Managed Resource Scaling in Amazon EMR

Vishal Vyas*
vishavya@amazon.com
Amazon Web Services

Andrei Paduroiu*†
andreipm@amazon.com
Amazon Web Services

Srikanth Kandula
kansri@amazon.com
Amazon Web Services

Hari Ohm Prasath Rajagopal†
harrajag@amazon.com
Amazon Web Services

Mukesh Punhani
mpunhani@amazon.com
Amazon Web Services

Marco Manzo
mnzomar@amazon.com
Amazon Web Services

Ankur Goyal
ankugoya@amazon.com
Amazon Web Services

Santosh Chandrachood
sanchas@amazon.com
Amazon Web Services

Rick Sears
rsears@amazon.com
Amazon Web Services

Joseph Marques
jmarques@amazon.com
Amazon Web Services

Sushant Majithia
majithia@amazon.com
Amazon Web Services

Abstract

Compute elasticity is a primary benefit of using cloud-based data processing platforms such as Amazon EMR, where clusters can be scaled both horizontally and vertically. For example, a query scanning petabytes of data can run faster in a cluster with thousands of nodes compared to one with only a few hundred. However, not all workloads require the same computational power or have the same resource utilization patterns throughout their lifetime. Optimizing solely for performance by over-provisioning clusters can result in extra costs for customers stemming from unused capacity. Under-provisioning clusters, on the other hand, will keep costs low but can delay job completion time and cause SLAs to be missed. To further complicate the problem, a single cluster may run concurrent workloads, each with different resource needs. Typical solutions to managing time-varying resource needs involve frequent manual cluster rescaling or using different-sized clusters for different types of queries, which increases the cluster administrator's workload. This paper presents Amazon EMR Managed Scaling, a feature that continuously and automatically resizes EMR clusters with a goal to optimize the cost/performance ratio, with minimal user input. We describe how we iteratively built our engine-agnostic scaling approach that uses a cluster's telemetry, topology and past behavior to adjust its resource utilization in response to changing resource requirements. We show how each of the individual techniques presented herein solve specific workload execution patterns, and how EMR combines them into a single, unified algorithm that can make correct scaling decisions within seconds.

*Both authors contributed equally to the paper
†Work done while at Amazon Web Services

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724443

CCS Concepts

• **Information systems → DBMS engine architectures**; **Relational parallel and distributed DBMSs**; **Autonomous database administration**; **Online analytical processing engines**.

Keywords

cloud databases, elastic scaling, platform agnostic scaling, engine agnostic scaling, Amazon EMR, resource allocation

ACM Reference Format:
Vishal Vyas, Andrei Paduroiu, Srikanth Kandula, Hari Ohm Prasath Rajagopal, Mukesh Punhani, Marco Manzo, Ankur Goyal, Santosh Chandrachood, Rick Sears, Joseph Marques, and Sushant Majithia. 2025. Managed Resource Scaling in Amazon EMR. In *Proceedings of Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724443

1 Introduction

Amazon EMR[6] is a cloud-based data processing platform that enables customers to run distributed ETL jobs, interactive analytics and machine learning using open-source technologies like Apache Spark[20], Apache Hive[18], Apache Flink[16], Presto[26] and Trino[27], with the latest open table formats such as Apache Iceberg[19]. EMR builds and packages these technologies into ready-to-use images which are published periodically, adding to about 150+ such releases to date[4], and customers may choose to deploy them to several platforms that offer trade-offs between convenience and control over the compute configuration. EMR on EC2[7] gives customers fine-grained control over their compute by enabling them to tune most aspects relating to cluster provisioning and configuration, while EMR on EKS[3] abstracts out some of the compute details. Finally, EMR Serverless[9] offers the most convenience of the three by managing most compute-related decisions, with minimal intervention needed.

Regardless of the deployment model chosen, the cost an EMR customer would incur for their clusters depends on hardware configuration, capacity (the number of nodes) and how long this capacity is allocated for. Assuming that hardware configuration is the same, adding more capacity or keeping that capacity for longer generally

increases the cost, while reducing either tends to lower the cost. Customers are charged for allocated capacity whether or not they use it, so it is in their best interest to maximize a cluster's *utilization*, defined as the amount of time the nodes making up that cluster perform work useful to the customer, divided by the total amount of time those nodes are allocated.

Customers want to also maximize the *performance* of the workloads running in their EMR clusters, by making them run as fast as possible. In general, this can be achieved by increasing the resources available to them, however Amdahl's law applies to this case[33]: some workloads begin to scale sub-linearly after a certain point, and there are ceilings above which adding more capacity does not improve performance. The problem is exacerbated by the fact that workloads running in EMR seldom have uniform resource requirements, which can vary both between and within workloads. For example, the same cluster may run both short SQL queries that power an interactive dashboard and resource-intensive ETL jobs that are scheduled on a periodic basis. As well, the cluster may also run streaming jobs such as by Flink or Spark Streaming[25], which alternate frequently between an idle state (little to no work) and a resource-intensive state that requires massive parallelism.

This leads to the difficult task of finding a good balance between utilization and performance. Improving utilization by reducing capacity can negatively affect performance, while improving performance by increasing capacity can lead to under-utilization. Since most queries do not scale beyond a certain level of parallelism, over-indexing on performance can lead to excessive costs without added benefit. A simple solution to this problem is to optimize for only one of utilization or performance, in which case customers with strict SLAs can over-provision the cluster or segregate workloads into differently-sized clusters. Another solution is to balance cost and performance by periodically rescaling the cluster by adding or removing nodes. Such cluster re-scaling can improve outcomes compared to static sizing, but requires constant manual effort or investment in heuristics that adapt cluster size in response to observations of performance and utilization. The problem increases in complexity when we factor in (a) noise in telemetry signals, (b) lag in changing the cluster size, (c) and unknown or potentially unpredictable future resource needs of workloads; all of these factors affect how well one can determine the *optimal* cluster size.

In this paper, we describe our overall EMR architecture and infrastructure with specific focus on EMR's Managed Scaling (EMS) [34]. EMS can be enabled for any EMR cluster and automatically scales clusters aiming to identify a user-specified sweet-spot in the trade-off between performance and utilization. EMS aims to remove the guesswork out of *when*, *what* and *how much* to scale, by continuously sampling cluster telemetry and combining that with customer intent and historical data. The techniques in this paper are presented in the context of Amazon EMR, however are applicable to other multi-tenant, multi-application data processing platforms which have similar telemetry signals. Our contributions are:

(1) We describe EMS, a platform- and engine-agnostic, automated cluster scaling system used in Amazon EMR, designed to support data processing engines that dynamically adjust their parallelism.

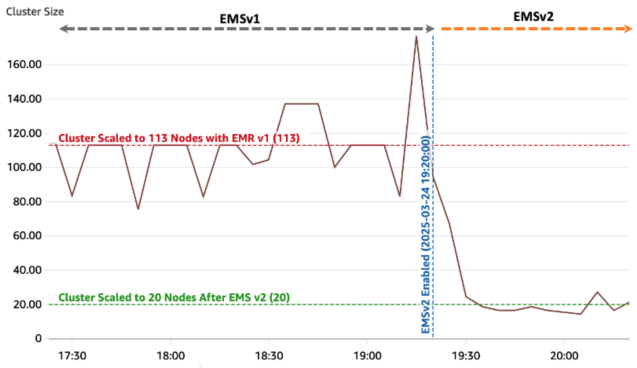

Figure 1: Changing the cluster scaling technique (EMSv1 → EMSv2) reduces by 82% the node usage of a TPCDS based benchmark while changing performance by less than 1%.

(2) We describe our production solution from first-principles including how we evolved the solution by challenging the assumptions in the initial release.
(3) We detail how we handle issues typical of multi-tenancy such as resource deadlocks during scaling events.
(4) We highlight telemetry-based techniques to track resource consumption and data placement that are helpful in handling opaque, unpredictable or non-uniform workloads, such as stream processing or code-based ETL pipelines. These include adjusting the scaling bias based on customer input.
(5) We evaluate the effectiveness of EMS on synthetic workloads, and show (1) that the *control-plane-only* scaling approach improves the cluster's resource utilization by up to 63% compared to fixed-sized clusters, and up to 31% when compared to other scaling technologies available in EMR, and (2) a further improvement by up to 62% can be achieved from *data-plane awareness*.

