KEA: Tuning an Exabyte-Scale Data Infrastructure

Yiwen Zhu[1] Subru Krishnan[1] Konstantinos Karanasos[1] Isha Tarte[1] Conor Power[1] Abhishek Modi[1]
Manoj Kumar[1] Deli Zhang[1] Kartheek Muthyala[1] Nick Jurgens[1] Sarvesh Sakalanaga[2*]
Sudhir Darbha[1] Minu Iyer[1] Ankita Agarwal[1] Carlo Curino[1]
[1]Microsoft, firstname.lastname@microsoft.com
[2]Salesforce, firstname.lastname@salesforce.com

ABSTRACT

Microsoft's internal big-data infrastructure is one of the largest in the world—with over 300k machines running billions of tasks from over 0.6M daily jobs. Operating this infrastructure is a costly and complex endeavor, and efficiency is paramount. In fact, for over 15 years, a dedicated engineering team has tuned almost every aspect of this infrastructure, achieving state-of-the-art efficiency (>60% average CPU utilization across all clusters). Despite rich telemetry and strong expertise, faced with evolving hardware/software/workloads this manual tuning approach had reached its limit—we had plateaued.

In this paper, we present KEA, a multi-year effort to automate our tuning processes to be fully data/model-driven. KEA leverages a mix of domain knowledge and principled data science to capture the essence of our cluster dynamic behavior in a set of machine learning (ML) models based on collected system data. These models power automated optimization procedures for parameter tuning, and inform our leadership in critical decisions around engineering and capacity management (such as hardware and data center design, software investments, etc.). We combine "observational" tuning (i.e., using models to predict system behavior without direct experimentation) with judicious use of "flighting" (i.e., conservative testing in production). This allows us to support a broad range of applications that we discuss in this paper.

KEA continuously tunes our cluster configurations and is on track to save Microsoft tens of millions of dollars per year. At the best of our knowledge, this paper is the first to discuss research challenges and practical learnings that emerge when tuning an exabyte-scale data infrastructure.

ACM Reference Format:
Yiwen Zhu[1] Subru Krishnan[1] Konstantinos Karanasos[1] Isha Tarte[1] Conor Power[1] Abhishek Modi[1], Manoj Kumar[1] Deli Zhang[1] Kartheek Muthyala[1] Nick Jurgens[1] Sarvesh Sakalanaga[2], and Sudhir Darbha[1] Minu Iyer[1] Ankita Agarwal[1] Carlo Curino[1] . 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457569

[*]The work was done while the author was at Microsoft.

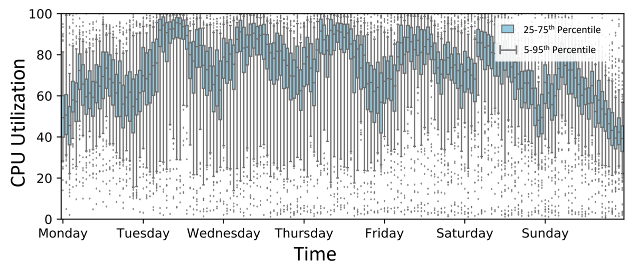


Figure 1: CPU utilization for a typical week

1 INTRODUCTION

Big-data infrastructures have empowered users with unprecedented ability to store and process large amounts of data, paving the road for revolutions in the areas of web search, analytics, and artificial intelligence [5, 6, 15, 19, 43, 44, 49, 50, 54].

At Microsoft, we operate *Cosmos*, one of the largest big-data infrastructures in the world, with over 300k machines serving billions of tasks daily. Tremendous engineering and research effort has been devoted over the last 15 years to improve Cosmos' scalability, efficiency, security, and reliability [10, 11, 13–16, 18, 21, 30, 31, 34, 42, 43, 56, 58, 60]. Given the significance and complexity of this infrastructure, almost every aspect of it has been carefully tuned by a dedicated team to optimize its efficiency, improving its performance and reducing operational costs. As a result, the team has reached industry-leading levels of utilization (e.g., >60% average CPU utilization, as shown in Figure 1). But despite the domain expertise and rich telemetry, our improvements started to plateau as we reached the limits of what we could achieve via manual tuning.

To overcome this plateau, we turned to very large-scale data science as a tool to further optimize our operations. Although such "ML-for-Systems" approaches have been widely employed in multiple settings lately [17], unsurprisingly, they are impractical at our scale. Among them, the more practically viable ones have focused on small-scale settings (e.g., to tune a single DBMS instance), being based on the fundamental assumption of repeatedly changing system tunables and running experiments to measure the resulting performance [2, 4, 20, 33, 41]. This *"experimental tuning"* approach can be unrealistic in our setting: deployments must roll out progressively across tens of thousands of machines, noisy workloads require long windows of observation (>weeks), and running a "bad" configuration could have devastating effects on some of the most business-critical operations at Microsoft. As we will show, we use experimental tuning only as a last resort.

The system and methodology we developed, named *KEA*, is the result of a multi-year effort aimed at evolving our processes to be fully automated and data/model-driven. KEA systematically combines domain expertise and principled data science to capture the complex dynamic behavior of a big-data cluster as a collection of "descriptive" and "predictive" ML models, i.e., models that can describe and predict the system's behavior. These models power automated optimization procedures and are used both for direct parameter tuning, but also to guide our leadership in tactical engineering and capacity decisions (such as hardware and data center design, software investments, etc.). KEA employs rich "observational" tuning (i.e., without requiring to modify the system) with judicious use of "flighting" (i.e., conservative testing in production). This allows us to support a broad range of tuning applications.

We suspect that a handful of other companies are operating solutions akin to what we describe here, but to the best of our knowledge this is the first paper describing the data science-driven tuning of an exabyte-scale analytics infrastructure. In presenting this work, we focus on the peculiar challenges that arise at scale, and present a methodology and tools to tackle them, as well as lessons learned in deploying these ideas in production settings.

In summary, we make the following contributions:

- We define a methodology to cope with the system complexity and create compact, sound, and explainable models of a cloud infrastructure based on a set of tractable metrics, which can explain "why" the optimal configuration is chosen.
- We present the end-to-end architecture of KEA and provide details for the three types of tuning that KEA enables: (i) *observational* tuning, which employs models for picking the right parameters, avoiding costly rounds of experiments; (ii) *hypothetical* tuning, an ML-assisted methodology for future planning; and (iii) *experimental* tuning, our fall-back approach that judiciously performs experiments when it is not possible to predict the system behavior otherwise.
- Deployed in production, KEA continuously tunes our Cosmos clusters and is on track to save the company tens of millions of dollars yearly. We share our experience of running KEA in production at such a massive scale.

The remainder of this paper is organized as follows. Section 2 provides background and motivates our cluster tuning problem. Section 3 presents the rationale behind our system conceptualization and the derivation of relevant metrics. Section 4 describes the overall architecture and summarizes the different tuning modes. Sections 5, 6, and 7 describe each tuning approach in more detail. Section 8 discusses related work, then we conclude.

2 BACKGROUND & MOTIVATION

At Microsoft, we operate *Cosmos*, a data analytics platform for the internal analytics needs across the company [18]. Cosmos is one of the largest data infrastructures worldwide with more than 300k machines spread across several data centers. Cluster resources are spread across tens of thousands of users that submit more than half a million analytics jobs daily. Table 1 summarizes several scalability metrics of this massive infrastructure.

Table 1: Cosmos statistics [18, 45]

Description	Size
Number of jobs per day	>600k
Number of tasks per day	>4B
Number of users	>10k
Total number of machines	>300k
Number of machines per cluster	>45k
Total Hardware Capital Expenditure in US Dollar	>$1B

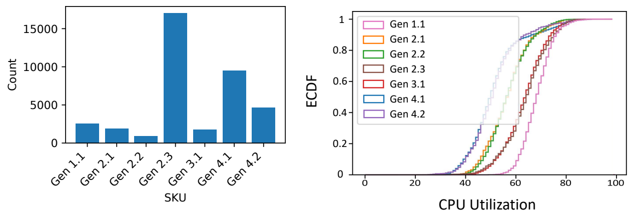


Figure 2: Machine count (left) and utilization level (right) for different hardware generations for one of the clusters

The vast majority of the submitted jobs are written in Scope [59], a SQL-like dialect (with heavy use of C# and Python UDFs). Scope jobs are translated to a DAG of operators that are spread for execution across several machines. Each job is comprised of up to hundreds of thousands of tasks, i.e., individual processes each executed in one *container*. A YARN-based [51] resource manager is used for scheduling tasks and sharing cluster resources across jobs [18]. More than four billion tasks per day get scheduled across Cosmos.

