L. Hellerstein, Kevin Rose, Adam J. Storm, and Christian Garcia-Arellano. 2007. Control Theory: a Foundational Technique for Self Managing Databases. In *Proceedings of the 23rd International Conference on Data Engineering Workshops, ICDE 2007, 15-20 April 2007, Istanbul, Turkey*. IEEE Computer Society, 395–403.

[20] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.* 34, 2 (2016), 6:1–6:33.

[21] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *Proc. VLDB Endow.* 15, 11 (2022), 3098–3111.

[22] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. [n.d.]. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, Jianping Wu and Wendy Hall (Eds.). 270–288.

[23] Michael Mitzenmacher. 1999. On the Analysis of Randomized Load Balancing Schemes. *Theory Comput. Syst.* 32, 3 (1999), 361–386.

[24] Xiao Qin, Hong Jiang, Yifeng Zhu, and David R. Swanson. 2006. Improving the performance of I/O-intensive applications on clusters of workstations. *Clust. Comput.* 9, 3 (2006), 297–311.

[25] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 16:1–16:16.

[26] Bianca Schroeder and Mor Harchol-Balter. 2004. Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness. *Clust. Comput.* 7, 2 (2004), 151–161.

[27] Haiying Shen and Liuhua Chen. 2018. Resource Demand Misalignment: An Important Factor to Consider for Reducing Resource Over-Provisioning in Cloud Datacenters. *IEEE/ACM Trans. Netw.* 26, 3 (2018), 1207–1221.

[28] Haiying Shen and Liuhua Chen. 2020. A Resource Usage Intensity Aware Load Balancing Method for Virtual Machine Migration in Cloud Datacenters. *IEEE Trans. Cloud Comput.* 8, 1 (2020), 17–31.

[29] Haiying Shen and Liuhua Chen. 2022. A Resource-Efficient Predictive Resource Provisioning System in Cloud Systems. *IEEE Trans. Parallel Distributed Syst.* 33, 12 (2022), 3886–3900. https://doi.org/10.1109/TPDS.2022.3172493

[30] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*. ACM, 5.

[31] Amoghavarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing, WOSC@Middleware 2019, Davis, CA, USA, December 09-13, 2019*. ACM, 19–24.

[32] Selome Kostentinos Tesfatsion, Luis Tomás, and Johan Tordsson. [n.d.]. OptiBook: Optimal resource booking for energy-efficient datacenters. In *25th IEEE/ACM International Symposium on Quality of Service, IWQoS 2017, Vilanova i la Geltrú, Spain, June 14-16, 2017*.

[33] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant J. Shenoy, Mike Spreitzer, and Asser N. Tantawi. 2005. An analytical model for multi-tier internet services and its applications. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada*. ACM, 291–302.

[34] Bhuvan Urgaonkar and Prashant J. Shenoy. 2004. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Trans. Parallel Distributed Syst.* 15, 1 (2004), 2–17.

[35] Bhuvan Urgaonkar, Prashant J. Shenoy, Abhishek Chandra, and Pawan Goyal. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC 2005), 13-16 June 2005, Seattle, WA, USA*. IEEE Computer Society, 217–228.

[36] Bhuvan Urgaonkar, Prashant J. Shenoy, and Timothy Roscoe. 2002. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*.

[37] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[38] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. 2009. Sandpiper: Black-box and gray-box resource management for virtual machines. *Comput. Networks* 53, 17 (2009), 2923–2938.

[39] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. 2018. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-Located Workloads. In *Proceedings of the 19th International Middleware Conference*. 146–160.

[40] Kejiang Ye, Xiaohong Jiang, Dawei Huang, Jianhai Chen, and Bei Wang. 2011. Live Migration of Multiple Virtual Machines with Resource Reservation in Cloud Computing Environments. In *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*, Ling Liu and Manish Parashar (Eds.). IEEE Computer Society, 267–274.