To preview the practical value from EMR's managed scaling (EMS), Figure 1 plots the requested nodes in an EMS-enabled benchmark cluster which periodically ran all TPCDS queries, ten times each, on a 3TB dataset over a five hour window around the time when the cluster was upgraded from EMSv1 (the initial version of EMS) to EMSv2. Once EMSv2 is enabled, we observe (1) a reduction of 82% in Requested Nodes to run the workload (113 → 20, on average) and (2) a smaller variability in cluster size (fewer oscillations and spikes from a range of 60 to 13). Much of these gains are from the techniques described in §5 which let EMSv2 continuously adapt the cluster resource consumption to better model the requirements of the application(s) running inside it.

The rest of this paper is organized as follows. In Section 2, we provide a background of how EMR workloads are executed, followed by design goals and a high-level overview of EMS in Section 3. Sections 4 and 5 present the evolution of the EMS algorithm by starting with a pure control-plane-based implementation that looks only at coarse telemetry of the clusters that it scales, and continuing with a data-plane-aware implementation that takes signals from the applications it is managing resources for. Finally, we describe an evaluation of these techniques in §6, discuss lessons learned in §7, related work in §8 and conclude the paper in §9.

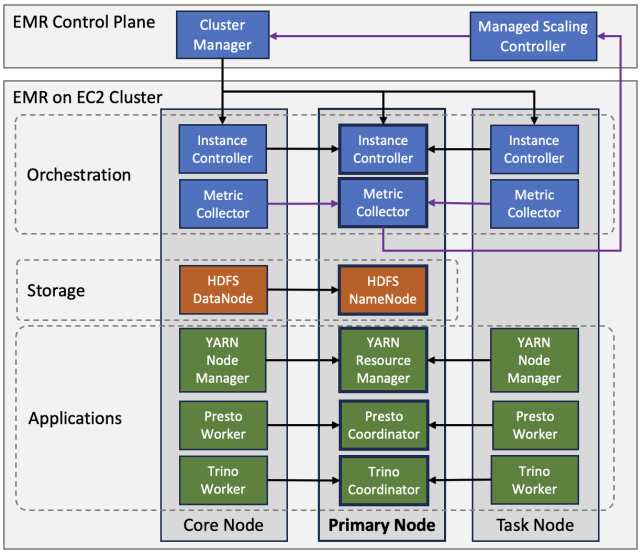

Figure 2: The node types in an EMR-EC2 cluster and how our managed scaling component functions: Instance Controller agents on each node listen to the control plane and perform up-keep of node's software components. *Task Nodes* (shown in the right column) only execute tasks and can host workers from any of the application running in the cluster. *Core Nodes* (in the left column) execute tasks and participate in data storage. The *Primary Node* (in the middle) hosts the coordinators for all installed applications and collects metrics from the other nodes, forwarding them to the EMS Controller.

2 Background (EMR and cluster autoscaling)

The central component of Amazon EMR is a *cluster*. A cluster is a collection of Amazon EC2 [7] instances. Each instance or node[1] has a role, referred to as the *node type*, which determines the software components that will be installed on the node during provisioning. Figure 2 shows the three node types in an EMR cluster (Core, Primary and Task) and their software components. 1/ A *Primary Node* shown in the middle column manages the cluster by hosting processes such as the YARN ResourceManager [17] and the HDFS NameNode [24] which coordinate the distribution of tasks and data among other nodes. The Primary node also monitors cluster health by processing telemetry from each node. A cluster may have up to three Primary Nodes (for high availability), and it is also possible to create a single-node cluster with only the Primary Node. 2/ *Core Nodes* shown in the left column run *tasks* from a rich class of applications as shown and also store HDFS data; multi-node clusters must have at least one Core Node. 3/ *Task Nodes* are optional; they run compute tasks but do not store any data in HDFS.

A cluster is created from an EC2 AMI [5] (*Amazon Machine Image*) which contains all the necessary software. AWS provides images that pre-package open-source analytical tools such as Presto, Trino, and YARN-based applications such as Spark, Hadoop MapReduce, Hive and Flink. EMR installs the AMI on each node and then starts the coordinators of enabled applications on the Primary Node and their associated workers on Core and Task Nodes. A cluster may run a plurality of supported applications concurrently and so can have several *application sub-clusters*, where each sub-cluster runs a different application. This is a key consideration for the design of EMS as we discuss later in the paper.

Each active cluster emits *telemetry* that tracks its activity. *Events* notify when a cluster changes state (started, running, terminated, etc.) or when workloads begin or end. *Metrics* are output on a periodic basis to gauge the overall cluster health, including storage, memory and compute utilization, but also to track the progress of individual workloads (or jobs), by indicating how many workloads are running, their resource demands, etc.

Such telemetry is essential to the process of *Cluster Scaling*, which lets customers add or remove nodes to their clusters throughout their lifecycle. EMR offers three rescaling options:

Manual resizing is an on-demand option triggered by the cluster administrator.

Auto-scaling, released in 2017, enables defining policies with triggers based on the metrics that EMR emits, with the administrator still responsible to determine the "right" rules and thresholds.

Managed scaling is a hands-off experience, requiring only the cluster min-size and max-size, with EMR continuously determining the optimal target size of the cluster based on the workload. The first version of Managed Scaling (EMSv1) was introduced in 2020, while the current offering (EMSv2) was released in 2022. We discuss both versions in this paper. We observe significant benefits from EMSv2 across various workloads in terms of both cost and performance and hence EMSv2 is the current default algorithm for customers workloads that run on EMR.

EMR can only scale applications that have *built-in elasticity* (i.e., can adjust their resource demands after starting), hence we put an emphasis on YARN-based technologies such as Spark, Hive, Hadoop or Flink in this paper. Such applications typically run their components in virtual *containers*, with one or more such containers fitting on a single node. Both (1) the demand for containers and (2) the maximum number of containers that can fit on a node (which varies based on the cluster spec and application settings), are essential inputs into the EMS algorithms as we will see shortly.

Regardless of the scaling method, EMR ensures that *decommissioning* a node will not adversely affect the applications running on the cluster. EMR only terminates a node when (1) all active containers running on the node have exited and (2) all data on the node has been replicated to other nodes, by building on top of decommissioning workflows from YARN and HDFS. The latency to shut down a node is hence non-trivial and is an important factor that EMS considers in its scaling decisions.

3 Overview of Managed Scaling

This section introduces EMR's managed scaling (EMS) by describing the challenges we aimed to solve, the associated design goals and our algorithms.

3.1 Challenges and Key Design Goals

The ideal scaling algorithm would adapt a cluster's size, as needed, to improve both the cluster's utilization and the performance of the currently running workloads. When we introduced the rules-based *Auto-Scaling* to EMR, in 2017, we found that administrators were

[1] we use instances and nodes interchangeably in this paper

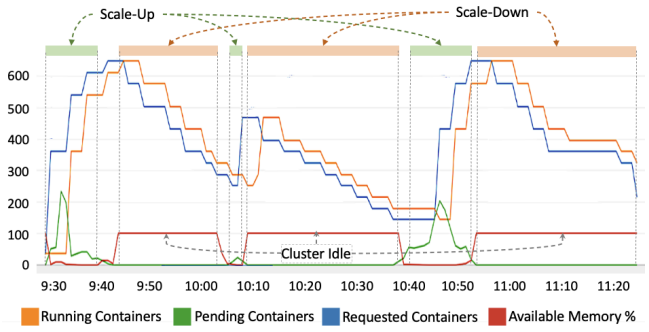

Figure 3: Example Auto-Scaling behavior: When three jobs ran one after the other on a cluster, notice that the first and third jobs incur sizable provisioning delay (peaks in their pending container metric) because the scaling rules held back the cluster, for approximately 15 minutes, from reaching its peak size of 600 nodes. Also, the cluster never fully downsizes during the idle period between jobs.

successfully relieved from manual resizing, but the scaling rules were non-trivial to setup correctly and easily lead to both over- and under-scaling especially when workload patterns varied, as shown in Figure 3. We used this opportunity to identify several challenges, which were key to designing the successor–EMS.

3.1.1 When and how much to scale. Timing a scaling operation is an intricate decision, as it may have a material impact on the cluster. A premature scale-up will immediately result in additional costs to the customer, but may not bring in an appropriate improvement in performance. Similarly, down-scaling too early reduces costs but risks impairing currently running workloads by slowing them and in some cases can block them altogether (e.g., due to insufficient disk space for shuffle exchanges).