To make matters even more complicated, along with scale comes heterogeneity. After a decade of operation, Cosmos involves more than 20 hardware generations of machines (with varying CPU cores, RAM, HDD/SSD) from various manufacturers and software configurations (e.g., mapping of drives to SSDs/HDDs). Hereafter, we use the term stock keeping unit, or *SKU*, to refer to a hardware generation and *SC* to a software configuration. Each cluster consists of 6 to 9 SKUs for working machines (compute nodes), which are the main focus of this work. Figure 2 shows the distribution of machines per SKU for one of the clusters.

Operating such a complex infrastructure requires tuning hundreds of parameters across user applications and the underlying infrastructure. In this paper, we focus on infrastructure-level configurations, and in particular on cluster-wide configurations, as they are very impactful and traditionally harder to tune. As we show, at this scale we cannot simply perform tuning based on a large round of experiments or trivial A/B testing—a more sophisticated approach is required.

At the same time, the starting point for our optimization journey is a non-trivial one to beat: years of domain experts fine-tuning Cosmos to maximize efficiency and reach an industry-leading CPU utilization of over 60% (average across all our clusters). However, this manual effort is time-consuming and error-prone, and it requires continued adjustments as workloads shift and new hardware and software is deployed. Having no rigorous way to evaluate the current parameter choice and suggest more promising values leads to significant missed opportunities in terms of operational cost and performance.

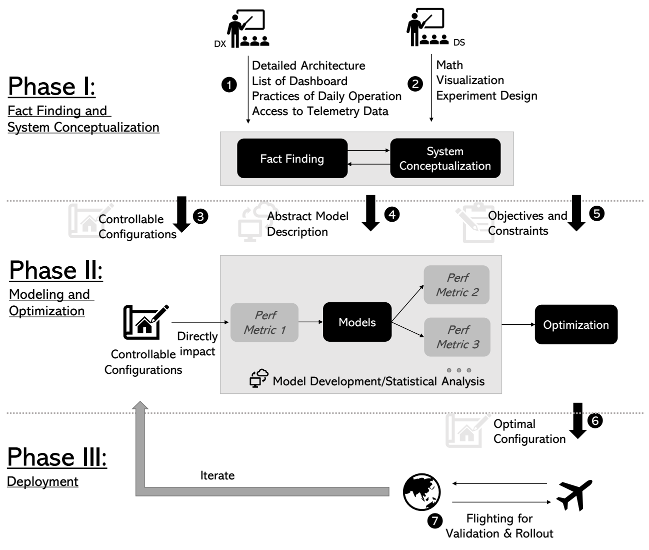


Figure 3: End-to-end methodology of KEA

Tuning cluster-wide parameters is a scary and time-consuming affair, as changes must be rolled-out progressively across the fleet, mistakes are costly as performance may crater, and long-term workload seasonalities impose long observation windows. This leads to very slow progress as operators need to proceed very cautiously and validate for long periods of time. A symptom of this becomes directly evident on the right-hand side of Figure 2: while overall utilization is good, older-generation machines (i.e., the ones we have been tweaking for longer) are substantially more utilized, as we have slowly learned how hard we can push the hardware before negatively impacting stability or user-perceived performance. As our workloads keep evolving and we continuously roll out new (and thus never-tested-before) SKUs, this slow time-to-optimal tuning puts a cap on our maximum efficiency (as a fraction of workloads/hardware is always sub-optimally used).

Realizing the limitations of the current manual approach in parameter tuning, three years ago we embarked in a journey to tune our clusters in a principled data-driven way, and the KEA project was born.

3 END-TO-END METHODOLOGY

With KEA we introduce both a methodology and a software architecture to support tuning of large scale systems. In this section, we focus on the methodological aspects, which are designed to guide a data science team to interact with the engineering group owning and operating the system to be tuned. The KEA software architecture is discussed in Section 4.

3.1 Phases of a KEA project

The KEA methodology is structured in three distinct phases, as shown in Figure 3.

Fact Finding and System Conceptualization (Phase I). In this phase, all the stakeholders in a "tuning project" come together to identify the goal and scope of the project. More precisely, the data scientists (DS) engage with the engineering team owning the system, referred hereafter as the domain experts (DX), in a series of interviews and brainstorming sessions aimed at *Fact Finding* and *System Conceptualization. Fact finding* aims at obtaining a thorough architecture description of the system, collecting intuition on the system dynamics, defining the objectives of the project (e.g., improving latency/operational cost), and identifying the controllable configurations. The DS are granted access to telemetry data and dashboards used by DX for daily monitoring of the clusters, and come to learn the various practices of daily operation of the system, such as workflows for debugging and incident resolution (which are typically rich with data-to-insight information). *System conceptualization* aims at simplifying the complex architecture of the analyzed system to a smaller number of blocks with annotations of flows that describe the interaction between them. This is crucial to capture the system complexity in a compact and mathematically tractable way. The DS and DX identify the potential practical constraints that need to be satisfied in the tuning process, such as ensuring latency requirements. Neither DS nor DX can achieve this step alone and a close collaboration is required to leverage the collective expertise and knowledge. We describe the system abstractions we employ during conceptualization in Section 3.2.

In this conceptualization effort, DS leverages data-driven and statistical tools to observe correlations between key performance metrics, and collaborate with DX to understand the causal relationships between components, creating a correct model of the complex system. The combined domain knowledge and numerical validation creates a deeper understanding of the system, and it is in its own right valuable to the system operators (often the resulting visualizations are embraced by the engineering teams, even beside the rest of our automation work). Note that at this stage we have not built ML models yet, just a crisp understanding of how components relate to each other and what we could expect to predict effectively.

The *output* of Phase I includes the list of controllable configurations, the description of possible models to build in Phase II, along with the objectives and constraints that need to be considered.

Modeling and Optimization (Phase II). In this phase, we use the *system conceptualization* from Phase I and dive into a more quantitative analysis with actual ML, optimization, statistical, and econometric methods to model the behavior of the system using mathematical formulations based on appropriate assumptions. The result is a model of the entire system we can use to answer interesting "what if" questions. The optimizer uses this model to pick the optimal configurations. This phase mostly involves DS for the analysis and model development, while results are interpreted and validated by DX.

The *output* of this phase includes the optimal configuration setting for the target parameter(s) that maximizes the chosen objective function, subject to the constraints we defined in Phase I.

Deployment (Phase III). In this phase, we conduct "flighting" of the proposed configuration for validating the models and assumptions. The flighting service is an internal tool that allows cluster operators to deploy new software versions and new configuration parameters to a live production environment. The infrastructure gives us flexibility to deploy to partial or complete clusters, and to carefully measure the impact on user-observable and cluster-internal internal metrics (e.g., CPU utilization per node). Iteratively,

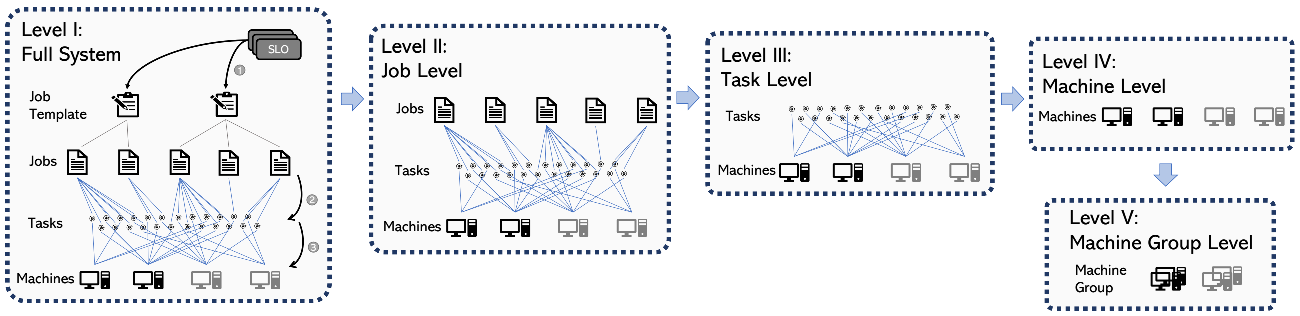


Figure 4: Simplifying Cosmos: from job-level SLO compliance to machine-centric metrics

DS fine-tunes the models and works closely with DX to monitor the cluster behavior and prepare for the final configuration roll-out.

