Intelligent Pooling: Proactive Resource Provisioning in Large-scale Cloud Service

Deepak Ravikumar [*†]
Purdue University, USA
dravikum@purdue.edu

Alex Yeo[*†]
Netflix, USA
alexsyeo@gmail.com

Yiwen Zhu[*]
Microsoft, USA
yiwzh@microsoft.com

Aditya Lakra
Microsoft, USA
adityalakra@microsoft.com

Harsha Nagulapalli
Microsoft, USA
hanagula@microsoft.com

Santhosh Ravindran
Microsoft, USA
saravi@microsoft.com

Steve Suh
Microsoft, USA
stsuh@microsoft.com

Niharika Dutta
Microsoft, USA
nidutta@microsoft.com

Andrew Fogarty
Microsoft, USA
anfog@microsoft.com

Yoonjae Park
Microsoft, USA
yoonjae.park@microsoft.com

Sumeet Khushalani
Microsoft, USA
sukhusha@microsoft.com

Arijit Tarafdar
Microsoft, USA
arijitt@microsoft.com

Kunal Parekh
Microsoft, India
kunalparekh@microsoft.com

Subru Krishnan
Microsoft, Spain
subru@microsoft.com

ABSTRACT

The proliferation of big data and analytic workloads has driven the need for cloud compute and cluster-based job processing. With Apache Spark, users can process terabytes of data at ease with hundreds of parallel executors. Providing low latency access to Spark clusters and sessions is a challenging problem due to the large overheads of cluster creation and session startup. In this paper, we introduce Intelligent Pooling, a system for proactively provisioning compute resources to combat the aforementioned overheads. Our system (1) predicts usage patterns using an innovative hybrid Machine Learning (ML) model with low latency and high accuracy; and (2) optimizes the pool size dynamically to meet customer demand while reducing extraneous COGS.

The proposed system auto-tunes its hyper-parameters to balance between performance and operational cost with minimal to no engineering input. Evaluated using large-scale production data, Intelligent Pooling achieves up to 43% reduction in cluster idle time compared to static pooling when targeting 99% pool hit rate. Currently deployed in production, Intelligent Pooling is on track to save tens of million dollars in COGS per year as compared to traditional pre-provisioned pools.

PVLDB Reference Format:
Deepak Ravikumar, Alex Yeo, Yiwen Zhu, Aditya Lakra, Harsha Nagulapalli, Santhosh Ravindran, Steve Suh, Niharika Dutta, Andrew Fogarty, Yoonjae Park, Sumeet Khushalani, Arijit Tarafdar, Kunal Parekh, and Subru Krishnan. Intelligent Pooling: Proactive Resource Provisioning in Large-scale Cloud Service. PVLDB, 17(7): 1618 - 1627, 2024.
doi:10.14778/3654621.3654629

[*]Authors contributed equally to this research.
[†]Work done while at Microsoft

Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654629

1 INTRODUCTION

Cloud computing has emerged as a top choice for executing big data analytic workloads in various business domains. To cope with rapidly increasing demand, cloud vendors (e.g., Amazon AWS [3], Microsoft Azure [36] and Google GCP [20]) have funneled sizable resources into their own managed Spark services [38, 48], including Google Cloud's Serverless Spark [21], Spark through Vertex AI [22], Azure HDInsight [33], Azure Synapse Analytics [34] and AWS EMR [6]. The flexibility of such cloud offerings allows users to easily lease and release compute resources as required and, consequently, enjoy potentially significant cost-effectiveness. However, to provide such flexibility, service providers must address various challenges with respect to resource provisioning.

The implementation of multi-tenancy and scalability in such systems results in prolonged latencies in accessing clusters. With Azure Synapse [34], it is common to experience a cluster initialization time of over 60 seconds. According to Databricks [14], this provisioning time can be even longer than the duration of the job execution. However, proactive provisioning solutions are often challenging due to: (1) The unpredictability of user behavior and, (2) the difficulty in developing any policy that both enhances performance and decreases cost-of-goods-sold (COGS), which requires an explainable, comprehensive decision-making process in real-world production. Accurately modeling multi-tenant cloud performance and its impact on customer experience can be complicated involving complex modeling [13, 52].

State-of-the-art approach. Proactive auto-scaling (after application starts) has been introduced in data stream processing engines

(e.g., Apache Storm [5], Apache Flink [4]) and network provisioning [10, 51] to dynamically *scaling up* (or down) the compute resources when the workload is heavy [27], and performance modeling is developed to ensure the QoS requirements are fulfilled [28]. Time-series forecasting is used to determine the possible repeating patterns as inputs [8]. A detailed review of similar applications in streaming systems can be seen in [29].

For Spark [49] clusters, there has not been an automated solution to manage cluster provisioning. Some vendors, like Databricks [14], provide mechanisms to maintain clusters until a fixed threshold of idle time is reached and offer customers instruction on "best practices" for managing Spark clusters on their own. Despite its potential benefits, this approach still requires manual tuning from the user's perspective and may result in unsatisfactory performance.

In this paper, we tackle the issue of improving the customer experience of waiting for the *initialization* of Spark clusters where a cluster needs to be prepared for a newly submitted Spark job at its startup time, which is one of the major bottlenecks for many Spark systems. Compared to auto-scaling while the application is running, this problem can be more challenging because at the application submission time or even before (if supporting proactive provisioning), there can be little-to-no information known for a particular customer or application. Auto-scaling relies on real-time information such as cardinality estimates, number of tasks queued, etc., which becomes available when the application starts, to adjust the number of nodes and executors. Auto-scaling during the lifetime of an application is out of scope for this paper, and for Fabric, there is a different service to scale up/down the number of nodes in the compute cluster as well as the number of executors in real time based on the incoming workload characteristics and the tracking of task execution, which include richer information about the application and the Service Level Agreements (SLAs) that need to be met [9, 39, 44].

Challenges. In order to reduce the wait time for cluster initialization, we aim to use machine learning to proactively provision Spark clusters. However, this presents a number of challenges, including::

Uncertainty of user behavior [C1]. Intuitively, it would be possible to provision a cluster in advance if we could accurately predict when a customer will submit a job. However, this is difficult to achieve in practice due to the high degree of uncertainty of individual user behavior. Training an individual model for each customer, as is done in [40], is not feasible due to scalability constraints.