The "ideal" cluster size should then be based upon 1/ the cluster utilization metrics such as the storage used and IOPS on disks and the occupancy of memory and CPU, 2/ and the resource needs of currently running applications. EMR clusters may host several running applications at a time, each with their own requirements and SLAs, so application-specific needs must also be accounted for as well. Finally, examining the requirements at any one point-in-time may not necessarily be indicative of future resource needs; for example, a query with tens of thousands of partitions may have just begun implying the need for an immediate scale-up, but if predicate push down successfully eliminates many of these partitions, the scale-up can become un-necessary since the resources needed can reduce by one or more orders of magnitude.

EMS evaluates several telemetry inputs and historical cluster data through a set of predefined rules that output a decision on whether to scale up, down, or take no action at all. EMS looks for trends (or emerging patterns) and incorporates heuristics to amortize oscillations and eliminate outlier signals that indicate a momentary condition, but not necessarily the beginning of a trend.

3.1.2 What to scale. Recall that nodes play different roles in the cluster, based on their type. The *Primary Nodes* manage the cluster and host the coordinators for running applications and scaling them down can even cause application termination without additional care in transferring state correctly. The *Core Nodes* host both storage and run tasks and so have a larger impact or cost if scaled (up or down) when compared to the *Task Nodes* which only host worker tasks. Also, different node types may be configured with different amounts of memory and CPU power which results in different amounts of provisioning availability and provisioning lag.

The *cluster topology* is another important aspect to consider here since the available storage space, network bandwidth and IOPS may become un-balanced with respect to the nodes that have available compute capacity. EMS categorizes the incoming telemetry across several dimensions to cover aspects such as stranded memory, CPU or storage capacity, and uses a dynamic set of rules to identify which node type needs a scaling adjustment based on the existing cluster topology, .

3.1.3 Handling non-uniform workloads. A node has three major phases in its lifecycle: provisioning, active and decommissioning. While only usable during its active phase, the customer still pays for the node throughout its lifecycle. The provisioning time for a node, T_P, depends on the node specification (CPU, memory, storage type) as well as the software that needs to be installed on the node. On the other hand, the decommissioning time, T_D, depends primarily on the amount of data that must be evacuated from the node (e.g., shuffle or other HDFS data). A complication here arises from typical task durations – T_T – being significantly lower than $T_P + T_D$. Often task durations are shorter by over one order of magnitude. As such, there is a high chance that by the time a node enters its active phase, that node may no longer be needed as exemplified also in Figure 3.

Predicting the duration of existing queries or anticipating the resourcing needs of a cluster running relational workloads has seen extensive research [1, 15, 30, 32, 36]. However, most of the solutions that we consulted focus on repetitive queries and analyze plans for patterns that were previously encountered. EMR hosts a sizable fraction of tasks that run customer code (e.g., Spark RDDs [37]) and are hence less amenable to such pattern analyses. Streaming jobs, for example using Flink[16] or Spark Streaming[25], add their own complexity by alternating unpredictably between idle states (little to no work, where only the coordinator is active), and resource-intensive states requiring massive task parallelism, disk space for shuffle data, etc. Even Spark Dataframe[11] (relational) jobs execute several distinct queries in short sequence, with each query touching or joining different tables with varying partition counts, or applying a different combination of operators to them. Such *non-uniform* workloads pose a difficult challenge for any algorithm attempting to optimize the utilization/performance ratio of a cluster.

EMS relies on determining *trends* in resource consumption; if repeated sampling indicates that a cluster's utilization is likely to change for some duration of time, the scaling algorithm compares that to the current node provisioning time to make a decision. More recently, EMR accepts additional input from the cluster's administrator specifying whether EMS should bias more towards better utilization or better performance; this "scaling slider" helps EMS to more effectively react to changes in resource consumption patterns based on criteria that customers care most about.

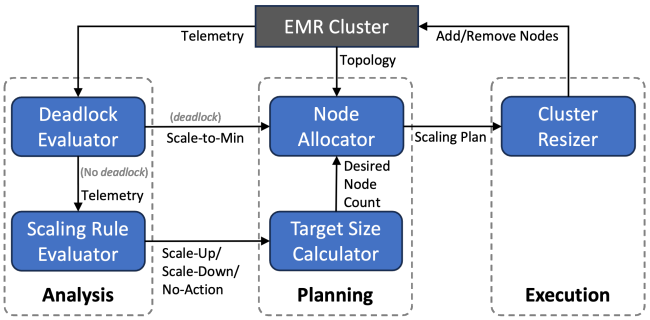

Figure 4: EMS performs a scaling event in three stages. The *Analysis* phase evaluates cluster telemetry and determines an *action* to perform. The *Planning* phase determines the target cluster size based on the action, and identifies which nodes (and of which type) must be added or removed. Finally, the *Execution* phase applies the resulting *Scaling Plan* to the cluster by physically adding or removing nodes.

3.1.4 Handling multi-tenancy. EMR clusters may host multiple applications at the same time, so they are prone to typical multi-tenant issues such as the noisy-neighbor problem [2]. Concurrent applications run independently of each other and have individual resource needs, however they all share the resources allocated to the cluster and thus "compete" with each other. A common situation that needs special attention is *resource deadlocks*, where an already started job requires more resources to make progress, but is unable to get any because the cluster is running at capacity (with resources being allocated to other applications). Meanwhile, this job holds on to resources which are not available to other applications running in the cluster. EMS identifies workloads that are not progressing and uses scaling to reallocate their resources and ensures that resource deadlocks do not arise.

3.2 Managed Scaling Architecture

EMS runs as a service inside the EMR Control Plane as shown in the top of Figure 2. EMS subscribes to the *telemetry* stream emitted from scaling-enabled clusters, and has a detailed knowledge about each cluster's *topology*. A typical scaling event runs in three successive phases, as depicted in Figure 4, and has one of three outcomes: add a certain number of nodes, remove select nodes, or take no action.

The *Analysis* phase processes raw telemetry data (CPU, memory, disk usage, among others) and makes a determination on what type of action is needed on the cluster. If a *resource deadlock* is determined, either the entire cluster (if single-tenant) or the stalled sub-cluster is scaled down to the minimum configured bound, to prevent excessive costs with no benefit to the customer. Otherwise a set of *scaling rules* are evaluated which determine if the cluster needs to be scaled up, down, or if no action is needed.

Once a decision on what action is needed, the *Planning* phase determines the extent to which to perform that action. It calculates the cluster's *target size* based on current and historical telemetry datapoints, and then compares the estimated resource needs with the available resources in the cluster (by consulting its *topology*). The outcome of this step is a *Scaling Plan*, which indicates (1) how many nodes of each type to add (for a scale-up), or (2), which specific nodes need to be removed (for a scale-down). The Scaling Plan is empty if no action is needed at this time.

The *Execution* phase applies a Scaling Plan to the cluster by physically adding or removing nodes. Scaled-down nodes are gracefully decommissioned by evacuating any HDFS data from them and either awaiting tasks to complete or reassigning them to other, equivalent nodes. Since this step is the same whether the request came from a Manual Rescaling, Auto-Scaling or Managed Scaling event, we have chosen not to detail it further in this paper.

4 EMR Managed Scaling V1 (EMSv1)

The initial release of EMR Managed Scaling V1 (EMSv1) follows the flow presented in Figure 4 and works off the telemetry collected from the clusters it manages.

4.1 Telemetry Collection

The *Metrics Collector* (MC) is an EMR process that continuously collects telemetry from managed clusters in the form of metrics and relays that in near-real-time to EMS. Metrics cover a wide range of data, but the ones most relevant to scaling relate to cluster health, those reported by the software infrastructural components (e.g., YARN), as well as application-specific information. The *Primary Metrics* that the MC prioritizes are:

- **Cluster Utilization**, calculated as the ratio of containers allocated to the total number of containers that can fit on all the nodes in the cluster. This metric helps determine whether compute resources are over- or under-utilized.
- **Pending Tasks and Container Demand** is retrieved mainly from YARN. This metric gauges the unmet needs of the applications that are running on the managed cluster.
- **Storage Utilization**, calculated as the ratio of storage used to total allocated storage space. This metric helps determine if the cluster's storage is over- or under-utilized. We also use related metrics to reason whether storage is adequately distributed across the nodes.
- **Application Metrics** include on-disk and in-memory shuffle data statistics, as well as information about execution stages and tasks. These metrics determine whether clusters have enough capacity to start new applications and if EMS can safely scale down certain node types. EMSv2 uses application Metrics at the granularity of individual data processing stages and partitions within each job as we discuss in §5.
- **Scheduler and Node type** metrics allow EMS to decide which instance group or instance fleets to scale up or down. This is critical as customers can use Yarn Node Labels or Yarn Queues to constrain their jobs or parts of jobs to run on different node types. A common example is customers choosing to run Drivers on *Core* nodes with all CORE nodes using EC2's *on-demand* instances while the Executors are retricted to run on *Task* nodes which are configured to run using EC2s *spot* instances.