3.2 System Abstractions

In this section, we discuss the abstractions we introduce to allow us to model the highly complex infrastructure we want to tune in a tractable manner. As discussed above, these abstractions play a key role in the *system conceptualization* of Phase I.

Modeling all individual components of a large-scale infrastructure is intimidating as it involves a multitude of components, while detailed simulation or modeling of each aspect of the system is often intractable. For example, when applying KEA to the Cosmos infrastructure, we must consider the behavior of and the interactions between the scheduler, compiler, query optimizer, job manager, and task executor. Predicting the runtime of hundreds of thousands of jobs by modeling the task-level interference across billions of tasks spanning tens of thousands of machines would require models with a massive number of parameters and unacceptable training times. Modeling abstractions and simplifications are crucial. In this paper, we demonstrate that the right abstraction process can lead to a handful of small models, which in turn are capable of capturing enough of a system behavior to inform the selection of an optimal configuration. As we show in Sections 5–7, this process outperforms the existing manual tuning approach.

To make this modeling more concrete, we now discuss the introduced abstractions we made to model Cosmos. Our end goal is to improve Cosmos efficiency, i.e., pick new configurations that will allow us to run more jobs without affecting existing job runtime performance.

Full system abstraction (Level I). On the left of Figure 4, we depict the full complexity of Cosmos. Each machine runs a mix of tasks that belong to many job instances of various job "templates".[1] Modeling the system by including all task-level interferences and intra-job dependencies is prohibitive. We thus describe several simplification steps (from left to right in the figure) leading to increasingly more manageable levels of abstraction, as shown in Figure 4 and described below. Note that our abstractions take advantage of the fact that Cosmos uses a monolithic resource manager that assigns directly tasks to machines, and the majority of jobs are SCOPE jobs (see Section 2). In the case of multi-framework two-level schedulers, such as Mesos [27] or Borg [52], similar abstractions could be made for different groups of machines (e.g., based on the jobs that typically land on each group).

Job-level abstraction (Level II). The initial intuition on how to simplify this picture comes from the DX who informed the team that "most jobs in Cosmos have implicit runtime SLOs". This means that users expect a predictable behavior but don't have explicitly defined SLOs. In practice this means that recent runtime behaviors of a job induce an implicit SLO on the next execution of the same job template [16, 31]. This allows us to quantify a vague customer expectation in the following constraint: $\forall i,\ \text{runtime}(\text{job}_i, \text{conf}_{\text{new}}) \leq \text{runtime}(\text{job}_i, \text{conf}_{\text{old}})$, which states that the runtime of job_i should be no worse with the new configuration (conf_{new}) than with the old configuration (conf_{old}).[2] These constraints are statistical in nature due to naturally occurring variances based on job specification, input data, and transient cluster conditions. The DS quantifies the natural variance and leverages statistical testing techniques to verify whether the constraint holds during our experimentation.

Task-level abstraction (Level III). The next intuition from the DX is that "job runtimes are dominated by slow tasks in the critical path" (i.e., a set of slowest tasks in each stage of the job execution) [45]. The important implication is that ensuring that the performance of slow tasks is not impaired by a configuration change is a sufficient condition to satisfy the job-level performance constraint introduced above. The DS validates this by confirming that slow tasks are indeed on the critical path. The validation is more rich and complex, but the high level idea can be peeked in Figure 5: tasks landing on slower machines (that are older and busier) are disproportionately more likely to be slower and therefore be part of the job's critical path.

Machine-level abstraction (Level IV). This abstraction is based on the observation made by the DS that the scheduler randomizes tasks uniformly across nodes, as shown in the left of Figure 6. This allows us to focus directly on optimizing machine behavior, ignoring task-to-task interactions. Again this is held true in a statistical sense. Fortunately, big-data systems are by design very resilient to individual failures or misbehaving nodes, thus a statistical improvement is all we care for.

Machine group-level abstraction (Level V). The final observation from the DS is that tasks are also spread uniformly across SKUs

[1]A job template represents a recurring job. It is created by removing the specific data inputs from the corresponding job script [18].

[2]The notation is simplified for the sake of presentation; more precisely we should compare the runtime of an upcoming job with an amortized historical runtime of instances from the same job template.

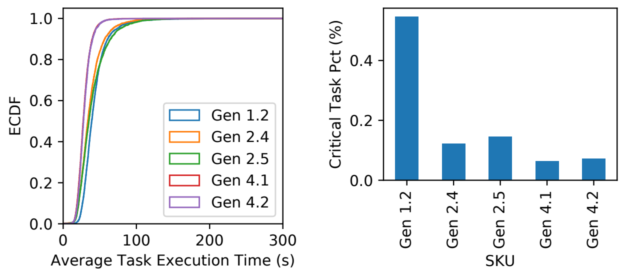


Figure 5: Task execution time distribution across different SKUs indicates that tasks executed on slower machines are more likely to be on the critical path of a job execution.

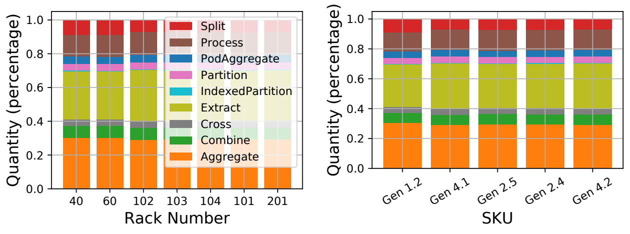


Figure 6: Task type distributions across racks (left) and SKUs (right) are very similar, indicating that, at the aggregate level, machines are fairly receiving a similar combination of workloads that is representative to the whole cluster.

(see the right of Figure 6). This allows us to model at the machine group-level, ignoring specific machine-level effects.

Level V is sufficiently abstract to allow us to model the whole infrastructure in a tractable yet comprehensive manner. In particular, we can now only focus on machine group-level metrics—examples of such metrics are shown in Table 2. Such metrics can be directly affected by machine group-level parameters and the resulting impact can be mathematically quantified.

Table 2: Examples of performance metrics at the machine group level

Name	Description	Affected System Metrics
Total Data Read	Total bytes read per hour per machine	Throughput rate
Number of Tasks	Total number of tasks finished per hour per machine	Throughput rate
Bytes per Second	Ratio of sum of the total data read and total execution time per machine	Throughput rate
Bytes per CPU Time	Ratio of sum of the total data read and total CPU time per machine	CPU processing rate
CPU Utilization	Time-average CPU utilization per hour in percentage	Utilization level
Average Running Containers	Time-average running containers per hour	Utilization level

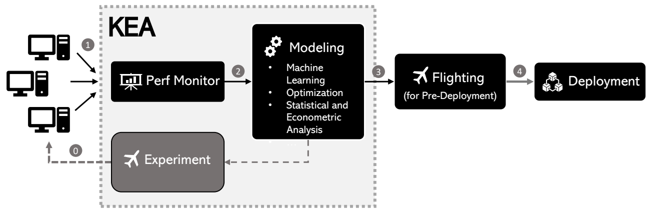
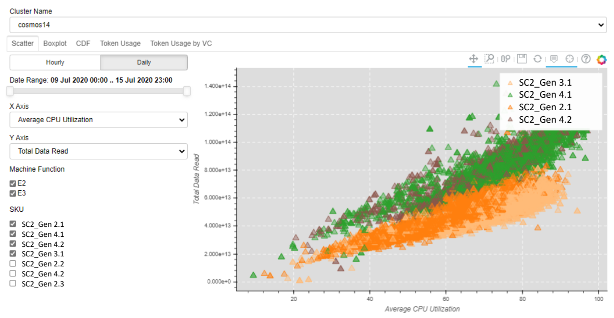


Figure 7: Overall architecture of KEA. The Experiment module and the flighting module share the same infrastructure to deploy new configurations. The Experiment module is only used for Experimental Tuning.


