Lorentz: Learned SKU Recommendation Using Profile Data (DMDS)

NICK GLAZE*, Microsoft, USA
TRIA MCNEELY*, Microsoft, USA
YIWEN ZHU*, Microsoft, USA
MATTHEW GLEESON, Microsoft, USA
HELEN SERR, Microsoft, USA
RAJEEV BHOPI, Microsoft, USA
SUBRU KRISHNAN, Microsoft, Spain

In response to diverse demands, cloud operators have significantly expanded the array of service offerings, often referred to as Stock Keeping Units (SKUs) available for computing resource configurations. Such diversity has led to increased complexity for customers to choose the appropriate SKU. In the analyzed system, only 43% of the resource capacity was rightly chosen. Although various automated solutions have attempted to resolve this issue, they often rely on the availability of enriched data, such as workload traces, which are unavailable for newly established services. Since these services amass a substantial volume of telemetry from existing users, cloud operators can leverage this information to better understand customer needs and mitigate the risk of over- or under-provisioning. Furthermore, customer satisfaction feedback serves as a crucial resource for continuous learning and improving the recommendation mechanism.

In this paper, we present Lorentz, an intelligent SKU recommender for provisioning new compute resources that circumvents the need for workload traces. Lorentz leverages customer profile data to forecast resource capacities for new users based on detailed profiling of existing users. Furthermore, using a continuous learned feedback loop, Lorentz tailors capacity recommendations according to customer performance vs. cost preferences captured through satisfaction signals. Validated using the production data from provisioned VMs supporting Azure PostgreSQL DB, we demonstrate that Lorentz outperforms user selections and existing defaults, reducing slack by >60% without increasing throttling. Evaluated using synthetic data, Lorentz's personalization stage iteratively learns the user preferences over time with high accuracy.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Modeling methodologies**; • **Information systems** → **Autonomous database administration**.

Additional Key Words and Phrases: resource management, machine learning, simulation

*Authors contributed equally.

Authors' addresses: Nick Glaze, Microsoft, 1 Memorial Drive, Cambridge, MA, 02142, USA, nickglaze@microsoft.com; Tria McNeely, Microsoft, 1 Memorial Drive, Cambridge, MA, 02142, USA, triamcneely@microsoft.com; Yiwen Zhu, Microsoft, Mountain View, USA, yiwzh@microsoft.com; Matthew Gleeson, Microsoft, 1 Memorial Drive, Cambridge, MA, 02142, USA, mattgleeson@microsoft.com; Helen Serr, Microsoft, 1 Memorial Drive, Cambridge, MA, 02142, USA, helenserr@microsoft.com; Rajeev Bhopi, Microsoft, One Microsoft Way, Redmond, WA, 98052, USA, rajbho@microsoft.com; Subru Krishnan, Microsoft, Barcelona, Spain, subru@microsoft.com.

ACM 2836-6573/2024/6-ART149
https://doi.org/10.1145/3654952

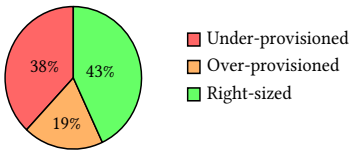


Fig. 1. Users improperly provision many resources on Azure PostgreSQL DB (flexible server).

ACM Reference Format:
Nick Glaze, Tria McNeely, Yiwen Zhu, Matthew Gleeson, Helen Serr, Rajeev Bhopi, and Subru Krishnan. 2024. Lorentz: Learned SKU Recommendation Using Profile Data (DMDS). *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 149 (June 2024), 25 pages. https://doi.org/10.1145/3654952