4.2 Workload Analysis

EMS evaluates the running workloads and current cluster state based on a set of heuristics and decides whether to (1) scale up,

(2) scale down, or (3) maintain the current capacity (do nothing). As a general rule, sustained high Pending Tasks or Container Demand will label the cluster as eligible for *Scale-Up*, and a trending reduction in the same metrics will mark the cluster as eligible for *Scale-Down*. The next stages of EMS will decide the magnitude of action to perform since scaling-up is subject to real-time infrastructure availability and other constraints.

4.2.1 Deadlock Detection and Mitigations. We note a key limitation of any scaling technique that relies solely on rules and triggers defined on utilization metrics – the scaling technique's view of the available resources in the cluster may over time diverge from that of the applications running in the cluster leading to severe performance problems. For example, the *Auto-Scaling* feature which preceded EMSv1 would consider a node to be healthy and contributing even though Yarn may have marked that node as unhealthy due to failing disk health checks and even after Spark may have added the node (or containers) to a deny list due to repeated task failures. Such dissonance leads to poor performance in general. In an extreme instance, the auto-scaling algorithm recorded a cluster as having reached its scaling limit (i.e., the peak size allowed by the customer) but no workloads could be scheduled on the cluster nor were any existing applications making progress on that cluster.

EMSv1 detects such deadlocks early in the *Analysis Phase* using the metrics described in §4.1, and may take one of several actions. EMSv1 first tries to re-balance the Core and Task sub-groups of the cluster. Doing so triggers the self-healing mechanisms present in both YARN and HDFS. If this doesn't succeed, EMSv1 downscales either the cluster or just the stalled application, in order to both clear the state of the applications and reduce cost; a subsequent upscale triggered by the pent-up demand metric would allocate fresh nodes and/or rehydrate the application state from a clean beginning. EMSv2 further improves deadlock prevention (§5) by accounting for *data-plane awareness*.

4.3 Planning New Cluster Size

The next step of the EMSv1 algorithm is to determine whether the *Scale-Up* or *Scale-Down* signal received from the previous step constitutes an outlier or whether it is part of a trend. Although we considered several approaches in calculating the new cluster size, including using statistical models and building a single, all-encompassing formula, we settled on a simple, 5-step algorithm that first assigns a numeric value to each scaling dimension, then weighs them into a final number. We preferred the engineering benefits of this method, such as easier debuggability, observability and tuning, to the theoretical advantages of other approaches, a decision that proved crucial in the subsequent development of EMSv2. The *Target Node Count* is calculated as follows:

(1) The *Primary Metrics* from §4.1 are categorized into *dimensions*, depending on their type (CPU, memory, storage, etc.).
(2) For each dimension, we calculate a *Desired Node Count* (DNC) using empirically derived rules that take as input the latest and historical telemetry value of that dimension.
(3) We calculate the cluster's *Optimal Node Count* (ONC) as a weighted average of the DNCs, with each dimension's weight depending in part on the magnitude of change being requested (smaller differences lead to smaller weights) so as to balance the recommendations from the different signals while allowing any one signal to trigger an appropriate scaling response if necessary.
(4) We obtain an *Adjusted Node Count* (ANC) by constraining the ONC to customer-specified boundaries and accounting for real-time constraints, such as capacity pool availability.
(5) Finally, we calculate the *Target Node Count* by applying a dampening function to the ANC and comparing with recently calculated values to avoid oscillations and pick up on *resource demand trends*.

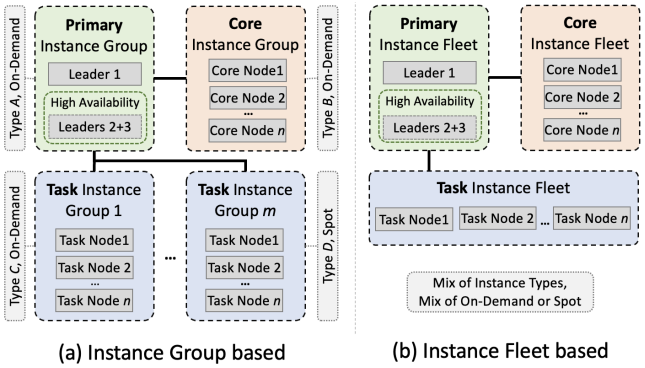

Figure 5: Physical Cluster Topology. Instance Groups (a) consist of homogeneous collections of nodes of the same instance type, lifecycle and availability zones. Instance Fleets (b) mirror a Logical Topology and are characterized by heterogeneity, allowing more flexibility in node choices.

4.4 Capacity Type Allocation

If a cluster's *Target Node Count* (see the end of §4.3 above) differs from its actual size, EMS will alter the cluster's topology and allocates new targets for each sub-cluster as described here.

4.4.1 A Primer on EMR Cluster Topology. A cluster's *logical topology* is an abstraction that applications interface with, and consists of the *Primary*, *Core* and *Task* node types that we had described earlier in §2). The cluster's *physical topology* is a concrete manifestation of the logical topology on EC2 hardware and includes instance types (node specs), availability zones and capacity allocation strategies. The EMR control plane manages the physical topology for a cluster, and we describe how how EMS makes the related decisions.

An *Instance Group* topology (Figure 5a) groups nodes by instance type and allocation mode (*On-Demand*, where an instance is owned until explicitly returned, or the cheaper *Spot*, where instances may be reclaimed by EC2 at any time for a variety of reasons). The Primary and Core nodes each have a single Instance Group, while up to 48 Task Instance Groups may be created.

An *Instance Fleet* topology (Figure 5b) is a fully-managed, heterogeneous layout that closely resembles the *logical topology*. Nodes can be mixed from different instance types, allocation types or availability zones, with each node type from the logical topology mapping to exactly one Instance Fleet component.

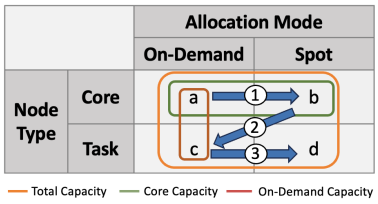

Figure 6: Default EMSv1 Allocation Strategy. Arrows depict spillover *policy*, with Core-On-Demand nodes scaled up first, followed by Core-Spot, Task-On-Demand and Task-Spot groups. Rectangles show spillover *thresholds*, consisting of customer-provided limits and EC2 capacity pool availability.

4.4.2 Scaling Strategy. Whether scaling clusters organized as Instance Groups or Instance Fleets, EMSv1 distributes the scaling target in a predetermined fashion across nodes based on their (1) node type and (2) allocation mode, as shown in Figure 6.

For scale-up, the *Core* nodes are prioritized ahead of *Task* nodes, and within each node type, *On-Demand* nodes are preferred to *Spot* nodes. For scale-down, we use the opposite precedence, with Task nodes scaling down before Core nodes, and within each node type, *Spot* nodes being released before *On-Demand* nodes. This strategy (which can be overridden by the customer) ensures that storage scales with compute when possible, and the nodes acquired from the EC2 pool as *On-Demand* are leased to EMR until it is done with them (vs. *Spot*, which can be taken away at any time).

Due to Instance Group homogeneity, when scaling clusters with multiple Task Groups, EMS chooses the group whose node specs have the deepest and most stable capacity pools at that time, which helps balance the available EC2 capacity for an instance type in a specific availability zone. Instance Fleets have a single, heterogeneous Task Fleet, which gives EMS more flexibility in choosing not only what node type to add, but also from which availability zone, which further improves EC2 capacity balancing.

4.5 Multi-tenancy support in EMS

Recall from §3.1.4 the challenges, including resource deadlocks, that arise when scaling clusters that are used by multiple applications. We discuss how EMS guards against these deadlocks in §4.2.1 and some pertinent changes in EMSv2 are in §5.2. Stepping back, EMS can be thought off as sizing the cluster based on the union or the sum of the resource demands of the various applications that share the cluster. Experiments in §6 specifically mix multiple diverse applications in the same cluster. Outside of deadlock avoidance, EMS plays no role in dividing a cluster's resources between applications – customers can use priority classes [22], weighted fairness [23] or dominant resource fairness [28] as desired.