Difficulty of modeling performance-cost trade-offs [C2]. In tandem with the proactive provisioning mechanism, one needs to model the performance observed by customers (for example the wait time for accessing a cluster and starting a job) and estimate the extraneous COGS from the operator's point of view. Any mismanagement of resources, including over-provisioning or under-provisioning, will result in either significant financial losses or an unsatisfactory customer experience.

Compliance for service level agreements [C3]. The inherent unpredictability and opacity of machine learning is always the biggest concern. While most cloud operators are mandatory to meet specific service level agreements, a robust and consistent algorithm is critical. However, the algorithm may fail to converge in certain corner cases, and using a black-box approach like ML introduces significant challenges in debugging and error triage.

Requirements of full automation [C4] and low latency [C5]. For a production-level system, it is necessary to have a fully automated and reliable solution. Additionally, the provisioning system must be able to adapt to a constantly changing environment by taking into account the real-time state of the system. To achieve this, it is necessary to develop and maintain a low-latency monitoring system, as well as simple and efficient algorithms.

Introduction to Intelligent Pooling. To overcome these challenges, we propose Intelligent Pooling, a self-adaptive solution that proactively creates clusters based on monitoring of demand.

We introduce the notion of a Spark "live pool", where a number of clusters are proactively created and pre-configured for various users. [C1]. Whenever a customer requests a cluster, one cluster will be immediately evicted from the pool and made available for use. At the same time, a new cluster is provisioned and added to maintain a constant cluster number, a process referred to as "re-hydration."

We introduce a self-tuning system to dynamically learn the optimal pool size based on demand, considering the cost-performance trade-offs [C2, C5]. One of the biggest concerns of the live pool mechanism is the COGS. At the scale of Microsoft Fabric [32], we expect simple threshold-based provisioning to quickly exceed 10,000 CPU cores, resulting in tens of millions of dollars in COGS. To address this challenge, our work proposes dynamically adjusting the size of the pool based on customer demand. We propose an efficient linear programming (LP) solution to model the two factors (performance and cost) based on the Pareto frontier to determine the optimal pool size. A self-adaptive system is proposed that automatically balances the trade-off between the cost of maintaining idle clusters and the potential for long wait times for customers.

We introduce an efficient, and robust hybrid time-series forecasting algorithm of the future demand at the aggregate level with high accuracy and robustness [C1, C3, C5]. The models predict future demand, measured by the cluster request rate, based on historical demand and the latest observation of the cluster creation request rate. The predictions then serve as inputs to the optimization model. With an end-to-end run time (training, inferencing, and optimizing) reduced to *mere seconds*, we ensure that the recommendations are always up-to-date by retraining the model with high frequency (e.g., < 5min). To improve the robustness of the model prediction [C3], we developed a new policy to smooth the input data, which significantly reduces performance regressions.

We have implemented a real-time monitoring system that provides continuous inputs to constantly learn the optimal provisioning policy [C4, C5]. The telemetry data collected is then fed into our optimization algorithm in real-time. The same dashboard is also used to evaluate the performance of the system.

The end-to-end solution is lightweight and implemented in C# as an integral part of the core Spark infrastructure [C5]. The recommendation engine can be executed in a single invocation of the pipeline, with fast and accurate recommendation generation and persistence in configuration files in mere seconds, ready to use for the pooling service. We integrate the modules with the new Fabric service, the new Microsoft data analytics offering that supports Spark [32], deployed in all production Azure regions in Nov, 2023.

Contribution. In sum, our contributions are:

- A simple linear programming formulation for solving the optimal pool size that captures the trade-off between improving performance and reducing COGS;
- An efficient hybrid ML algorithm combining deep learning and traditional ML for predicting future demand and optimal pool size with extremely low latency;
- A robust strategy to account for demand uncertainty to ensure high service level;
- Deployed in production, we achieved millions of dollars in annual COGS compared to preexisting pooling.

The remaining sections are organized as follows: Section 2 provides background and motivates the problem. Section 3 presents the overall design principles and architecture. Sections 4 and 5 describe the optimization and ML modules, respectively. Section 7 goes over how we evaluated the algorithm using production data. Section 8 discusses related work, and Section 9 concludes the paper.

2 BACKGROUND

One common issue related to Spark cluster initialization is a long wait time [14]. Complexity in the underlying infrastructure introduces a wide range of potential slowdowns that are difficult to detect, diagnose and mitigate. Examples of this complexity include hardware heterogeneity, unreliable network communication, and inter-node service coordination which is compounded by the strict multi-tenancy requirements in the cloud. Efforts are undergoing to reduce the *tail latency* of cluster initialization time, including making hedged requests [30] and using tied requests [15, 30]. However, these approaches are not able to completely address the issue.

In May 2023, Microsoft announced Fabric, a new data analytics that offers both data engineering and data science experiences, operating as a multi-tenant managed Spark service, with a security boundary scoped to individual users. On this platform, the typical underlying process to initiate a Spark session consists of 60-120 seconds for the cluster creation and 30-40 seconds for the session creation [2, 48]. When Generic Job Service requests a new cluster, This prolonged process is primarily influenced by four main processes: VM configuration, allocation, and boosting time; stitching VMs to form Spark clusters; configuring libraries; and creating a Spark session—each taking 30-60 seconds, contributing to the extended wait time.

To minimize the latency experienced by the end-user, we propose to proactively provision Spark clusters. In this work, we propose to create a shared pool of actively-running clusters, which we call a *live pool*. We categorize live pools into two buckets: *session pools* and *cluster pools*, i.e. interactive and batch mode respectively. Both consist of pooled clusters; the difference is that session pools also have an actively-running *Spark session* in each cluster, which we call a *pooled session*. Session pools are useful for notebook scenarios, when a pre-created session can be used to run a notebook instantaneously. Pooled clusters, by contrast, are useful for running batch jobs with pre-defined job definitions (e.g., a json file that describes a .jar file location, Spark configurations, etc.) and Spark sessions that require ad hoc customization. For the rest of the paper, we discuss the methodology with respect to cluster pools, though the same can be applied to session pools. Similar to the concept of inventory management in retailing [16], we maintain a constant number of resources in a given pool, and upon receiving a client request, a pre-provisioned resource can be used instantly. To maintain the target number of resources in the pool, we send a new request, referred to as a re-hydration request, to Generic Job Service to add a new cluster or session back to the pool whenever a pooled resource is consumed or fails (due to exceeding a pre-defined lifespan or unexpected system failures). For Fabric, two pools per region (one for session and one for cluster) with a fixed cluster size, e.g., 3-median nodes, are created.