Cluster Name: cosmos14

Date Range: 09 Jul 2020 00:00 .. 15 Jul 2020 23:00

X Axis: Average CPU Utilization

Y Axis: Total Data Read

Machine Function: ☑ E2 ☑ E3

SKU: ☑ SC2_Gen 2.1 ☑ SC2_Gen 4.1 ☑ SC2_Gen 4.2 ☑ SC2_Gen 3.1 ☐ SC2_Gen 2.2 ☐ SC2_Gen 4.2 ☐ SC2_Gen 2.3

Figure 8: Scatter view of the performance monitor. We observe a linear trend between the total throughput (Total Data Read in the y-axis) and the machine CPU utilization level (x-axis). The distribution varies across machine groups.

4 SYSTEM OVERVIEW

In this section, we present KEA's architecture (Section 4.1) and the three tuning approaches supported by KEA (Section 4.2).

4.1 Architecture

KEA's architecture is shown in Figure 7. It consists of the following three main modules.

The **Performance Monitor** joins data from various Cosmos sources and calculates the performance metrics of interest, providing a fundamental building block for all the analysis. To collect and prepare these metrics at a daily basis, an end-to-end data orchestration pipeline is developed and deployed in production on Cosmos itself. Figure 8 shows an instance of our visualization dashboard using the processed data. The *scatter view* depicts the data in a disaggregated way with each point corresponding to one observation for a machine during one hour.

The **Modeling Module** proposes the optimal configurations as described in Phase II of Figure 3 (see Section 3). Depending on the application, different methods can be used, such as ML, optimization, statistical analysis, or econometric models.

The **Experiment Module** is used to perform experiments on a group of machines to gather performance data for new configurations, when this is required to model the system.

The **Flighting Tool** facilitates the deployment of configuration changes to any machine in the production cluster as a safety check before performing the full cluster deployment using the **Deployment Module**. In the flighting tool, users can specify the machine names and the starting/ending time of each flighting and create new builds to deploy to the selected machines.

As the modeling and experiment modules vary across different applications, they are discussed in more detail for each application in Sections 5–7.

4.2 Tuning Approaches

We now describe the different tuning approaches enabled by KEA.

Our goal is to avoid, to the degree possible, "experimental tuning", i.e., the modeling of the system based on rounds of experiments that are costly and often prohibitive, given the scale and constraints of our infrastructure. To this end, we make the crucial observation that *in many scenarios we can model the effect of new configurations to the system's behavior based solely on data collected from existing cluster operating points*. For example, to determine the maximum number of tasks to run per machine, we can simply look at the performance of the system when different numbers of tasks happened to run on those machines (given that the number of tasks per machine changes over time based on the cluster load). Note that in cases of future cluster planning (e.g., how much memory to use for future machines), experimental tuning is not even an option—relying on existing cluster operation points is the only feasible way. However, some applications require additional exploration to actually deploy the new configuration and evaluate the observed performance—those usually involve configurations that the system has not seen before. To this end, we introduce the following three tuning approaches:

Observational Tuning aims at improving the operation efficiency of an existing fleet of machines. It can be applied to configuration scenarios for which the telemetry data collected from past cluster operation is sufficient to predict the performance of a new configuration, thereby completely bypassing experiments. Based on the modeling results, we only need to conduct flighting as a pre-deployment safety check to validate the model prediction. The KEA modules (see Figure 7) required for this type of tuning are: (1) performance monitor, (2) modeling, (3) flighting, and (4) deployment.

Hypothetical Tuning aims to support planning for future cluster machines. It also uses modeling based on existing cluster operational points. In this case, there is no flighting nor actual deployment, since the output is configuration of machines that do not yet exist. Applications fueled by this tuning approach focus on forecasting for future scenarios, such as choosing memory and CPU for future machines. The required KEA modules here are: (1) performance monitor and (2) modeling.

Experimental Tuning is used when the current telemetry data is not sufficient for predicting the performance of a new configuration. Used as our last resort (due to its increased cost), it strategically performs experimental deployments of new configurations to gather new cluster operational points. The data collected during experiments are then used to model the system. This approach uses all KEA modules in Figure 7.

By employing these tuning approaches, KEA can tackle a wide variety of configuration scenarios that we encounter in Cosmos. In this paper, we focus on four scenarios that we have already addressed using KEA in production. Optimizing these scenarios, which are summarized in Table 3 and described below, has led to over tens of millions of dollars yearly savings for the company.

Table 3: KEA applications

Application	Tuning Approach	Tuning Parameter
YARN Configuration	Observational Tuning	Maximum running containers for each SC-SKU combination
SKU Design	Hypothetical Tuning	Amount of RAM, SSD per machine
Power Capping	Experimental Tuning	% below current provision level
Software Configurations	Experimental Tuning	Binary choice between SC1 and SC2

(1) YARN Configuration Tuning. The goal is to re-balance the workload across different groups of machines and achieve better cluster throughput by tuning YARN configuration parameters. Specifically, we focus on the `max_num_running_containers` parameter that can be specified for each machine group corresponding to a software-hardware combination. This parameter limits the maximum number of containers that can be executed simultaneously on a machine. `max_*/min_*` is a commonly seen parameter in various configuration settings, and KEA can be employed to tune many such parameters.

(2) Machine Configuration Design. The goal is to determine the hardware configuration (amount of RAM and SSD size) of future machines in a way that does not leave any of the resources idle and without impacting throughput either. The most cost-efficient configuration tailored to current customer workloads is desired.

(3) Power Capping. The bottleneck for fitting more machines in a rack or in a data center is provisioned power, not space. With power capping we focus on determining new power limits. Originally, the Cosmos machines were provisioned with a conservatively high power consumption limit—based on years of observation, this power limit turns out not to be cost-effective. By capping the power utilization and provisioning less power per machine, we are able to increase the number of machines per rack and thus per data center. This way the fixed costs of racks and data centers, which account for a large portion of a data center cost [7], are amortized across more machines.

(4) Selecting Software Configurations (SCs). Machines are purchased with different hardware (i.e. SKU) and deployed with different software versions. In Cosmos we use two main software configurations, denoted here as SC1 and SC2, corresponding to different mappings of logical drives to physical media. We use KEA to determine which of the two options is preferable.

Application (1) can be tackled based solely on existing observational data that allow us to build models and predict performance for different maximum number of running containers per SC-SKU combination. Hence, it is a good example of Observational Tuning. Application (2) belongs to Hypothetical Tuning, as it focuses on future planning of the system. Finally, for applications (3) and (4) we use Experimental Tuning, because without actual experiments with new configurations, it is very difficult to predict the impact of new power capping limits or new software configurations.

5 OBSERVATIONAL TUNING

In this section, we develop an Observational Tuning approach to tune the configurations by building the predictive models (see Phase II in Figure 3) to avoid the need of rounds of cluster-wide experiments, which are required by the black-box approaches, such as Bayesian optimization (BO) and reinforcement learning (RL) [23, 53, 55]. We find that cluster-wide experiments are impractical in large-scale production environments like Cosmos because of how slowly and carefully changes to production need to be made. By properly modeling the dynamics of the system, we can predict the potential performance changes with different configurations.

5.1 Modeling

The Observational Tuning approach consists of two modules: (1) the *What-if Engine* to predict the performance metrics given different configurations and (2) the *Optimizer* to select the optimal solution.

In the What-if Engine, we aim to predict the resulting performance given a new set of configurations. On the path to tackle this problem, we make two crucial observations:

- *The change of a particular set of configuration parameters usually affects one (or a few) sets of metrics directly, and the impact is easy to measure (see Phase II in Figure 3).* For instance, by changing the configuration for the maximum number of running containers, the metrics directly impacted are the actual running containers of a machine and its distribution. It is clear that reducing the max will shift its distribution towards the lower end.
- *We can capture the dynamics between different sets of parameters and better understand how the change in one set of metrics affects the others using ML models, i.e. the chain-effect.* In the observational data, due to the natural variance of the system operation, we have a full-spectrum of the ranges of the performance metrics (see Figure 8 where we have observations for machines running with various levels of CPU utilization). Based on this variation, we can develop models to mimic the dynamics between different sets of metrics and map one metric to another.