5 EMR Managed Scaling V2 (EMSv2)

EMSv1 uses rules to determine if resizing is needed and, only if thresholds are breached, calculates the new cluster Target Node Count (§4.3). EMSv1 does so to avoid *thrashing* (alternating between repeated scale-ups and scale-downs), which often happens when the workloads are spiky and point-in-time metrics are used for adaptation. While this approach worked for the majority of EMR customers that used EMSv1, these thresholds, in some edge cases, lead to both over-scaling and under-scaling. In the two years since the launch of EMSv1, we had built over ten customer-specific variants of the cluster scaling algorithm for customers for whom EMSv1 wasn't working as intended. We invested in EMSv2, in part, to avoid maintaining multiple permutations of the scaling algorithm which had lead to a fragmented customer experience.

EMSv2, the second major release of EMS, differentiates itself from EMSv1 by introducing *data-plane awareness* which allows EMSv2 to use more detailed signals from the underlying data processing applications and make better scaling decisions that improve both utilization and stability. EMSv2 completely overhauls the EMS workflow by following a from-first-principles approach: (1) we challenged the assumptions made in the design of EMSv1 using observed data from production workloads, (2) we built an understanding of the wider variety of workload patterns that EMS must support and (3) we defined a new, unified, scaling model that meets diverse customer needs. Another key addition in EMSv2 is our *Workload Pattern Simulator* (§6.2), which helped us to define and fine-tune algorithms such as *Gradual Scale-Up*, *Aggressive Scale-Down* and *Dynamic Rules*. These additions are, in general, independently useful and together reduce cluster costs by over 40% on top of EMSv1.

We note that EMSv2 also contains reliability improvements that we have added to open-source software such as YARN or HDFS. In this paper, we focus only on the most important additions which we believe to be reusable in other settings.

5.1 Shuffle Data Awareness

Map-reduce [14] based applications such as Spark split input data into partitions and assign them to different executors for concurrent processing. Partitioning is usually over *group-by* columns or *join* predicate columns and is often followed by a *shuffle* or *exchange* operation. Some jobs may require re-partitioning if a subsequent *join* or *aggregation* requires a different partitioning scheme. Map (or partition) stages write their output to a *shuffle store* which the subsequent reduce stages read from. A majority of EMR clusters use the Node level External shuffle Service, which, in the case of Spark for example, is using the Shuffle Services Memory and the local Disk; that is, the mappers write shuffle data to their local disk. Thus, the success of the shuffle relies on the availability of the shuffle data on the mapper nodes. If any of these nodes are terminated, due to a cluster scaling event, the engine must re-run mapper tasks to re-compute the affected partitions by re-triggering the sequence of tasks that generated them [10].

In many analytics engines, including Spark, the above scenario holds for all intermediate stage outputs even those that do not require a shuffle: that is, 1/ the output of a stage is written to local memory or disk, and 2/ if a node is terminated before downstream stages that depend on these results can begin, the engine will have to re-compute the missing intermediate results.

Figure 7 shows a sample Spark job running on a dataset with data skew; roughly 75% of the tasks complete in < 23 seconds, but the longest task takes 1.2 hours to complete. To EMS, skewness manifests as significant under-utilization that necessitates a scale-down since a very small number of containers are running and most allocated capacity is idle. The scale-down and decommissioning

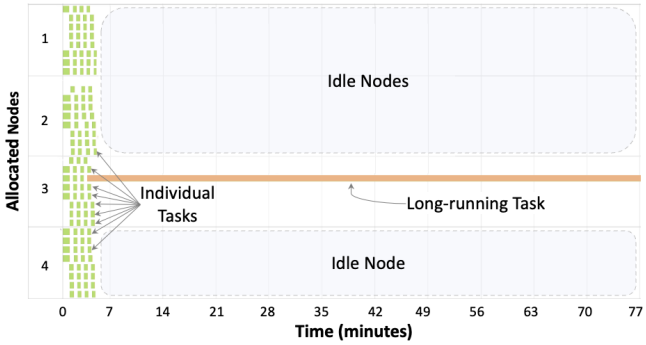

Figure 7: An example 5-stage Spark Job with data skew that ran on 4 nodes with 40 concurrent containers possible on each node. 128 of the tasks complete within 1 minute whereas the longest task takes 1.2 hours. Three out of four nodes were computationally-idle for a ~ 70 minute period but if any of the idle nodes were to be scaled-down, this job would either fail due to the lost shuffle data on these scaled-down nodes or will have to recompute all of the lost shuffle data.

workflow of EMSv1 is aware of YARN- and HDFS-provided metrics but is unaware of shuffle data and can remove idle nodes that contain the inputs for tasks that haven't begun yet. Shuffle data loss leads to re-computations, increasing job runtime and cost.

To resolve this, EMSv2 now uses the *Application Metrics* described in §4.1. We have modified the data plane engines to emit application level telemetry – such as *shuffle maps* – which represent the amount of active (unread) shuffle data on each node. Using Spark as an example, EMS scans an application's event log for the last (or currently) executing stage, identifies the executors involved, and examines the shuffle-related metrics to determine which of them accepted shuffle writes.

EMSv2 also uses an *enricher* component that modifies the *Scaling Plan* produced by the core algorithm (§4.3) based on additional criteria. The *Shuffle Enricher* consults the cluster's shuffle map and protects nodes that still have active shuffle data from being scaled-down. This approach allows EMS to make global optimizations in the control plane which do not intrude with data plane activity.

5.2 Changes to the Core Scaling Algorithm

EMSv2 is a statistical model that learns from evaluations of up to two years of telemetry from EMSv1. We look at the workload execution patterns, the behaviors of EMSv1, and whether EMSv1's responses were adequate or not. Internally, EMSv2 is a collection of sub-algorithms, each handling a specific class of patterns.

5.2.1 Gradual Scale-Up. EMSv1 was making point-in-time decisions on how many nodes to add to the cluster based solely on the cluster health telemetry. For EMSv2, we store *Application Metrics* in a cluster's timeline, and apply a *dampening function* that looks back (up to 24hrs) in the timeline to check for pattern matches (comparing recent Application Metrics to anything seen before). Doing so improves the amplitude of scaling decisions and reduces erroneous ones (i.e., avoiding unnecessary scale-ups for short-lived spikes in resource demand) by looking at cluster-specific patterns.

5.2.2 Aggressive Scale-Down. An important observation during our workload analyses was that resource demands are transient. That is, most demand spikes go down faster than the time to provision new nodes. Thus, aggressive scale-down directly hurt performance because scale-up is too slow to bring up the necessary resources when demand arrives. EMSv1 was hence conservative in scaling down and waited up to 10 minutes after a node was provisioned before considering it for removal during a scale-down. As we noted above, a static timeout, even a conservative one, could lead to recomputations if the terminated nodes contain unread shuffle data. With built-in Shuffle Data Awareness (§5.1), EMSv2 scales-down much more quickly and without recomputations – EMSv2 initiates (gradual) scale-down within one minute of inactivity which significantly reduces costs for workloads that have spiky demands.

5.2.3 Dynamic Rules. Part of our analysis was to understand *why* EMSv1 made the scaling decisions it made, and our primary conclusion was that one size does not fit all. Recall that EMSv1 applies rules on each metric from telemetry and combined the results using dampened weights as described in §4.3. We discovered that a large percentage of erroneous scaling triggers (i.e., a scale was triggered when none was warranted) were due to only one or two of the rules producing *Desired Node Counts* that differed from the current node count. A deep-dive revealed that this problem wasn't due to having too many rules or overly complex rules, but rather that some rules would produce accurate decisions on certain cluster configurations but had outsized errors on other cluster configurations. An example rule that was accurate for small and medium-sized clusters, but wasted resources for large or extra-large clusters prevented EMSv1 from triggering a scale-down unless 80% of the cluster memory was free – the total free memory is noisy and changed with cluster size.

EMSv2 dynamically selects the rules to apply based on the cluster configuration, thus avoiding sensitivity to rules that do not help a given cluster based on its configuration. EMSv2 determines rule eligibility based on current cluster size, available memory and storage space. Based on our testing, dynamic rule selection alone contributed up to 20% additional cost savings.

5.2.4 Resource Fragmentation aware Container Allocation. Many of the YARN applications, which EMS scales, run in containers. A close analysis of production data revealed that, for some clusters, even when the application metrics indicated a need for more nodes because no nodes were available for new containers, other metrics showed a much lower cluster utilization, often below 85%.

This happened due to two common reasons.

First, resource fragmentation occurs in clusters that use only equi-sized containers and use nodes whose specs are not a whole multiple of their container specs. EMSv2 dynamically detects how many containers fit in each node, and uses that to pick appropriately-sized nodes. When that was not possible, because the customer mandates a different instance type, EMSv2 corrects its utilization telemetry to account for the fragmentation waste.