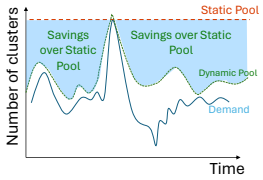


Figure 1: Benefits of Intelligent Pooling

The general idea of Intelligent Pooling is to dynamically determine the optimal number of resources in a pool and scale the pool up or down as needed in real-time. A larger pool can lead to wasted COGS in a low-demand scenario. On the other hand, a smaller pool has a higher likelihood of being drained out in high-demand scenarios, where numerous customers need clusters or sessions at the same time and the system does not have enough time to sufficiently replenish the pool. The client request in this situation must go through the original protracted startup process (referred to as "on-demand"). With dynamic pooling, by adjusting the pool size according to (predicted) demand, we can achieve potentially significant savings over the static pool (see Figure 1). Given that ML algorithms are in general never perfect, with margins in prediction errors, the optimal provisioning strategy remains a challenge.

3 INTELLIGENT POOLING OVERVIEW

In this section, we discuss the overall architecture of Intelligent Pooling. Intelligent Pooling consists of two main modules (see Figure 2):

- The **Sample Average Approximation (SAA) Optimizer** formulates a simple linear programming problem to optimize the pool size based on input demand, which can either be historic or predicted by the ML Predictor. The optimized results are then saved as configuration files in Cosmos DB [35] (Section 4).
- The **ML Predictor** makes real-time time-series predictions by constantly fetching historic observations from the Kusto store [31]. Using predicted demand as opposed to historic demand helps to react more accurately to real-time changes in the system, though it can potentially lead to longer model-training latencies (Section 5).

Intelligent Pooling leverages the existing infrastructure used by live pools for its own execution and deployment. Specifically,

- **Generic Job Service** is responsible for orchestrating Spark batch jobs and interactive sessions; providing APIs to perform CRUD operations on Spark jobs; and managing and processing Spark job-related metadata.
- **Cluster Service** is responsible for requesting virtual machines (VMs) from Azure and "stitching" them to form Spark

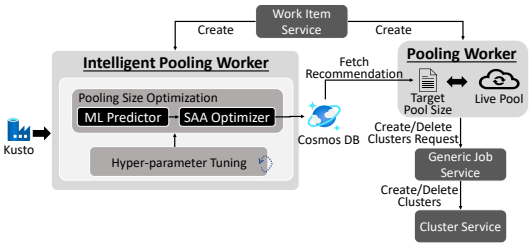


Figure 2: Architecture

clusters; providing APIs to perform CRUD operations on Spark clusters; and managing and processing Spark cluster-related metadata.

- **Work Item Service** is a background service that supports various workloads, including but not limited to Spark cluster and session pooling, Spark job submissions, and the Intelligent Pooling infrastructure. It is responsible for monitoring available *work items* that represent these workloads and spinning up worker processes that execute the workloads.
- The **Intelligent Pooling Worker** is responsible for periodically running the ML pipeline that includes the aforementioned SAA Optimizer and ML Predictor and persisting the recommendation files in Cosmos DB [35].
- The **Pooling Worker** is responsible for maintaining a target pool size by invoking Generic Job Service to create and delete resources and, if applicable, fetching the latest pool size recommendation file emitted by an Intelligent Pooling Worker.

The hyper-parameter tuning module is developed to constantly fine-tune the hyper-parameters for the optimization algorithm to avoid over- or under-allocating resources by setting the tuning knobs to minimize the COGS while satisfying the SLA. It can be executed at a lower frequency while the ML pipeline runs at a higher frequency such that it captures the rapid change of the environment and adapts faster to the demand. More details on the hyper-parameter tuning process can be seen in Section 6.

4 SAMPLE AVERAGE APPROXIMATION OPTIMIZER

In this section, we formulate the live pool mechanism as a queuing system and propose linear programming to reach optimality.

4.1 The Live Pool Mechanism

We illustrate the live pool mechanism through a graph plotting a set of cumulative values (see Figure 3). Specifically,

- $D(t)$: the cumulative number of clusters requested by customers (demand);
- $N(t)$: the target pool size as a function of time that we want to maintain;
- $A(t)$: the cumulative number of cluster re-hydration requests made to add a cluster to the pool in order to keep it at the target pool size of $N(t)$;
- τ: the cluster initialization time, i.e., the time lag before a cluster can be ready for use after the creation request is sent;
- $A'(t)$: the cumulative number of clusters ready for use.

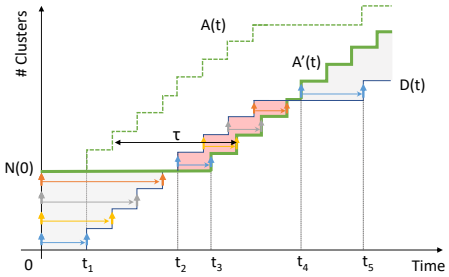


Figure 3: Cumulative cluster creation number $D(t)$, cumulative re-hydration requests $A(t)$, number of clusters ready $A'(t)$ at time t, and pool size at time 0, $N(0)$, the total wait time (red area) for obtaining a running cluster and idle time (grey area) for clusters in the pool.

For instance, at time $t = 0$, a pool is created with $N(0) = 4$ clusters, and whenever a user request for a new cluster is received, (corresponding to an increase in $D(t)$, e.g., at t_1), a cluster will be emitted from the pool to be used by the customer. At the same time, the pooling worker will initiate a re-hydration request to Generic Job Service to add a new cluster back to the pool (corresponding to an increase in $A(t)$). As a result, the curve of $A(t)$ can be seen as a simple "shift-up" of the curve of $D(t)$, and the gap between them equals the target pool size $N(t)$. Similarly, because cluster creation takes time, the cumulative number of clusters that are ready to use, $A'(t)$, is a "shift-right" of the number of requests made, $A(t)$, by the cluster creation latency, τ:

$$A(t) = D(t) + N(t), \quad \forall t \tag{1}$$

$$A'(t) = A(t - \tau), \quad \forall t \geq \tau \tag{2}$$

$$A'(t) = N(0), \quad \forall t < \tau. \tag{3}$$

Assuming a first-come-first-serve rule, the clusters in the pool will be acquired by customers based on the user request arrival time. For instance, in Figure 3, the first four clusters in the pool will be used by the first four requests that are received (blue, yellow, grey and orange respectively). The fifth created cluster is triggered by the arrival of the blue request at t_1, and is not ready until t_3. It will be used by the fifth request received at t_2.