The above two observations are important building blocks to capture the relationship between changes in configurations to the changes in the objective functions (or constraints) that we hope to optimize upon. In summary, we use the following three step process: (1) based on the set of parameters that we hope to tune, we identify the set(s) of metrics that will be directly impacted; (2) we build ML models to understand how this set of metrics affects the others, especially the ones that relate to our objective functions/constraints; and (3) based on the resulting formulation, we perform optimization to pick the optimal configuration (see Phase II in Figure 3).

For the development of the ML models, we conclude that the dynamics between the different sets of metrics remain the same, even with different configuration settings. Those system aspects reflect the mechanics of the infrastructure and the characteristics of the workloads. For instance, in Figure 8, even with different levels of CPU utilization or the workload levels, the *relationship* between the resulting throughput and the CPU utilization level can be expressed with the *same* formulation for each group of machines with a particular software-hardware combination. This relationship will not be affected by the external configuration, such as YARN configuration settings. In this research, we leverage those system fundamentals to predict the resulting performance under new configurations to avoid the need for experiments. Those are the calibrated models we want to develop in Phase II as in Figure 3.

Based on the observational data, sets of ML models can be built, such as $g_k(\cdot)$, $h_k(\cdot)$ and $f_k(\cdot)$, for each SC (software configuration)-SKU (hardware) combination k, to capture the relationship between the different sets of metrics, such as (1) the number of running containers versus the CPU utilization level, (2) the CPU utilization level versus the number of tasks finished per hour, and (3) the CPU utilization level versus the task latency respectively:

$$x_k = g_k(m_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{1}$$

$$x'_k = g_k(m'_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{2}$$

$$l_k = h_k(x_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{3}$$

$$l'_k = h_k(x'_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{4}$$

$$w_k = f_k(x_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{5}$$

$$w'_k = f_k(x'_k)\ \forall k = 1, 2, 3, \cdots, K, \tag{6}$$

where,

k:	the index for the SC-SKU combination, $k = 1, 2, 3, \cdots, K$.
m_k:	the number of running containers (simultaneously) per machine with SC-SKU combination k.
m'_k:	the original average number of running containers per machine with SC-SKU combination k.
x_k and x'_k:	the CPU utilization level for machines with SC-SKU combination k, given the number of running containers m_k and m'_k respectively.
l_k and l'_k:	the number of tasks finished per hour on a machine with SC-SKU combination k, given the CPU utilization level x_k and x'_k respectively.
w_k and w'_k:	the average task latency for machines with SC-SKU combination k, given CPU utilization level x_k and x'_k respectively.

Many other ML models can be developed to involve a larger set of metrics of interest, such as the resource utilization of SSD, RAM (used in Section 6), or network bandwidth, for different applications. In general, we use regression models as the predictors, such as linear regression (LR), support vector machines (SVM), or deep neural nets (DNN). Linear models are more explainable, which is critical for domain experts.

For a complex system, given the fact that there are only a handful number of machine groups with different software-hardware combinations, based on the needs of different projects, a small number of models per group are sufficient to mimic the full dynamics of the system, which is tractable and easy to maintain.

In the Optimizer, different objective functions and constraints can be easily plugged-in with respect to the goals of the applications, and the corresponding ML models with respect to the directly impacted performance metrics and the ones related to the objective functions/constraints can be used.

5.2 Yarn Configuration Tuning

For the application of tuning the maximum running containers in YARN, after Phase I, we decide to maximize the total number of running containers (relates to the sell-able capacity for the cluster) subject to the same overall average task latency at the cluster level as in the current operation. Therefore, the directly impacted performance metric is the number of running containers on the machine. And we maintain the same level of task latency (cluster-wide average) as the constraint. Thus, the optimization problem can be formulated with a closed-form objective function as a linear programming (LP) problem:

$$\max_{m_k, k=1,2,3,\cdots,K} \sum_{k=1,2,3,\cdots,K} m_k n_k, \tag{7}$$

$$\text{s.t. } \bar{W} \leq \bar{W}', \tag{8}$$

$$\bar{W} = \frac{\sum_k w_k l_k n_k}{\sum_k l_k n_k}, \tag{9}$$

$$\bar{W}' = \frac{\sum_k w'_k l'_k n_k}{\sum_k l'_k n_k}, \tag{10}$$

$$(1) - (6).$$

where,

n_k:	the number of machines in the cluster for machine function-SKU combination k.
$\bar{W}$ and $\bar{W}'$:	the overall average task latency for the full cluster, given CPU utilization level x_k and x'_k respectively, calculated as the weighted average of task latency running on different groups of machines.

The optimal solution of the optimization, $m_k^* \ \forall k = 1, 2, 3, \cdots, K$, can be obtained by commercial solvers, indicating the optimal workload distribution across different groups of machines. Based on the changes of the workload distribution, we can modify the configuration for the maximum running containers accordingly, increasing or decreasing it for different SC-SKU combinations. In the production system, we are also conservative about the changes in configuration. Therefore, extra constraints can be added to limit the range of the decision variables.

In the current system, we observe that the tasks running on slower machines are more likely to be the stragglers that slow down a job. The re-balancing of the workloads suggested by the model reduces the workload skew and shifts traffic from slower machines to faster machines to improve the overall efficiency. With the increased utilization level on faster machines, we expect mild performance degradation for their tasks. However, those are less likely to be on the critical path of a job that directly impacts the job-level latency (see Figure 5). Therefore, even though the constraint for the optimization formulation ensures the same average task-level latency, the tuning improves the performance of the straggler tasks thus improving the job-level latency.

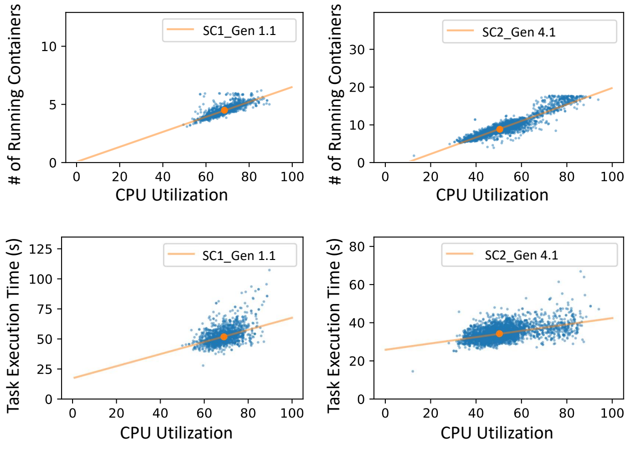


Figure 9: The set of calibrated models for different SCs and SKUs

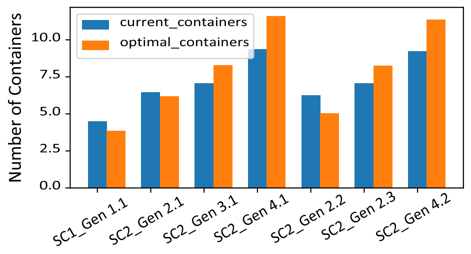


Figure 10: Suggested configuration change

5.2.1 Modeling Results. We present the training and tuning results from one of our production clusters with over 45,000 machines for a seven day period. The full KEA pipeline was set up to collect data on a daily basis and propose a new configuration each day.

We used a Huber Regressor for the prediction of the set of performance metrics in the What-if Engine, which is more robust to outliers compared to the Least Squares Regression [40]. Figure 9 shows the set of calibrated models (see Phase II in Figure 3) to depict the running containers and task execution time in seconds versus CPU utilization level. Each small dot corresponds to an observation aggregated at the daily level for a machine. The line shows the model estimation. The large dot in the center of the figure indicates the median level of the variables across all observations.

Figure 10 shows the optimization results in terms of the suggested shift of current workloads (calculated as the number of containers running per machine). For slower machines, such as `Gen 1.1`, the model suggests to decrease the utilization by reducing the number of running containers, while for faster machines, such as `Gen 4.1`, the model suggests to increase it. We also ran the same optimization model focusing on a higher percentile of CPU utilization level, corresponding to the situation where the whole cluster is running with heavy workloads. The suggested configuration change is the same in terms of the direction for the gradients. We conclude that for the cluster when running with low/median/heavy workload, the same configuration change is desired.