Next, resource fragmentation occurred even in clusters that ran multiple applications or used different container sizes because the

containers were placed on nodes without a careful eye to avoid resource fragmentation. This is the classical bin-packing problem [13] and EMSv2 uses a tighter integration with the YARN allocator to address this issue. Specifically, EMS requests container-related information from the application, including specs and placements, overlays that on top of its own view of the cluster's Physical Topology and provides each YARN application with hints as to which node(s) to prioritize placing new containers on.

A full treatment of fragmentation-related under-utilization likely requires data plane changes [29]. EMS is a control plane scaling mechanism with no direct control on the actual task placement. Nevertheless, our experiments show that the above two solutions improve cluster utilization.

5.3 Customizing the Utilization/Cost Ratio

A key design principle behind EMS is to make optimization decisions on behalf of customers, to improve their clusters' utilization while keeping performance parity with a comparable baseline. However, even after the release of EMSv2, a subset of customers told us that, given a choice, they would prefer trading off performance to save on costs. Most of their workloads scaled sub-linearly, where the marginal cost of adding extra capacity outweighed the marginal benefit received on performance.

We enhanced EMSv2 with a *Utilization-Performance Index* (UPI) feature, which allows customers to control their cluster's scaling strategy to meet a desired balance between resource utilization and performance. Launched at AWS re:Invent'24, the UPI applies pre-configured multipliers to the weights of the different dimensions (see §4.3) that determine the new cluster size.

5.3.1 Capturing Customer Intent. To configure the UPI, EMR customers can choose how to bias EMS by using a *slider* on a scale of 1 to 100. Three of the most common choices are explained below:

- **Utilization-Optimized** (slider=1) prioritizes saving costs and resources; it uses a less aggressive scale-up behavior and primarily benefits workloads that have regular, short-lived spikes in resource demand.
- **Balanced** (slider=50) considers both resource utilization and job performance. It is suitable for several types of workloads, including those that have gradual changes in resource demands or have a mix of short and long-running stages. This most closely resembles the default behavior of EMSv2.
- **Performance-Optimized** (slider=100) prioritizes performance over cost and aggressively scales-up; it is designed for workloads that are latency sensitive or have strict performance targets or service-level-agreements (SLA).

5.3.2 Applying Scaling Bias. UPI customizes EMSv2 by translating the scaling bias accepted from the customer into a set of *multipliers* that weight the different dimensions used by EMS to make decisions, with the most important ones being:

- **Amplification Factor** controls the degree of aggressiveness in the scaling behavior, expressed as a fractional multiplier applied to the magnitude of any scaling operation.
- **Cool-Down Period** measures the amount of time that EMS awaits between re-evaluating whether a new scaling operation should be made or not.

Table 1: Parameter tuning when applying scaling bias

Parameter	Situation	Scaling Bias	
		Utilization	Performance
Amplification Factor	Scale-Up	Lower	Higher
	Scale-Down	Higher	Lower
Cool-Down Period	Scale-Up	Higher	Lower
	Scale-Down	Lower	Higher
Threshold Factor	Scale-Up	Higher	Lower
	Scale-Down	Lower	Higher

- **Threshold Factor** represents a minimum fractional value change in cluster size necessary to trigger a scaling operation. This is especially useful in dampening frequent oscillations.

Table 1 shows how these dimensions are tuned based on the scaling bias captured from the customer. While we are not able to reveal the actual values used in EMS, we provide directional ranges. During scale-up, the *Amplification Factor* is higher for performance-biased slider values, and both the *Cool-Down Period* and *Threshold Factors* are lower for performance-biased slider values. For scale-downs and for utilization-biased slider values, we set each parameter in the opposite direction.

6 Empirical Evaluation

In this section, we present a series of benchmarks comparing the use of EMSv1 with 1/ using no scaling and 2/ using the rule-based auto-scaling feature (§6.1). Next, we introduce our workload simulator (§6.2) and use the simulator to perform a comparative evaluation of EMSv2 (§6.3). Lastly, we evaluate the effect of applying scaling bias (§6.4).

6.1 EMSv1

We benchmarked EMSv1-enabled clusters against (1) Fixed-Size clusters with no scaling enabled and (2) Auto-Scaling enabled clusters. In all tests, we configured the clusters with default settings (except scaling), 16 VCPUs [8] per node with 4 VCPUs per YARN Container, and use Spark as the application to scale. Fixed-Size clusters were setup with 20 nodes, while Auto-Scaling clusters were configured with a range of 1-20 nodes, with an initial capacity set to 20. We submitted multiple parallel jobs running the entire TPC-DS [35] suite (each query in sequence) against a 3TB dataset in three batches, with a pre-set idle time between batches, without shutting down or otherwise interfering with the cluster in any way.

6.1.1 Comparison with Fixed-Size and Auto-Scaled clusters. We calculate the *Cluster Utilization* as the ratio of *(VCPU Time Used)* to *(VCPU Time Allocated)*, and report *Utilization Improvement* as the ratio of *(Utilization(EMS Test) −Utilization(Baseline Test))* to *Utilization(Baseline Test)*, expressed as a percentage. As a proxy for *performance*, we measure the end-to-end time for the whole test, including both the execution time of the job batches and the constant idle time between batches. We report *Time Difference* again as a ratio between *(Duration(EMS Test)-Duration(Baseline Test))* and *Duration(Baseline Test)*, expressed as a percentage.

Table 2: EMSv1 vs. Fixed-Size or Auto-Scaling Clusters

Baseline	EMS Config			Idle Time	Utilization	Time
	Min	Max	Init.	(minutes)	Improv.	Diff.
Fixed-Size	1	20	20	10	21.1%	+0.5%
	1	20	20	30	59.9%	-3.7%
	1	20	1	10	63.9%	-3.9%
	5	20	20	10	47.3%	-1.3%
Auto-Scaling	5	20	20	10	27.4%	-3.5%
	5	20	20	30	31.2%	+7.1%

Figure 8: A closer look at EMSv1, when configured to scale a cluster from 1 to 20 nodes, with an initial setup of 1 node wherein each node has 16 VCPUs and can store up to 4 containers. We see a spike in *Requested VCPUs* at the start of each job batch immediately translates into a scale-up indicated by a change in the *Container Pending Ratio*. When the application completes, EMS scale-down is triggered by the drop in usage of VCPUs and allocated memory as shown by the drop in the *Running Containers* metric.

Table 2 shows EMSv1 improving utilization between 21.1% and 63% when compared with equivalent Fixed-Size clusters, and between 27.4% and 31.2% when compared with clusters with Auto-Scaling enabled. EMSv1 maintained performance parity on average and improved it slightly in some cases. Not surprisingly, EMSv1 has the biggest impact (vs. Fixed-Size) when either the idle time is high (in this case, 30 minutes) or when it starts with a small cluster (59.9% and 63.9% utilization improvement, respectively).

EMSv1 showed a 7.1% increase in the total time when compared with Auto-Scaling at 30-minute idle time between the job batches because it fully scaling down the cluster during the idle period (while Auto-Scaling was more conservative in scaling down), and thus EMSv1 needed more time to scale up and provision new nodes when demand for resources picked up. However, this effect was offset by a 31.2% improvement in overall utilization, resulting in a net improvement in the ratio of utilization-over-performance.

6.1.2 A close look at EMS behavior. Here, we dive deeper into how EMSv1 scales-up and scales-down by examining the cluster telemetry emitted for test #2 from Table 2. We look at the case when EMSv1 was configured to scale between 1 and 20 nodes, starting with 1 node, on a cluster where we ran three successive batches of TPC-DS Spark jobs, with a 30 minute idle time between batches.

Figure 8 visualizes how EMSv1 reacted to the changes in workload and scales the cluster up when needed and releases resources when idle. EMSv1 uses the *Requested VCPUs* and *Available Memory %* metrics as a proxy for application resource demand as well as the *Running Containers* metric, which indicates how many application-level YARN containers were active (each container uses 4 VCPUs and up to 4 such containers fit on a single Node). When *Batch 1* starts, the cluster has only one node running, with no containers active. The application immediately requests up to 300 VCPUs, and EMS responds by allocating 75 containers (~19 nodes) in 3 minutes. About 7 minutes later, the running containers count indicates that the application has completed (the *Requested VCPUs* also drops to zero, correlated with the *Available Memory %* going to 100%), thus EMS scales-down this cluster. EMS resizes the cluster down to a single node in about 2 minutes, and the cluster stays in that state until the next batch of jobs begin. The same pattern repeats in *Batch 2* and *Batch 3*.