4.2 Optimal Pool Size

There are two factors to optimize for:

- The total idle time pooled clusters are alive and unused by customers; and
- The total wait time for customers (when the pool is drained out and the customer must wait for the full cluster startup duration).

In Figure 3 we show the one-to-one mapping between a created cluster and the request that will use the corresponding cluster based on a first-come-first-serve (FCFS) rule. We observe that whenever $A'(t) > D(t)$, idle time occurs, and whenever $A'(t) < D(t)$, wait time occurs. For example, at time $t = 0$, four clusters have been created, and they will be used by the first four requests that come in (blue, yellow, grey and orange). For those clusters, their aggregate idle time equals the area of the gap between the curves of $A'(t)$ and $D(t)$, highlighted in grey. However, the fifth cluster creation request on curve $A(t)$ (marked in blue) was not ready until t_3 while

the fifth customer request on curve $D(t)$ occurs at t_2. Therefore, for the fifth request, the customer has to wait for $t_3 - t_2$, and the wait time was highlighted in red (similar for the sixth, seventh and eighth requests in yellow, grey, and orange respectively)[1].

In sum, the total wait time of customers is the red area where $A'(t) < D(t)$, and the total idle time of clusters in the pool is the grey area where $A'(t) > D(t)$. With this, we can estimate the optimal pool size where both the total wait time and the idle time are minimized. Specifically, we can leverage linear programming with minimization as the objective to calculate the areas:

$$\Delta^+(t) \geq A'(t) - D(t), \quad \forall t \tag{4}$$

$$\Delta^+(t) \geq 0 \quad , \forall t \tag{5}$$

$$\Delta^-(t) \geq D(t) - A'(t), \quad \forall t \tag{6}$$

$$\Delta^-(t) \geq 0, \quad \forall t. \tag{7}$$

If the objective function involves minimizing $\Delta^+(t)$ and $\Delta^-(t)$, one can prove that in the optimal solution, if $A'(t) \geq D(t)$, $\Delta^+(t) = A'(t) - D(t)$ and $\Delta^-(t) = 0$. If $A'(t) \leq D(t)$, $\Delta^-(t) = D(t) - A'(t)$ and $\Delta^+(t) = 0$. And the sum of Δ^+ and Δ^- calculates the total area of grey (idle time) and red (wait time) respectively. And the constraints are all linear.

Sample Average Approximation (SAA) Optimizer. Based on the above discussion, we can formulate the optimization program with the objective to minimize the total cost (considered as a weighted sum of the wait time and idle time). We use the sample average approximation (SAA) method [26] based on the input demand data to minimize the expected total cost over the whole observed period. Denote α and β the hyper-parameters representing the penalty of having long idle time versus wait time, a larger α will result in an optimal solution trying to minimize the idle time more than wait time, and vice versa. By changing the hyper-parameter values, one can achieve the full Pareto curve [18, 46] of the trade-off between idle time and wait time. The detailed formulation is as follows:

$$\min \alpha \cdot \sum_t \Delta^+(t) + \beta \cdot \sum_t \Delta^-(t) \tag{8}$$

$$\text{s.t.,} \qquad (1) - (7).$$

With this minimization formulation, for the optimal solution, $\Delta^+(t)$ equals the number of idle clusters at time t, and Δ^- the queued demand. All the constraints are linear. For the maximum number of requests made, one can add a constraint with the maximum number of requests per time interval, MAX NEW REQUEST:

$$N(t) - N(t-1) \leq \text{MAX NEW REQUEST} \quad \forall t \geq 1. \tag{9}$$

For the analyzed system, we also added the following constraints:

$$\text{MIN POOL SIZE} \leq N(t) \leq \text{MAX POOL SIZE} \quad \forall t \geq 1, \tag{10}$$

$$N(t) = N(\lfloor t/\text{STABLENESS} \rceil * \text{STABLENESS}) \quad \forall t \geq 1. \tag{11}$$

where Constraint (10) sets the minimum and maximum pool size (i.e., MIN POOL SIZE and MAX POOL SIZE), and Constraints (11) ensures that the pool size is stable for Δt = STABLENESS intervals.

[1]Note that currently, when a pool is drained out, "on-demand" cluster creation requests will be sent to accommodate the cluster requests, and their wait time becomes τ. The clusters being ready at t_3 and onwards will be served to later requests. In this case, the FCFS rule is violated as the order of using the clusters is modified. In this work, we still assume FCFS for simplicity as an approximation of this mechanism.

In production, the MIN POOL SIZE and MAX POOL SIZE are set according to regional capacity.

For a simplified intelligent pooling policy, one can also add constraints to ensure that the pool size for the same day of week or time of day is the same as for a more static controlling policy. Note that all the constraints are still linear and can be solved by commercial solvers with low latency (in a few seconds).

5 ML PREDICTOR

In this section, we detail the prediction model and pipeline that we chose. Specifically, Section 5.1 discusses initial model exploration and Section 5.2 discusses the limitations. Section 5.3 proposes a hybrid model that combines a traditional machine learning algorithm with deep models. Section 5.4 proposes an alternative way to combine the SSA optimizer with the ML predictor.

5.1 Cost-Performance Trade-offs

We evaluate the following four different models, each representing a different category/approach: (1) Singular Spectrum Analysis (SSA) [19] implemented by ML.NET [1]; (2) Inception Time [24]; (3) TST [50]; and (4) mWDN [45]. These models are chosen since each represents a different category—SSA is a traditional ML model, TST is a transformer-based deep learning approach, mWDN is wavelet decomposition-based approach and Inception Time is a 1D convolution model.