5.2.2 Flighting and Deployment. The flighting module is one of the most important components for KEA that leads to its applicability

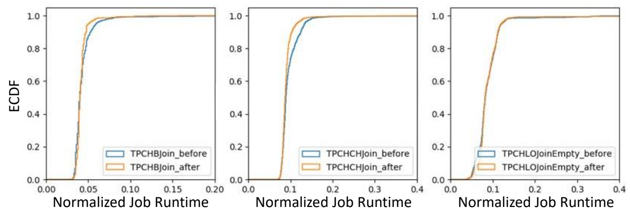


Figure 11: Job runtime distribution for 3 benchmark jobs before and after KEA deployment

to large production systems. Before our first deployment, to establish confidence and demonstrate robustness, we did five rounds of flighting where we validated the possibility of increasing maximum running containers for different SKUs to increase utilization.

The first pilot flighting was on 40 `Gen 1.1` machines to confirm that reducing the `max_num_running_containers` in the YARN configuration file is affecting the real observed maximum number of running containers. The second piloting flighting experiment was on `Gen 4.1` machines to confirm that increasing the `max_num_running_containers` in the YARN configuration is effective and allows the machines to run more workloads. The third piloting experiment was on two sub-clusters [18] of machines (each with around 1700 machines) to validate if the updated configuration changes the workload distribution. The fourth pilot flighting was for three sub-clusters of machines and validated the benefits of tuning, i.e. adding more containers to the sub-cluster with better performance.

The production roll-out process is very conservative where we only modify the configuration by a small margin, i.e. decrease or increase the maximum running containers for each group of machines by one. We extracted the performance data for the periods of one month before and one month after the roll-out. We use *treatment effects* to evaluate the performance changes [28] during the two periods with significant tests. We observe that with the same level of latency (measured by average task latency), the throughput (measured by Total Data Read) is improved by 9%.

We also evaluate a set of benchmark jobs originating from TPC-H and TPC-DS before and after KEA deployment. The average job runtime in this case was improved by 6% (see Figure 11).

For this round of deployment, conservatively, we gain 2% sellable capacity from the cluster (measured by the total number of containers with the same level of latency as before) by only modifying the maximum running containers for each SKU-SC by one. This is equivalent to tens of million dollars cost saving per year. We are expecting 5% more over the existing capacity in the next round where we are allowed to modify the maximum running containers by two. The t-value for the Student’s t-test [48] is 4.45 and 7.13 respectively, indicating that the resulting difference in the data distributions for the before and after periods is significant.

5.3 Discussion

The What-if Engine can be extended to other performance metrics of interest by identifying the most relevant sets of performance metrics and modeling the dynamics between them using predictive models.

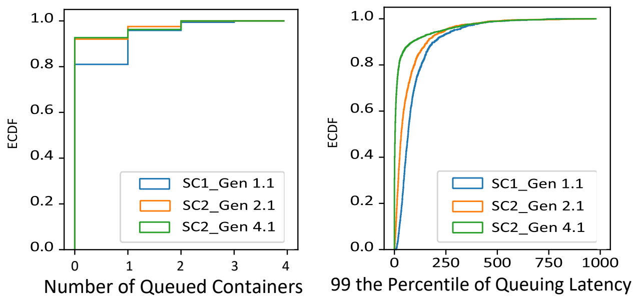


Figure 12: Number of queued containers (left) and 99th percentile of queuing latency in ms (right) for 3 SKUs

In the analyzed system, low priority containers will be queued on each machine when all machines in the cluster reach the maximum number of running containers. We observe that the queuing length and latency vary significantly for machines with different SKUs and SCs (see Figure 12). As faster machines have faster de-queue rate, we can allow more containers to be queued on them. Therefore, similar tuning methodology can be used to learn the relationship between the tuned parameters, i.e. the maximum queuing length, and the objective performance metrics, such as variance of queuing latency, to achieve better queuing distribution.

Based on the ML models proposed in Equations (1)-(6), KEA can also be used to convert any performance improvement into capacity gain (given the same task latency), allowing detailed quantitative evaluation for all engineering changes in monetary values.

6 HYPOTHETICAL TUNING

In this section, we discuss the applications focusing on scenarios for future planning. The same methodology for modeling is applicable as in Section 5.1. The main focus is still to develop the predictive models in the What-If Engine to capture the relationship between the resulting objective functions with respect to a particular configuration change and let the Optimizer pick the optimal solution without deployments or experiments.

6.1 Machine Configuration Design

In this application, we focus on the resource utilization metrics of the machines (as opposed to the throughput or latency) that will influence decisions around what hardware components to purchase in future machines. We have determined the CPU configuration for the next generation. In this work, we build a model to predict how much SSD and RAM we would need to buy given the number of CPU cores.

The different configurations for SSD and RAM directly impact their actual usage and distributions. Specifically, using ML models, we want to project the SSD and RAM usage as a function of the number of CPU cores used based on the observational data. Given a larger number of CPU cores available in the new generation, we can predict the usage of SSD and RAM based on the same functions:

$$s = p(c) = \alpha_s + \beta_s c, \tag{11}$$

$$r = q(c) = \alpha_r + \beta_r c, \tag{12}$$

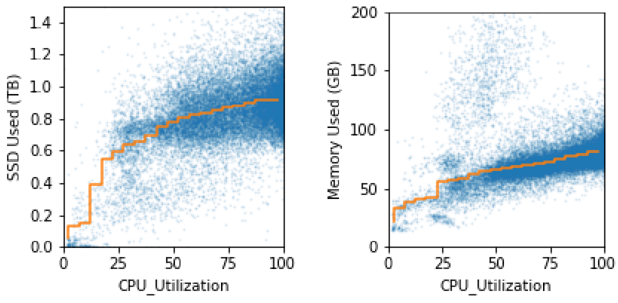


Figure 13: Resource utilization for SSD and RAM

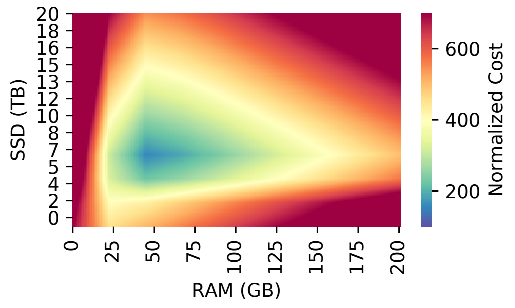


Figure 14: Expected cost with respect to different configurations

where,

c:	number of CPU cores used in the observational data.
s:	amount of SSD used when using c CPU cores.
α_s, β_s:	parameters to predict the SSD usage.
r:	amount of RAM used when using c CPU cores.
α_r, β_r:	parameters to predict the RAM usage.

In Equations (11) and (12), for $p(c)$ and $q(c)$, we use a simple linear regression model. Based on the current data, we calibrate the values for α_r, α_s, β_r and β_s.

Figure 13 shows the current resource utilization for SSD and RAM with respect to different levels of CPU utilization for a particular SKU running with the production workloads. The observation is for each second for a full day with around 10.4 million records.

The α_s and α_r are the intercepts of the projection, indicating the SSD and RAM usage levels when running with 0 cores. The β_s and β_r indicate the SSD usage per core and RAM usage per core. A full distribution with regard to the α_s, α_r, β_s and β_r can be derived based on each observation to capture the nature variances and noises.

For the Optimizer step, the objective is to determine the most cost-efficient size of SSD and RAM for the new machines that have 128 CPU cores. Instead of having a closed form as Equation 7 in Section 5, we use a Monte-Carlo simulation to estimate the objective function, i.e. the expected total cost of each configuration.