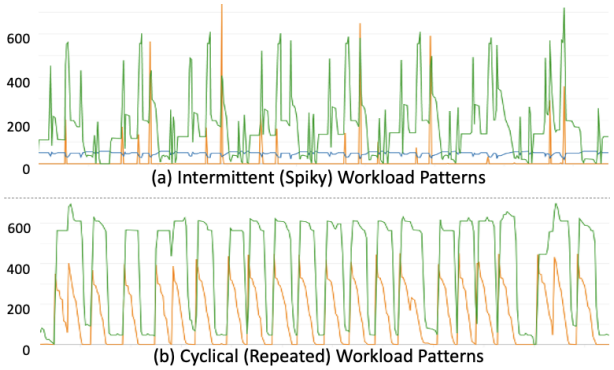

Figure 9: Depiction of EMS Simulation Framework Patterns.

6.2 Workload Pattern Simulation

EMSv1 was a significant improvement over the technologies available in EMR at the time but we knew that it could be enhanced further. To ensure that any changes to EMSv1 1/ improve efficacy on the targeted workloads and 2/ do not regress other workloads, we built an *EMS Simulation Framework* (ESF). ESF generates synthetic workloads that follow specific target patterns and we evaluate each development on EMS by executing the ESF-generated synthetic workloads on EMR clusters. ESF was a crucial enabler in the design and validation of *EMSv2* in 2022, and is still in use today.

To build the simulator (ESF), we analyzed telemetry from hundreds of EMS-enabled production clusters and identified two key workload patterns:

- **Intermittent workloads** (Figure 9a) have stable resource requirements most of the time but can spike suddenly by an order of magnitude or more for a short duration, after which they revert back to their stable pattern. Streaming workloads, and interactive (SQL) workloads fit this pattern.
- **Cyclical workloads** (Figure 9b) alternate between two (or a handful of) different resource requirement levels at relatively constant interval. ETL jobs that are triggered programmatically fit this pattern.

ESF further categorizes these patterns along a second dimension that accounts for *oscillation frequency*, which is the average time between spikes for Intermittent Patterns, or the average time between peaks for Cyclical Patterns. We define four sub-patterns

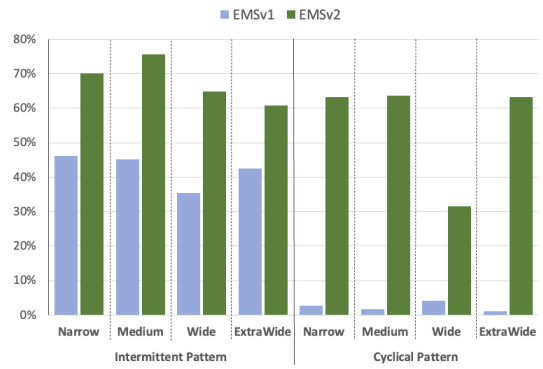

Figure 10: Comparing cluster utilization when enabling EMSv1 vs. EMSv2 on different workload patterns.

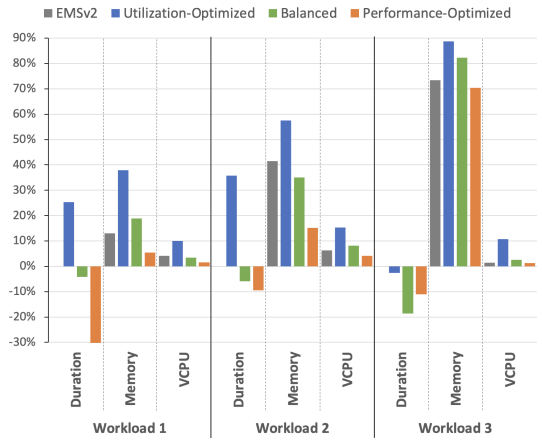

Figure 11: Comparing the three scaling bias presets from §5.3 with the *default* EMSv2 behavior on three types of workloads; see §6.4.

(*Narrow*, *Medium*, *Wide* and *Extra Wide*), with each increasing the frequency by one order of magnitude over the previous one. We found that such a two-dimensional classification can approximately simulate more than 95% of the production traffic that EMS supports and have decided to use these to benchmark scaling mechanisms.

6.3 From EMSv1 to EMSv2

We use the ESF (from §6.2) to compare *EMSv2* with *EMSv1* by running a total of eight tests that combine all of the synthetic patterns (Intermittent, Cyclical) with all supported frequencies (Narrow, Medium, Wide and Extra Wide). Our primary metrics are the same as in §6.1.1 and are calculated in the same way.

As shown in Figure 10, EMSv2 exhibited an 18%–31% improvement in Cluster Utilization for *Intermittent Pattern* workloads; the utilization across different oscillation frequencies improves from an average of 42.1% to an average of 67.5%. For the *Cyclical Pattern* workloads, the corresponding improvements are 28%–62% with the average utilization (across oscillation frequencies) improving on average from 2.1% to 48.2%. The biggest contributing factor to these improvements was making EMSv2 data-plane aware wherein we take cues from the application on when one can scale-down aggressively (when node has no unread shuffle data) or must wait to avoid job recomputations. Without *shuffle data awareness*, recall that EMSv1 would wait up to 10 minutes of inactivity before considering a node for decommissioning which is both too slow (when node has no unread shuffle data) and too fast (i.e., will cause recomputations if the node being terminated has unread shuffle data). In cyclical workload patterns, EMSv2 safely and aggressively scales down during the idle time between jobs wherein there is no unread shuffle data. We call out that the other aspects discussed in §5.2 also improve utilization albeit to a smaller extent on this particular synthetic experiment.

6.4 Effect of Changing the Scaling Bias

Here, we evaluate how the three preset values for the *Utilization-Performance Index* from §5.3 compare to using an unbiased EMSv2 on different types of workloads. We conducted these experiments in a controlled environment with EMSv2 configured with a min-max range on cluster size of $[1, 2000]$ and an *Instance Group* topology.

For each test, we measure (1) the job duration (as a proxy for performance) and (2) the average memory and VCPU utilization for the cluster. Figure 11 shows the results for each workload:

- **Workload 1** involves approximately 90 short and long-running stages processing 10TB of data. For the *Utilization-Optimized* preset, the cluster's average memory utilization increased 3× to 38% and the VCPU utilization 2.5× to 10%, but the job duration also increased by 25%. Enabling the *Performance-Optimized* preset saw job duration decrease by 32%, with memory and VCPU utilization also decreasing by 2.3× to 5.5% and 2.6× to 1.55%, respectively.
- **Workload 2** is similar to Workload 1 but only parses 3TB of data. The *Utilization-Optimized* preset improved memory utilization by 38% to 57% and VCPU utilization 2.4× to 15%, with the job duration increasing by 36% (but shorter in absolute terms than for Workload 1, due to the smaller dataset size). Moving the presets towards *Performance-Optimized* decreased the duration by 10% and overall memory and VCPU utilization by 2.7× to 15% and 1.5× to 4%, respectively.
- **Workload 3** executes Workload 2 twice, in parallel. Interestingly, even though each execution had its own alternating pattern of resource demand, when run concurrently, those patterns evened out by naturally filling in the idle gaps that would otherwise be left by running the workload once. Both *Utilization-Optimized* and *Performance-Optimized* improved the duration by 3% and 11%, respectively. All presets yielded memory utilization within 10% of the baseline, however the *Utilization-Optimized* preset yielded a 7.8× improvement in VCPU utilization.

7 Lessons and Takeaways

The field of cluster resource management is filled with a rich set of techniques that can generalize and apply in settings outside of their original target. EMS is no different; while some of the techniques that we leveraged have roots in previous research, we worked backwards from the practical observations and constraints in our

production EMR system to offer a a novel capability to customers that effectively optimizes the utilization and performance curves for cloud-based data processing applications. We highlight some of the lessons that we learned along this journey.

Start simple, observe, iterate, optimize. Fully-managed scaling is a complex operation, and one of our key tenets was to avoid premature optimizations. The rule-based EMR Auto-Scaling (predating EMS) provided valuable insights as to where we should start adding more automation into the process. We observed customer workloads, challenged previous assumptions and made improvements to the EMS algorithm, completely overhauling it over time.

Leverage reproducible synthetic workloads early and often. A key driver to our ability to iterate quickly and with a high confidence of not regressing was the EMS Simulation Framework (§6.2). Building this capability early enabled us to generate workloads that can approximate a large fraction of the workload patterns in production that we targeted EMS towards. We used ESF as the basis to decide which parts of the EMS algorithm needed improvement and to predict the impact each change may have on real workloads.