To train the models, we use an 80-20 train-test split. Specifically, for the deep learning models, the training set is further split into a 90-10 train-validation set. For DNN models, we use the validation set to ensure we do not overfit to the training set and to trigger an early stop. To directly embed the estimation of the wait-idle time trade-off into the training process, we use a modified loss function similar to the estimation of Δ^+ and Δ^- as in Equations 4-7, which is a proxy of the true wait and idle time to capture the trade-offs between cost and performance. We define the loss function $\mathcal{L}$ as:

$$\mathcal{L} = \alpha' \cdot \delta^+ + (1 - \alpha') \cdot \delta^- \tag{12}$$

$$\delta = y - \hat{y} \tag{13}$$

$$\delta^+ = \delta : \delta > 0 \tag{14}$$

$$\delta^- = -\delta : \delta < 0 \tag{15}$$

where y is the ground truth time series, $\hat{y}$ is the predicted output and α' is the hyper-parameter that controls the relative importance of idle time and wait time during optimization and training. We observe that the modified loss function in Equation (12) allows for the models to perform better at the extremes of the wait time and idle time constraints. It also allows the model to estimate the demand at a higher/lower percentile, tailored to our business needs.

5.2 Limitations

From our experiments, we found that the rate at which we update the model has a big impact on the idle time (i.e., cost savings). Thus, it was critical to identify and deploy models that were fast to update. We found that deep models (mWDN, TST and Inception Time) were significantly slower to train compared to SSA (see Figure 6 on data scaling) though they offer flexibility with customized loss functions, allowing tailoring to different performance and cost preferences for

demand prediction—whether conservative or aggressive. Lacking such flexibility, SSA failed to achieve sufficiently low wait times (refer to Figure 5). To address these two limitations, we propose a new *hybrid model* which combines the best of both.

5.3 Hybrid Model: SSA+

To address the above-mentioned limitation, in this paper, we propose the hybrid model to achieve low training latency and relatively good trade-off performance. We address the issues with SSA and the deep models by combining certain parts of each. The reason SSA fails to achieve low wait times is because there is no way to specify and control how much the predicted request rate must overshoot the ground truth (see Section 7.3). If the predicted usage/pool size is larger than ground truth, this will result in a larger pool size, lowering the average wait time. With deep models, the overshoot is controlled using the loss function defined by Equation (12). However, the issue with deep models is that the models are too computation-intensive for the task at hand and need lots of data and computational resources to train the over-parameterized model. Thus, the proposed hybrid model consists of an SSA forecaster followed by a shallow two-layer neural net (≈ 30 parameters, with ReLU activation for non-linearity) which acts as an error predictor. This error predictor can be trained using the loss from Equation (12) to learn the overshoot or undershoot needed to achieve the target wait time. The improved trade-off performance of the hybrid model can be seen in Figure 5.

5.4 End-to-end Recommendation Engines

While modern ML approaches can achieve high performance, they are not immune to errors. Thus, any processing that is performed post-ML prediction must not introduce more errors by propagation. This is similar to the game of telephone, where the ML model makes a prediction and subsequent processing introduces enough uncertainty to make the prediction useless. With the goal of minimizing the potential for errors, we explored two end-to-end recommendation pipelines:

- **2-step** pipeline where the ML model is trained on the input cluster request rate data. The ML model predicts cluster request data, which is then fed to the SAA optimizer which outputs the predicted optimal pool size.
- **End-to-End (E2E)** pipeline where we apply the SAA optimizer on the historic data, providing a ground truth optimal pool size for the past. The historic optimal pool size is then used to train the ML model, which predicts the optimal pool size for the future.

From our experiments, we found 2-step pipelines have a better Pareto curve when targeting low wait times (Section 7).

6 AUTO TUNING PARAMETERS

Constantly adapting the hyper-parameters to meet business needs is a challenge and adds to the cost of maintaining the service. Particularly, in production we want to achieve our service-level agreement on the performance (wait time) while minimizing the idle versus wait time penalty. Thus, we propose using a self-adaptive hyper-parameter tuning mechanism to close the feedback loop. We constantly monitor the system behavior (pool hit and pool misses) and adjust for the parameters accordingly such that we can always maintain the optimal balance as desired.

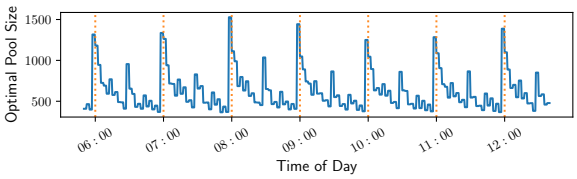


Figure 4: Pool size increases advance demand.

To achieve this, we eliminate the β hyper-parameter from Equation 8 and rewrite the objective function as:

$$\min \alpha' \cdot \sum_t \Delta^+(t) + (1-\alpha') \cdot \sum_t \Delta^-(t), \tag{16}$$

where $0 \leq \alpha' \leq 1$. It is easy to show that the formulation is equivalent to Equation (8).

Thus we have only one hyper-parameter to tune, which reduces our search space. We can model the relation between the business requirement (customer wait time) t_{wait} and the hyper-parameter α'. We approximate the relation $\alpha' = f(t_{\text{wait}})$ to be piece-wise linear. With this approximation, we try to fit the best line based on the previous 10 data points and update the value iteratively.

7 EXPERIMENT

In this section, we present the results of applying SAA on historic data (Section 7.1), compare the performance of various ML models (Section 7.2) and compare the performance of the two end-to-end pipelines integrated with the optimizer by displaying their wait-idle time trade-off curves (Section 7.3). We discuss the model efficiency in Section 7.4 and deployment results in Section 7.5.

In all assessments, we consolidate the input time series data into 30-second intervals, where the data signifies the number of cluster requests per interval. We maintain a constant pool size for 5 minutes to prevent abrupt changes in size recommendations. Various combinations of penalty values, such as α, β, and α', were examined to achieve diverse trade-offs between perf and cost. The evaluation was conducted on a node with 6 vCores and 64 GB RAM.

7.1 Sample Average Approximation (SAA) optimizer

We extracted production Azure Synapse data in the East US region from July 01 to July 15 in 2022 with hundreds of thousands of cluster requests. We estimated the optimal pool size by time of day and type of day (weekday versus weekend) using historic data. Several interesting findings emerged:

The pool size is correlated with the number of requests. In Figure 4, we find that the pool size increases 5 minutes before the start of every hour that is, 5:55, 6:55, 7:55, etc. This is due to the fact that many jobs are scheduled at 6AM, 7AM, etc. The optimization proactively prepares for this surge by increasing the pool size to cope with this demand.

There is a trade-off between longer wait time and longer idle time. As discussed previously, a larger pool size generally results in longer idle times and a decrease in the likelihood of the pool being drained out. We can tune the value of the cost penalty in the objective function to tune the pool size and obtain a Pareto curve.