We assume that the maximum number of running containers on a machine can be constrained by any of the three resources (CPU cores, SSD, and RAM). The cost of each configuration with different SSD and RAM sizes includes the penalty of idle CPU cores, SSD and RAM based on the unit cost of each resource and the extra penalty of running out of SSD or RAM. Running out of CPU is handled more gracefully in our system than running out of RAM or SSD. For a design with S SSD and R RAM, let α_s and α_r be the calibrated baseline usage for SSD and RAM respectively, we estimate the corresponding objective function by:

(1) Drawing random numbers β_s and β_r from the observational data;
(2) Calculating the maximum number of CPU cores that can be used, c, based on the inverse functions of p and q as following:

$$c = \min\{128, p^{-1}(S), q^{-1}(R)\}.$$

(3) Estimating the quantity of the idle resources. The number of idle CPU cores is:

$$I_c = 128 - c.$$

The amount of idle SSD is:

$$I_s = S - p(c).$$

The amount of idle RAM is:

$$I_r = R - q(c).$$

(4) Estimating the total cost based on the unit price. If there is no idle SSD (RAM), the machine is stranded by SSD (RAM), adding an extra penalty for running out of SSD (RAM).

By repeating the above process 1000 times, we estimate the expected cost for each design configuration with different amounts of SSD and RAM (see Figure 14). If the configuration is designed with insufficient SSD or RAM, the out-of-SSD or out-of-memory penalty dominates the cost. If the configuration is designed with too much SSD or RAM, the penalty of having idle resources increases. Therefore, we are looking for a "sweet spot" where the expected penalty based on the distribution of RAM and SSD usage per core is minimized (lower normalized cost in Figure 14).

6.2 Discussion

The same methodology of Observational Tuning and Hypothetical Tuning is also applicable for optimizing other resources utilization, such as network bandwidth. The optimizer can take either a closed-form formulation and use commercial solvers, or use simple heuristics. In either case, given a predictor of the resulting performance (instead of building a complicated simulation platform), one avoids the need for experiments to deploy new configurations in the production cluster. The set of machine learning models precisely captures the system dynamics in the complex production environment, tailored to the customer workloads.

7 EXPERIMENTAL TUNING

Although Observational Tuning and Hypothetical Tuning cover a large number of applications, the performance impact for some configuration changes, such as changing a software configuration that affects the I/O speed or the introduction of a new feature to improve the processor performance, is still unpredictable. In this case, Experimental Tuning is used. With the introduction of machine-level metrics (see Figure 4), experiments can be done by the deploying experiments to groups of machines in production and conducting A/B testing at a smaller scale.

In this section, we discuss the KEA applications that require experiments to deploy the configuration changes to a group of machines using the flighting tool and subsequent statistical analysis in Phase II (see Figure 3). For this group of applications, the key is the design of the experiments and the determination of performance metrics.

To have a fair comparison between the different groups of machines with different configurations, we hope to control the variables that can potentially affect the performance to the best effort, such as the hardware configurations, the time frame of data collection, even the physical location of the machines. To have statistical significance, we also want to have a relatively large sample size. In this paper, we summarize three possible experiment settings:

- *Ideal setting.* The ideal experiment setting is to have both the experiment group and control group from the same physical location, for example, choosing *every other* machine in the same rack as the control/experiment group. This setting is ideal as it ensures that the two groups of machines are receiving almost identical workloads throughout the experiment, and as they are physically located close to each other, they are often purchased at the same time, and storing data for similar customers. However, sometimes this setting is not feasible, and further adjustments are required.
- *Time-slicing setting.* The time-slicing setting is in general popular in A/B testing. For the same group of machines, this setting deploys consecutively the new and old configurations back-and-forth with a particular frequency, such as every five hours (instead of 24 hours to avoid day of week effects). The evaluation of different configurations is done by measuring the performance during different time intervals. However, this setting, even though it is popular in industry, has several limitations. In the production cluster, it is very difficult to frequently deploy new configurations, and workloads change in the different time intervals, therefore the selection of redeployment interval becomes tricky.
- *Hybrid setting.* When the ideal or time-slicing settings are not feasible, we use the hybrid setting that deploys different configurations to different groups of machines. For this approach to be effective, we need to (i) ensure that the groups of machines have similar characteristics, (ii) use a sufficiently long time period, and (iii) use metrics that are less sensitive to load/demand variation (e.g., normalized metrics).

In this section, we discuss two applications: selecting software configurations and power capping.

7.1 Selecting Software Configurations

In this application, we achieve the ideal setting, i.e. selecting two rows (with approximately 700 machines each) and choose *every other* machine in the same rack as the control/experiment group. We compare two different software configurations which represent using either SSD or HDD for the local temp store. SC1 puts local temp store on HDD and SC2 puts local temp store on SSD. The creation of the SC2 design was motivated by high local temp store write latency for SC1 caused by contention for I/O on the HDD. This write latency created a bottleneck for resource localization in our system.

The experiment was scheduled to run over five consecutive workdays. Table 4 shows the performance impact using metrics that directly reflect the latency and throughput of the system. The Total Data Read per day increased by 10.9% while the average task latency decreased by 5.2%, which is a very significant improvement. In all aspects of the performance of interest, the SC2 machines dominate and the result for Student's t-test shows that the changes are all significant.

Table 4: Performance metrics for different software configurations

Name	SC1	SC2	% Changes	t-value
Total Data Read (PB)	1.38	1.53	10.9%	40.4
Average Task Execution Time (s)	24.1	22.9	-5.2%	27.1

7.2 Power Capping

Compared to the experiment in the previous section, in this application, ideal experiment setting is not possible, i.e. dividing machines to the experiment and control groups in the same rack, as the power capping is at a higher level of control infrastructure and all machines in the same chassis have to be capped at the same level. Moreover, as we require multiple rounds of experiment to test the performance at different capping levels, the data will be collected for different time periods sequentially. Therefore, we use the hybrid setting in this application and focus on the normalized metrics such as Bytes per CPU Time (ratio between Total Data Read and CPU time) and Bytes per Second (ratio between Total Data Read and task execution time), that are less sensitive to the workload level.

We start with the experiment by capping the machines to different provision levels and evaluate their performance. We also evaluate the performance impact for machines with a new feature at the processor level to accelerate processor and graphics performance. For each round of the experiments with a particular level of capping, we collected data for four groups of machines for each SKU tested during the same time period to ensure that those groups of machines are receiving similar levels of workloads:

- Group A with no capping and Feature off
- Group B with no capping and Feature on
- Group C with capping and Feature off
- Group D with capping and Feature on

We selected 120 machines for each group and capped the machines at 10%, 15%, 20%, 25% and 30% below the original power provision level respectively. Each round of experiments ran for more than 24 hours.

Figure 15 shows the performance impact on the two metrics due to different power capping limits for machines with a particular SKU with/without Feature enabled. The y-axis indicates the performance change benchmarked to the baseline, i.e. Group A with no capping and Feature off. With 10% capping, with Feature enabled (blue bars), we can improve Bytes per CPU Time by 5%. Without Feature

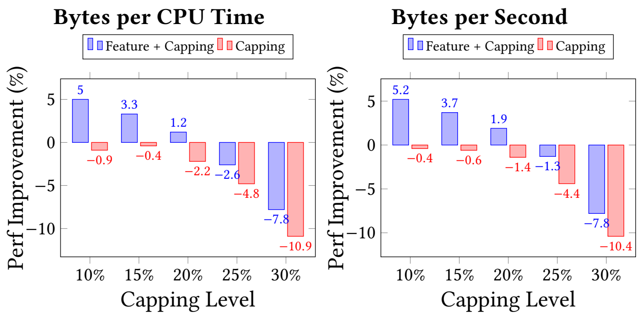


Figure 15: Performance impact due to power capping

enabled (orange bars), the same capping results in the performance degrading by 1%. We can see that at higher power capping levels, the impact becomes more significant. In all cases, having Feature enabled improves the performance.

Similar experiments were also conducted for other SKUs in different clusters to determine the optimal power provisioning limit. We have enabled a relatively conservative capping level which is still much lower than the original level and leads to approximately 10MW reduction in provisioned power that can be harvested to add more machines in the data center.

7.3 Discussion

The analysis for Experimental Tuning is feasible because of the introduction of machine-level metrics that reflect the performance of the machines when running with a large amount of production traffic. It is impossible to isolate the impacts of configuration changes in the job-level metrics, as we cannot control for each job to be executed only in the experiment group or the control group (see Figure 4). We collect data for a relatively long time period to ensure that the machines received a sufficiently large amount of work and the performance is relatively stable for the evaluation of statistical tests.