1 INTRODUCTION

The availability of public cloud services enables easy access to a wide range of data services with diverse analytic requirements (e.g., SQL/NoSQL databases, streaming, machine learning, business insight analysis, and a lot more) [39]. However, the complexity of choosing optimal offerings increases significantly when a large number of choices are exposed. Cloud operators have invested tremendous efforts to provide "knob-free" services by automating numerous aspects throughout the onboarding process. For instance, various tools [16, 19] have been developed to aid in selecting the appropriate target, facilitating smooth migration from on-premises to the cloud. However, many of these tools still necessitate substantial input from customers. Model-driven approaches like Doppler [2] construct customer profiles and utilize clustering analysis to glean preferences from existing customers, which can then be applied to new ones, achieving full automation. Automated configuration tuning, leveraging ML methods such as Bayesian optimization, has gained recent attention as a viable approach for constant, online configuration tuning [4, 6, 14, 22, 38]. However, tools designed to support such scenarios typically require a detailed historical trace for each workload they operate on, and thus are more applicable to optimizing existing workloads than new ones. As of late, there is limited discussion about automating SKU selection for initial resource provisioning. At Microsoft, for example, users of Azure PostgreSQL DB are simply provided a static default SKU, which they select much more often than is appropriate; this results in significant COGS loss as well as performance throttling. As shown in Figure 1, only 43% of users correctly provision their services, further exacerbating their losses.

In this paper, we tackle the selection of *initial* optimal SKUs for newly-created services (such as PaaS databases or VM instances). We discuss a method to circumvent the requirement for workload trace data, particularly for services whose configurations are inconvenient to modify after being initially provisioned (e.g., when autoscaling induces long reboot times). In approaching this problem, several challenges emerge:

Scarcity of input data [C1]: In contrast to scenarios involving service migration, where telemetry can provide comprehensive insights into each workload, initial workload configuration presents a distinct challenge due to the limited availability of information. In this scenario, inference must be made without data related to the workload traces, instead relying on customer profile data like industry, company, or customer IDs. With the recent advent of machine learning and data science, enriched profile data has demonstrated increasing utility in facilitating detailed customer management tasks [12, 32], showcasing significant potential for enhancing SKU recommendation processes.

Customer heterogeneity [C2]: Cloud customers have widely varying preferences in the trade-off between price and performance. Configuration recommendations must account for these preferences and integrate them into the solution. Employing a one-size-fits-all solution for all

customers is prone to resource over-provisioning or under-provisioning, failing to align with their distinct preferences.

Explainability [C3]: Since configuration recommendations have a significant influence on future cost and performance outcomes, customers often seek a comprehensive understanding of the underlying reasoning behind them. Customers rely on this transparency to ensure they are well-informed to make final judgments about the fidelity of recommendations they are presented with. Good explainability also leads to better user adoption given comprehension of the decision-making process and trust in results, encouraging potential user interaction.

Lack of properly labeled data [C4]: We observed a considerable proportion of instances in Azure PostgreSQL DB to be either over-provisioned or under-provisioned (Figure 1). Existing customers' choices are therefore unreliable as "labels" for training any downstream configuration recommendation models.

Introduction to Lorentz. To address these challenges, we propose *Lorentz*, an automated system for initial SKU recommendation that is applicable to provision VMs, DBs, or other cloud resources. Lorentz consists of three stages: (1) *capacity rightsizing* for exiting workloads based on slack (unused quantity of resources) and throttling (severity of resource crunching), leveraging auto-scaling strategies to account for biases in the training data; (2) an innovative explainable *SKU recommender* that bases predictions on similar customers' instances; and (3) a continuous running *personalizer* that dynamically learns the price-performance preference for each customer based on feedback signals such as Customer Reported Incident (CRI) information through a message propagation process.

Specifically,

(1) Lorentz aggregates data from diverse origins, including subscription (each corresponds to one user account) particulars, resource specifics, customer engagements, and feedback from current users to make each SKU recommendation [**C1**]. Lorentz draws insights from the selection patterns of existing users, predicated on their similarities. The rationale behind this is that similarity between customers' compute resources, subscriptions, or profile data might imply similarity between their resource capacity needs. For example, databases provisioned by Coca-Cola and Pepsi might have similar needs, since they share a common industry. Lorentz opens up the opportunity to gain useful business insight from profile data to support critical decision-making and is easily extensible should more enriched information becomes available.

(2) That Lorentz recommends based on similarity in customer profile data enables design of explainable recommender models [**C3**]. When generating each recommendation, Lorentz can additionally provide the "search result" from the referenced database to help customers understand what characteristics of themselves or their new compute resource were useful for the recommendation. We provide two distinct algorithms to characterize customers and their services to be provisioned.