High frequency in which the pool size is updated can improve both COGS and performance. We observed that by decreasing the *STABLENESS* as in Equation (11) the Pareto curve shifts towards the lower left, indicating better perf-cost trade-offs. However, in production, we are not able to update the pool size too frequently and potentially decreasing the pool size will also result in cancellation of re-hydration requests.

7.2 ML Model Comparison

Table 1: Performance comparison using a 2-step pipeline.

Region	Node Size	MAE ↓				
		SSA+	SSA[19]	mWDN[45]	TST[50]	IncpT[24]
West US 2	Small	***9.54***	14.13	10.28	10.86	10.00
East US 2	Small	7.78	8.02	7.23	***7.13***	7.64
West US 2	Medium	5.54	4.82	***3.77***	4.28	4.01
East US 2	Medium	1.42	1.60	***1.26***	1.33	1.33
West US 2	Large	3.28	3.67	***3.21***	3.23	3.40
East US 2	Large	1.89	2.42	***1.79***	1.90	1.98
Average		4.91	5.78	***4.59***	4.79	4.73

To identify the best ML model, we compared the performance of Singular Spectrum Analysis (SSA) [19], Inception Time [24], TST [50] and mWDN [45] models with our proposed hybrid model (SSA+) using 14 day's historic data. And each algorithm will predict 1200 steps ahead. The algorithm-specific hyperparameters are fined-tuned to achieve the best accuracy and latency on the training set. Table 1 presents the performance of each of the models on various datasets (each row is a different dataset from one region). We configured the hyperparameters as follows: window size 150, 15 epochs, batch size 768, horizon 1200, learning rate 0.001, and series length 1800. We present the performance in terms of two metrics: Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) between the prediction and the ground truth. From Table 1 we clearly see that mWDN [45] outperforms other models on average. The mWDN demonstrates strong performance, particularly in regions with larger and stable patterns, such as West US2. The TST model, requiring a longer period of input data due to their increased parameters, performs well, however, its latency is the longest among all (see Figure 6). In contrast, the IncepT model with a 1D-convolution layer may not be sufficiently powerful to capture the diverse patterns.

7.3 End-to-end Pipeline

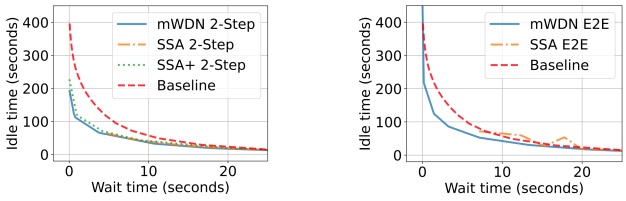
In this section, we evaluate the Pareto curve for the 2-step approach (predict future demand, and then apply SAA optimizer to the prediction) and the E2E pipeline (apply SAA optimizer to historic demand for historic optimal pool size, and then use ML to forecast).

Figure 5 shows the trade-off between idle time and wait time for various ML models using both the 2-step and E2E approach. "SSA+" denotes the results for the hybrid model. We use a no-intelligence model as baseline. A no-intelligence model's output is defined as:

$$\hat{y} = \gamma \cdot \max(y_{\text{train}}) \tag{17}$$

where $max(y_{\text{train}})$ is peak request rate of the training data, γ is a fixed constant and $\hat{y}$ is the predicted cluster/session request rate.

Figure 5 reveals some interesting results: (1) The difference in COGS (idle time) of baseline and other ML models increases as we target lower wait times. However, this is up to a certain point, after which the difference in COGS reduces as the wait time approaches zero; (2) SSA-based models fail to achieve very low wait times (e.g., <5) for both the 2-step (in Figure 5a) and the E2E approach (Figure 5b). However, for the mWDN model, by tuning the custom loss function (Equation 12), we can further increase the penalty for long wait times; (3) While we observe that an end-to-end pipeline has better prediction performance to predict optimal pool size directly, the trade-off curve suggests 2-step performs better (see Figure 5a compared to Figure 5b).



(a) 2-step approach **(b) E2E approach**

Figure 5: Wait time vs idle time for mWDN, SSA and baseline models trained with production data.

Table 2: Estimated annual cost savings with intelligent pooling for US (7 regions).

Target Wait (Hit rate)	Static Pool	SSA+	Savings SSA+	Savings mWDN
0.5s (~99.9%)	$>20M	$>15M	$>5M	$>5M
1s (~99%)	$>15M	$>10M	$>5M	$>5M
5s (~95%)	$>5M	$>5M	$>2M	$>2M

Targeting 99% pool hit rate (the percentage of cluster requests experiencing 0 wait time), the system achieves up to 43% reduction in idle time compared to static pooling. The COGS savings given different SLAs for customer wait time are shown in Table 2. With Intelligent Pooling, compared to the simple heuristics of static pooling, we are able to achieve large monetary savings.

7.4 Data Scaling

We evaluated the training time of the ML models using different data sizes (see Figure 6). The hybrid model built on top of SSA has a slightly increased training time compared to SSA, but it is still extremely fast (200x faster) compared to the pure deep learning models (mWDN, TST or InceptionTime). In production, we deployed the SSA+ model and trained it in an infinite loop, as it reaches similar performance as mWDN with significantly reduced latency. Compared with most ML pipelines where models are trained at a lower frequency and preserved into model files to be fetched at the inference time, such design significantly reduces the complexity of maintenance and development. The latency of the optimization module remains unchanged as the input data size equals the length of the predicted time frame (which is set to 1 hour for the production pipeline).

7.5 Production Deployment

We deployed Intelligent Pooling across all production regions within Fabric in Nov 2023, with results showing great promise in significantly reducing COGS (>60% for some production regions) and no impact on other workers co-hosted. This pipeline is scheduled to run in a continuous loop and generate pool size recommendations

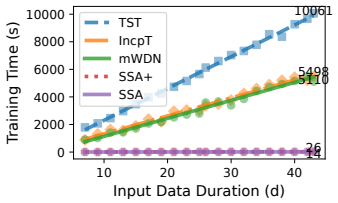


Figure 6: Training time vs input data size.