Use data-plane cues to scale effectively. EMSv1 was designed as an application-agnostic scaler that only examines cluster-level utilization metrics. While the tenet behind this was *simplicity*, EMSv1 introduced the potential for "split-brain" scaling, where EMSv1 guesses the state of the cluster differently from how the actual application uses the cluster. EMSv2 takes an important step forward by allowing the data plane to indicate redundant nodes, enabling their immediate removal and reducing costs for certain patterns. Some recent work [32] shows that results can potentially improve further by leveraging the execution plans of workloads to predict future resource demands rather than extrapolating the demands from historical trends.

8 Related Work

Some works adaptively size the resources for recurring or long-running jobs to meet deadlines economically. They differ from EMS primarily on scope and applicability. Several require knowledge of recurring/repeating jobs and do not apply to new/ad-hoc jobs. For example, Morpheus [30] describes a method to automatically right-size the resource profile of recurring jobs so that they finish within a deadline. This technique adapts the maximum resources available to a job (e.g., max degree of parallelism) based on previous runs of the recurring jobs. Jockey [15] uses a simulator to predict how much longer the unfinished tasks of an active job may take and adaptively varies the degree of parallelism so that the job may finish within a pre-specified deadline. TetriSched [36] considers a more complex case by planning/sizing resources of different types. Sarathi-Serve[1] considers similar problems in a different context by tuning the training and inference of large models.

Several other works size the resources for cloud databases based on their usage [31]. The typical challenges here are to pack databases into servers efficiently while allowing their usage to grow while minimizing movement (e.g., move a database from one server to another to add resources) which would disrupt ongoing activity. These works differ primarily from EMS on the workload – single database transactional queries versus distributed analytical/streaming-style queries – as well as on challenges unique to our distributed context and from having to support numerous data processing engines, such as Spark, Hive, Presto or Trino.

Nathan et al. [32] describe an auto-scaling logic for Amazon Redshift. While this work also considers a multi-machine setup similar to EMS, the key contributions are different in that they rely on (1) a per-query model that determines the resources to allocate to individual queries and (2) a workload forecaster that adapts the base cluster size. EMS eschews predicting at query granularity and also does not use detailed forecasting; rather, to be platform and engine agnostic, EMS adapts based on easily available telemetry.

Barnhart et al. [12] describe resource management in Amazon Aurora Serverless, with a focus on single machine scaling. The key techniques include (1) increasing the capacity units allocated to a database up to the max on a machine while allowing for live-migration when total load of instances on a machine exceeds capacity and (2) rate limiting the growth so as to avoid performance interference. EMS considers auto-scaling across multiple machines with a focus on (1) horizontal scaling (adding more instances) and (2) platform and engine agnosticism.

9 Conclusion

In this paper, we describe the evolution of EMR Managed Scaling (EMS), a feature available for every EMR cluster that automatically resizes clusters to reach a desired combination of performance and utilization. While the problems addressed by EMS are not inherently new, EMS primarily differentiates itself from other database scaling methods in that it is both platform and engine agnostic: EMS can scale any Spark, Flink, Hive and Hadoop applications running within an EMR cluster. EMS can work with cluster telemetry alone if needed, or it can take signals from the data plane to further optimize scaling decisions, and, to our knowledge, is the first cloud-based scaler that can handle the elasticity of both relational and opaque data processing workloads. EMS is able to balance the needs of a plurality of independent sub-clusters that share a single resource pool. We show how EMS can offer customers substantial improvements in cost, when compared to other methods, without affecting performance or requiring manual intervention. Fine-tuning the EMS algorithm remains ongoing work and, to that end, we describe a simulator that synthesizes workloads that resemble production traffic. We call out a few potential avenues for further exploration: (i) integrate better with engine-specific scalers (such as Spark's Dynamic Resource Allocation[21]) and (ii) leverage past telemetry as well as workload metadata for more accurate, real-time modeling of future demands.

Acknowledgments

We are grateful to the numerous Amazon EMR customers whose feedback helped the team build EMR Managed Scaling. We thank our leadership including Mehul Y. Shah and Abhishek Sinha for supporting us. Finally, we thank the many unnamed builders both past and present whose contributions have been crucial to the success of EC2, EMR and this project on managed scaling.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Amazon. 2023. SaaS Lens: AWS Well-Architected Framework. https://docs.aws.amazon.com/pdfs/wellarchitected/latest/saas-lens/wellarchitected-saas-lens.pdf. Accessed November 1, 2024.

[3] Amazon. 2024. Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks. Accessed: October 15, 2024.

[4] Amazon. 2024. Amazon EMR archive of release notes. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-whatsnew-history.html. Accessed October 15, 2024.

[5] Amazon. 2024. Amazon Machine Images in Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html. Accessed November 12, 2024.

[6] Amazon. 2024. Big Data Platform – Amazon EMR - AWS. https://aws.amazon.com/emr. Accessed: October 15, 2024.

[7] Amazon. 2024. Cloud Compute Capacity – Amazon EC2 - AWS. https://aws.amazon.com/ec2. Accessed: October 15, 2024.

[8] Amazon. 2024. CPU options for Amazon EC2 instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html. Accessed November 12, 2024.

[9] Amazon. 2024. Open-Source Big Data Analytics | Amazon EMR Serverless | Amazon Web Services. https://aws.amazon.com/emr/serverless. Accessed: October 15, 2024.

[10] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 265–278.

[11] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, 1383–1394. https://doi.org/10.1145/2723372.2742797

[12] 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. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4038–4050. https://doi.org/10.14778/3685800.3685825

[13] Edward G. Coffman Jr., János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. *Bin Packing Approximation Algorithms: Survey and Classification.* Springer New York, New York, NY, 455–531. https://doi.org/10.1007/978-1-4419-7997-1_35

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[15] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/2168836.2168847

[16] Apache Software Foundation. 2024. Apache Flink. https://flink.apache.org/. Accessed: October 18, 2024.

[17] Apache Software Foundation. 2024. Apache Hadoop YARN. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: October 15, 2024.

[18] Apache Software Foundation. 2024. Apache Hive. https://hive.apache.org. Accessed: October 15, 2024.

[19] Apache Software Foundation. 2024. Apache Iceberg. https://iceberg.apache.org. Accessed: October 15, 2024.

[20] Apache Software Foundation. 2024. Apache Spark. https://spark.apache.org. Accessed: October 15, 2024.

[21] Apache Software Foundation. 2024. Apache Spark. https://spark.apache.org/docs/latest/job-scheduling.html. Accessed: November 15, 2024.

[22] Apache Software Foundation. 2024. Hadoop: Capacity Scheduler. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. Accessed: April 7, 2025.

[23] Apache Software Foundation. 2024. Hadoop: Fair Scheduler. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. Accessed: April 7, 2025.

[24] Apache Software Foundation. 2024. HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed: October 16, 2024.

[25] Apache Software Foundation. 2024. Spark Structured Streaming. https://spark.apache.org/streaming. Accessed: October 16, 2024.

[26] Presto Foundation. 2024. Presto: Free, Open-Source SQL Query Engine for Any Data. https://prestodb.io/. Accessed: October 15, 2024.

[27] Trino Software Foundation. 2024. Trino | Distributed SQL query engine for big data. https://trino.io. Accessed: October 15, 2024.

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[30] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 117–134. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi

[31] Arnd Christian König, Yi Shan, Karan Newatia, Luke Marshall, and Vivek Narasayya. 2023. Solver-In-The-Loop Cluster Resource Management for Database-as-a-Service. In *50th International Conference on Very Large Databases*. VLDB Endowment, Inc. https://www.microsoft.com/en-us/research/publication/solver-in-the-loop-cluster-resource-management-for-database-as-a-service/

[32] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/3626246.3653394

[33] Daniel Richins, Tahrina Ahmed, Russell Clapp, and Vijay Janapa Reddi. 2018. Amdahl's Law in Big Data Analytics: Alive and Kicking in TPCx-BB (BigBench). In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 630–642. https://doi.org/10.1109/HPCA.2018.00060

[34] Abhishek Sinha, Joseph Marques, Srinivas Addanki, and Vishal Vyas. 2020. Introducing Amazon EMR Managed Scaling – Automatically Resize Clusters to Lower Cost. https://aws.amazon.com/blogs/big-data/introducing-amazon-emr-managed-scaling-automatically-resize-clusters-to-lower-cost/. Accessed October 16, 2024.

[35] TPC. 2021. TPC Benchmark DS. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf. Accessed: October 15, 2024.

[36] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 15–28.