The KEA infrastructure is used to evaluate many other features of the system and has become a standardized pipeline that leads to significant performance improvement.

8 RELATED WORK

Configuration tuning is vital for the performance of big data applications. Existing research focused on the tuning at the application level, leveraging experiment-based methods such as optimization [25, 29], hill climbing [32, 36], genetic algorithms [37], Bayesian optimization [4], and other heuristic algorithms [23, 57]. Similar methods are used for tuning DBMS instances on small-scale settings [2, 20, 41].

CherryPick [4] used Bayesian Optimization [12, 39, 47] to pick the optimal configuration for Spark and Hadoop applications, such as VM instance type and cluster size. The model aims at minimizing the user cost, calculated as the product of running time and the price per unit with AWS EC2 services subject to a maximum job runtime. As suggested in [22], the Expected Improvement (EI) algorithm was modified to incorporate this extra constraint by scaling the EI value by the probability of satisfying the running time constraint.

H-Tune [29] is a learning-based configuration tuning approach for map-reduce applications. A two-level regression model was trained to predict the execution time at the phase and job level. Based on the prediction models, a genetic algorithm [38] was used to pick the optimal configuration.

MRONLINE [36] is an on-line tuning approach for map-reduce jobs running on YARN-based clusters. The tuner used a gray-box based hill climbing algorithm that consists of two steps, a global search phase to find a promising area and a local search phase to pick the best configuration.

Gunther [37] is a search-based tuning algorithm for map-reduce jobs. A genetic algorithm [38] was used to pick the optimal configuration. The method was tested on two clusters with 16 slave nodes and the results show that with a small number of trials (less than 30), the method is able to obtain a new-optimal solution.

Starfish [26] is a cost-based approach for optimizing map-reduce job performance in an offline setting using a "what-if engine" trained on job profile data. Similarly, Selecta [35] proposed methods for auto-tuning cloud configurations for running spark jobs from the perspective of a cloud user. These work focus on user pricing models where in KEA, we are approaching it from a cloud operator perspective resulting in a entirely different optimization problem.

Zheng et al. [57] proposed a software infrastructure to tune the configuration files in Internet services. One of their key contributions of this method is the notion of dependency graph between the tuned parameters, based on which an optimal searching order for parameters can be derived to improve efficiency. Similarly, by measuring the performance of a target system under different configurations, [33] developed a learning-based technique to identify the most important parameters to tune for achieving better performance in database systems.

Note that all the above related work focuses on small-scale settings and relies on repetitive configuration deployment, which is not amiable (only used as last resort) in large-scale production environments such as ours.

ML and optimization techniques have also been used for resource allocation purposes to improve workload balance and reduce execution bottlenecks [1, 3, 8, 9, 24, 46]. These works leverage information on resource utilization and (predicted) workload characteristics. In contrast, KEA focuses on tuning static configurations to affect scheduling, avoiding the need for dynamic real-time control components that introduce additional parameters and overheads.

9 CONCLUSION

In this paper, we propose KEA, a data-driven system for the configuration tuning of an exabyte-scale data infrastructure. Built upon an innovative way of conceptualization of the system components, KEA captures the essence of our cluster dynamic behavior in a set of machine learning (ML) models and develops three tuning modes for different application scenarios (such as YARN configurations, hardware and data center design, and software investments). Evaluated in the production system, we demonstrate that KEA is able to continuously improve the performance step-by-step and introduce tens of millions of dollars saving per year for the company. Configuration tuning is becoming one of the biggest challenges for large scale systems, which might easily have hundreds even thousands of parameters. We believe that the KEA architecture can be extended for other configurations to further reduce the operational cost.

REFERENCES

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.).

[3] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C Eric Schrock. 2013. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 91–102.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[5] Amazon.com, Inc. 2020. *Amazon Athena*. Retrieved July 4, 2020 from https://aws.amazon.com/athena/

[6] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[7] André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool Publishers.

[8] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. 2020. Toward ml-centric cloud platforms. *Commun. ACM* 63, 2 (2020), 50–59.

[9] Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. 2012. Predicting execution bottlenecks in map-reduce clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.

[10] Eric Boutin, Paul Brett, Xiaoyu Chen, Jaliya Ekanayake, Tao Guan, Anna Korsun, Zhicheng Yin, Nan Zhang, and Jingren Zhou. 2015. Jetscope: Reliable and interactive analytics at cloud scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1680–1691.

[11] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[12] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[13] Nicolas Bruno, Sameer Agarwal, Srikanth Kandula, Bing Shi, Ming-Chuan Wu, and Jingren Zhou. 2012. Recurring job optimization in scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 805–806.

[14] Nicolas Bruno, Sapna Jain, and Jingren Zhou. 2013. Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment* 6, 11 (2013), 961–972.

[15] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[16] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1205–1223.

[17] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, et al. 2020. MLOS: An Infrastructure for Automated Software Performance Engineering. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. 1–5.

[18] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. 2019. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 177–192.

[19] Thomas H Davenport, T Hongsermeier, and Kimberly Alba Mc Cord. 2018. Using AI to improve electronic health records. *Harvard Business Review* 12 (2018), 1–6.

[20] Joyce Fang, Martin Ellis, Bin Li, Siyao Liu, Yasaman Hosseinkashi, Michael Revow, Albert Sadovnikov, Ziyuan Liu, Peng Cheng, Sachin Ashok, David Zhao, Ross Cutler, Yan Lu, and Johannes Gehrke. 2019. Reinforcement learning for bandwidth estimation and congestion control in real-time communications. *CoRR* abs/1912.02222 (2019).

[21] 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*. 99–112.

[22] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. 2014. Bayesian Optimization with Inequality Constraints.. In *ICML*. 937–945.

[23] Mikhail Genkin, Frank Dehne, Maria Pospelova, Yabing Chen, and Pablo Navarro. 2016. Automatic, on-line tuning of YARN container memory and CPU parameters. 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, 317–324.

[24] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 845–861.

[25] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1111–1122.

[26] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[28] Paul W Holland. 1986. Statistics and causal inference. *Journal of the American statistical Association* 81, 396 (1986), 945–960.

[29] Xingcheng Hua, Michael C Huang, and Peng Liu. 2018. Hadoop Configuration Tuning With Ensemble Modeling and Metaheuristic Optimization. *IEEE Access* 6 (2018), 44161–44174.

[30] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing*. 416–427.

[31] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 117–134.

[32] KC Kamal. 2015. *Performance Tuning of MapReduce Programs*. Ph.D. Dissertation. North Carolina State University, Raleigh, NC.

[33] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[34] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[35] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 759–773.

[36] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 165–176.

[37] Guangdeng Liao, Kushal Datta, and Theodore L Willke. 2013. Gunther: Search-based auto-tuning of mapreduce. In *European Conference on Parallel Processing*. Springer, 406–419.

[38] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT press.

[39] Jonas Mockus. 2012. *Bayesian approach to global optimization: theory and applications*. Vol. 37. Springer Science & Business Media.

[40] Art B Owen. 2007. A robust hybrid of lasso and ridge regression. *Contemp. Math.* 443, 7 (2007), 59–72.

[41] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.* (2019).

[42] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. 2019. Hyper dimension shuffle: Efficient data repartition at petabyte scale in scope. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1113–1125.

[43] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 51–63.

[44] Aurobindo Sarkar and Amit Shah. 2018. *Learning AWS: Design, build, and deploy responsive applications using AWS Cloud components*. Packt Publishing Ltd.

[45] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, et al. 2019. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-based Platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 441–452.

[46] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader! online optimization of distributed storage configurations. (2015).

[47] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[48] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.

[49] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[50] Jordan Tigani and Siddartha Naidu. 2014. *Google BigQuery Analytics*. John Wiley & Sons.

[51] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[52] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[53] Stefan Wager and Susan Athey. 2018. Estimation and inference of heterogeneous treatment effects using random forests. *J. Amer. Statist. Assoc.* 113, 523 (2018), 1228–1242.

[54] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[55] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[56] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y Li, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 295–308.

[57] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. 2007. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review* 41, 3 (2007), 219–229.

[58] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 13–24.

[59] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (2012), 611–636.

[60] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 1060–1071.