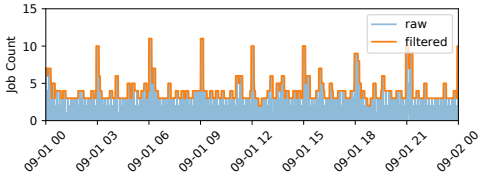


Figure 7: Raw versus filtered demand.

for the next hour. Nevertheless, in one specific region, the ML predictions exhibit lower accuracy due to sporadic spikes occurring approximately every 3 hours (albeit not precisely timed), posing challenges for precise demand and spike arrival forecasting. To bolster ML robustness and enhance pool hit rates, we implemented the following strategies: (1) prior to ML training, we applied a max filter to smooth the time-series data based on a SMOOTHING FACTOR (SF), resulting in "fatter" spikes (see Figure 7) by replacing the demand D in Equation (1) with $\bar{D}$ as in Equation (18); (2) for the linear optimizer, we extended the STABILITY period to 10 minutes, forcing Intelligent Pooling to recommend advanced pool size adjustments to accommodate spikes; (3) we applied a max filter (similar to Equation (18)) for the recommended pool size based using $SF = \tau$ to ensure that the pool size was increased for a sufficiently long period of time for spiky demand. Utilizing all the strategies, Intelligent Pooling effectively addresses demand spikes, and even if they occur irregularly, the pool size is always sufficient. These enhancements for model robustness were deployed in production, leading to a further increase in COGS savings from 18% to 64% by significantly reducing the pool size when demand is close to zero while maintaining the hit rate to 100%.

$$\bar{D}(t) = \begin{cases} \max\{D(t - \lfloor \mathrm{SF}/2 \rceil), ..., D(t + \lfloor \mathrm{SF}/2 \rceil))\}, & \forall t \geq \lfloor \mathrm{SF}/2 \rceil \\ \max\{D(0), ..., D(t + \lfloor \mathrm{SF}/2 \rceil)\}, & \forall t < \lfloor \mathrm{SF}/2 \rceil \end{cases} \tag{18}$$

Note that in production we: (1) run the the pipeline in a continuous loop to update the pool size with high frequency, that requires low the end-to-end latency of the algorithm (see Section 7); (2) set up a guardrail to validate the ML model's prediction accuracy before running the downstream optimization; (3) set up an alerting system for pipeline failures as well as a monitoring system such that we can be informed and investigate any potential issues. And we track the Intelligent Pooling status (succeeded, failed), metrics of average idle time, recommended pool size, demand request rate, pool miss/hit count/percentage, COGS saved, hydration status such as number of clusters in provisioning/ready/targeted in real-time. This comprehensive monitoring system is an essential part of the Intelligent Pooling.

7.6 Fault Tolerance

Potential system failures arise from two main sources: (1) inference pipeline failure and (2) other pooling worker issues. In terms of algorithm failure, we ensure fault tolerance by generating recommendations for the next hour for each run, while executing the algorithm at more frequent intervals, e.g., 30 min. This safeguards against a single run failure, as the system retains the previous recommendation output, albeit slightly outdated. Additionally, in the case of consecutive system failures, leading to missing recommendations, the inferencing reverts to default configurable values. A health check is maintained that tracks a pooling worker's assignment status (locked or available). This involves periodic checks to confirm consistent assignment to healthy workers, overseen by the Arbitrator service. Each pooling task is leased to a worker and undergoes refreshment upon lease expiration with periodic health checks, ensuring regular health checks for all workers involved, with prompt replacement of unhealthy ones.

8 RELATED WORK

Performance modeling has been used to support proactive autoscaling of resources to meet specific service-level agreements (SLAs) or Quality of Service (QoS) requirements [11–13, 25, 42, 43]. With the advent of ML algorithms, research has been done in workload-forecasting methods, specifically time-series analysis, to facilitate resource management. [23] proposes a workload classifier based on statistics such as maximum, coefficient of variation, etc., and enumerates over a set of time-series forecasting algorithms, selecting the most appropriate one. [41] claims that 77% of the database usage on Azure SQL Database Serverless is predictable and leverages ML predictions to proactively pause and resume databases. To automate the scheduling of backups for PostgreSQL and MySQL servers, Seagull [40] tests different ML models (including NimbusML [37], GluonTS [7], and Prophet [17]) to forecast user load for each specific server. The system identifies low-load windows with 99% accuracy using a simple heuristic, and this solution has been deployed across all Azure regions. However, there has been limited research focusing on proactive resource provisioning to reduce the cluster initialization latency for Spark applications.

CloudNet [47] introduced the dynamic pooling of VMs by migrating networks, disks, and memory. However, in Fabric, VM pooling is unpractical due to security and authentication issues as they need to be transfered across pooled network. In this work, we focus on cluster/session pooling.

9 CONCLUSION

In this work, we propose "pooling" (proactively provisioning) Spark clusters and sessions to eliminate the initial startup latencies. Intelligent Pooling involves predicting customer demand through an innovative hybrid machine learning model and dynamically adjusting the pool size to meet customer demand while reducing extraneous COGS using linear programming with low latency and high robustness. Evaluated using Fabric data, the system achieves up to a 43% reduction in idle time compared to static pooling while maintaining a 99% pool hit rate. Deployed in production, the system is running robustly for several regions with significant COGS savings by capturing the spiky demand patterns with dynamic pool sizes. Future works involve the operation of multiple pools with different configurations (cluster size, etc.).

REFERENCES

[1] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. 2019. Machine learning at Microsoft with ML. NET. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2448–2458.

[2] A Aleksiyants, O Borisenko, D Turdakov, A Sher, and S Kuznetsov. 2015. Implementing Apache Spark jobs execution and Apache Spark cluster creation for Openstack Sahara. 27, 5 (2015), 35–48.

[3] Amazon. 2022. *Amazon AWS.* Retrieved July 2, 2022 from https://aws.amazon.com

[4] Apache. 2022. *Apache Flink.* Retrieved July 2, 2022 from https://flink.apache.org

[5] Apache. 2022. *Apache Storm.* Retrieved July 2, 2022 from https://storm.apache.org

[6] AWS. 2022. *Azure Synapse.* Retrieved July 2, 2022 from https://aws.amazon.com/emr/features/spark/

[7] Amazon AWS. 2022. *GluonTS-Probabilistic Time Series Modeling in Python.* Retrieved July 2, 2022 from https://ts.gluon.ai/stable/

[8] Francisco J Baldan, Sergio Ramirez-Gallego, Christoph Bergmeir, Francisco Herrera, and Jose M Benitez. 2016. A forecasting methodology for workload forecasting in cloud systems. *IEEE Transactions on Cloud Computing* 6, 4 (2016), 929–941.

[9] Luciano Baresi and Giovanni Quattrocchi. 2018. Towards vertically scalable spark applications. In *European Conference on Parallel Processing*. Springer, 106–118.

[10] JV Bibal Benifa and D Dejey. 2019. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications* 24, 4 (2019), 1348–1363.

[11] Anshuman Biswas, Shikharesh Majumdar, Biswajit Nandy, and Ali El-Haraki. 2014. Automatic resource provisioning: a machine learning based proactive approach. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE, 168–173.

[12] Raouia Bouabdallah, Soufiene Lajmi, and Khaled Ghedira. 2016. Use of reactive and proactive elasticity to adjust resources provisioning in the cloud provider. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 1155–1162.

[13] Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-effective resource provisioning for Spark workloads. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2477–2480.

[14] DataBricks. 2022. *Best practices: pools for Databricks.* Retrieved July 2, 2022 from https://docs.databricks.com/clusters/instance-pools/pool-best-practices.html

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[16] Nicole DeHoratius, Adam J Mersereau, and Linus Schrage. 2008. Retail inventory management when records are inaccurate. *Manufacturing & Service Operations Management* 10, 2 (2008), 257–277.

[17] Facebook. 2022. *Prophet: Forecasting at scale.* Retrieved July 2, 2022 from https://facebook.github.io/prophet/

[18] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[19] Nina Golyandina and Anton Korobeynikov. 2014. Basic singular spectrum analysis and forecasting with R. *Computational Statistics & Data Analysis* 71 (2014), 934–954.

[20] Google. 2022. *Google Cloud Platform.* Retrieved July 2, 2022 from https://cloud.google.com

[21] Google. 2022. *Serverless Spark.* Retrieved July 2, 2022 from https://cloud.google.com/dataproc-serverless/docs

[22] Google. 2022. *Spark through Vertex AI.* Retrieved July 2, 2022 from https://cloud.google.com/vertex-ai-workbench

[23] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. 2013. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 187–198.

[24] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F Schmidt, Jonathan Weber, Geoffrey I Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. 2020. Inceptiontime: Finding alexnet for time series classification. *Data Mining and Knowledge Discovery* 34, 6 (2020), 1936–1962.

[25] Reihaneh Khorsand, Mostafa Ghobaei-Arani, and Mohammadreza Ramezanpour. 2019. A self-learning fuzzy approach for proactive resource provisioning in cloud environment. *Software: Practice and Experience* 49, 11 (2019), 1618–1642.

[26] Anton J Kleywegt, Alexander Shapiro, and Tito Homem-de Mello. 2002. The sample average approximation method for stochastic discrete optimization. *SIAM Journal on Optimization* 12, 2 (2002), 479–502.

[27] Yi-Hsuan Lee, Kuo-Chan Huang, Cheng-Hsien Wu, Yen-Hsuan Kuo, and Kuan-Chou Lai. 2017. A Framework for Proactive Resource Provisioning in IaaS Clouds. *Applied Sciences* 7, 8 (2017), 777.

[28] Jinzhao Liu, Yaoxue Zhang, Yuezhi Zhou, Di Zhang, and Hao Liu. 2014. Aggressive resource provisioning for ensuring QoS in virtualized environments. *IEEE transactions on cloud computing* 3, 2 (2014), 119–131.

[29] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12, 4 (2014), 559–592.

[30] Qinghua Lu, Liming Zhu, Xiwei Xu, Len Bass, Shanshan Li, Weishan Zhang, and Ning Wang. 2014. Mechanisms and architectures for tail-tolerant system operations in cloud. In *6th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 14)*.

[31] Microsoft. 2022. *Azure Data Explorer - Kusto.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/

[32] Microsoft. 2022. *Azure Fabric.* Retrieved July 24, 2023 from https://learn.microsoft.com/en-us/fabric/data-engineering/spark-compute

[33] Microsoft. 2022. *Azure HDInsight.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-overview

[34] Microsoft. 2022. *Azure Synapse.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-overview

[35] Microsoft. 2022. *Introduction to Cosmos DB.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

[36] Microsoft. 2022. *Microsoft Azure.* Retrieved July 2, 2022 from https://azure.microsoft.com

[37] Microsoft. 2022. *NimbusMLbu.* Retrieved July 2, 2022 from https://docs.microsoft.com/en-us/nimbusml/overview

[38] Peter P Nghiem and Silvia M Figueira. 2016. Towards efficient resource provisioning in MapReduce. *J. Parallel and Distrib. Comput.* 95 (2016), 29–41.

[39] Yoori Oh, Jieun Choi, Eunjung Song, Moonji Kim, and Yoonhee Kim. 2016. A SLA-based Spark cluster scaling method in cloud environment. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 1–4.

[40] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. In *PVLDB*. VLDB Endowment, 154–162.

[41] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. *PVLDB* 15, 6 (2022), 1279–1287.

[42] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, and Biswanath Mukherjee. 2018. Auto-scaling VNFs using machine learning to improve QoS and reduce cost. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[43] Jianfei Ruan, Qinghua Zheng, and Bo Dong. 2015. Optimal resource provisioning approach based on cost modeling for spark applications in public clouds. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*. 1–4.

[44] Kundjanasith Thonglek, Kohei Ichikawa, Chatchawal Sangkeettrakarn, and Apivadee Piyatumrong. 2021. Auto-scaling system in apache spark cluster using model-based deep reinforcement learning. In *Heuristics for Optimization and Learning*. Springer, 347–360.

[45] Jingyuan Wang, Ze Wang, Jianfeng Li, and Junjie Wu. 2018. Multilevel wavelet decomposition network for interpretable time series analysis. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2437–2446.

[46] Wikipedia. 2023. Pareto front. https://en.wikipedia.org/wiki/Pareto_front.

[47] Timothy Wood, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. 2011. CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines. *ACM Sigplan Notices* 46, 7 (2011), 121–132.

[48] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[49] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[50] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. 2021. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2114–2124.

[51] Xiaoxi Zhang, Chuan Wu, Zongpeng Li, and Francis CM Lau. 2017. Proactive VNF provisioning with multi-timescale cloud resources: Fusing online learning and online optimization. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[52] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens,

et al. 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *Proceedings of the 2021 International Conference on Management of Data*. 2667–2680.