(3) Lorentz learns each customer or customer segment's price-performance preferences by assessing historical customer interactions with resource provisioning, scaling actions, and performance-related CRIs [**C2**]. We provide a comprehensive feedback loop to constantly adjust the recommendation strategy at the individual level to capture customer heterogeneity. Each customer is tagged with *preference scores* which is updated periodically to reflect learning of their preference along the price-performance trade-off. We developed an innovative learning algorithm to account for potential network impact across different resources provisioned by the same user, or within the same industry (e.g., Banking or Retail).

(4) Lorentz provides flexible, modular, and general solutions that can support provisioning, pooling, and rightsizing operations across a multitude of data ecosystems and beyond. The general solution is straightforwardly extensible as additional profile data features become available.

(5) Built on top of existing autoscaling algorithms, such as [2, 29], we make adjustments to the existing customers to "provision" the right SKU as the reference [**C4**] to ensure accuracy and improve on satisfaction.

Contributions. In sum, our contributions are as follows:

- We develop an innovative prediction method for SKU recommendation based on profile data that leverages "similar" customers who have already been onboarded to the cloud service.
- We provide two algorithms (hierarchy traversing and target encoding) to enable flexibility and explainability.
- We provide personalization by incorporating an online learning mechanism to constantly adjust the recommendation policy for each customer by capturing signals about customer satisfaction along the price-performance trade-off.
- We evaluate Lorentz using production workload traces from Azure PostgreSQL DB as well as synthetic workloads. Compared to our baseline, Lorentz reduces wasted capacity by over 60% without increasing throttling. Validation on synthetic signals also confirms that Lorentz's personalization stage converges to align with customer preferences.

The remainder of this paper is organized as follows. Section 2 describes the cloud resource that Lorentz is currently applied to and evaluated upon. Section 3 presents the overall architecture and each module in detail, with implementation details provided in Section 4. Section 5 shows experiment results. Section 6 overviews related work, and, Section 7 concludes the paper and discusses future work.

2 BACKGROUND

Although Lorentz is applicable to any cloud resource with available customer profile data, we demonstrate it in this work for Azure PostgreSQL DB (flexible server). In this section, we discuss this database system and the data it exposes to Lorentz.

2.1 DB SaaS

Microsoft is investing in a 5x5 developer experience for Database (DB) Software-as-a-Service (SaaS). One of the keys to this experience is abstracting the provisioning and tuning of DB configurations, such as the SKU, to provide knob-free services. Estimating the necessary hardware capacity for DBs is a common pain point for developers, especially for newly created workloads. The current Azure PostgreSQL DB (flexible server) configuration tool merely provides the minimum-capacity SKU as the default choice, placing the burden of SKU selection entirely on users. Thus, the stated goal of Lorentz is:

> *"There will be no capacity input/usage prediction from end users needed during DB provisioning. The capacity management for the database instance will all be internally managed."*

That is, Lorentz must recommend initial virtual machine (VM) SKUs upon DB creation.

Existing works such as Doppler [2] and Autopilot [29] recommend DB capacities based on observed workloads: once a DB is up and running, these systems observe that DB's resource utilization patterns to identify future optimal capacities in various dimensions. The key innovation of Lorentz is making these recommendations before the DB is initialized—i.e., without any utilization data. While DB migration and autoscaling recommenders act on historical usage patterns, the Lorentz engine must recommend a capacity based only on customer profile data and limited database-specific profile data. Doppler [2] focuses on workload migration where the target database's traces are available. Compared to Lorentz, both methods are grounded in insights gleaned from existing

provisioned databases. However, Doppler requires extensive workload telemetry from the target database as input, rendering the method impractical for our scenario.

Lorentz is generalizable to capacity recommendation for all compute resources, not just DB services. For simplicity and ease of generalization, Section 3 illustrates the application of Lorentz on capacity recommendation for VM resources in general. This is purely a vocabulary decision—we still evaluate Lorentz 's efficacy solely on the set of Azure PostgreSQL DB databases. In a similar vein, we implement Lorentz in a generalizable way, allowing the system to be configured to the specific needs of arbitrary capacity recommendation tasks.

For Azure PostgreSQL DB at Microsoft, services are *stratified* into three *server offerings*, each catering to different use cases: Burstable (dev), General Purpose (small production), and Memory-Optimized (large production). The respective offerings induce distinct sets of candidate capacities:

$$\text{Burstable: } \{1, 2, 4, 8, 20\}$$
$$\text{General Purpose: } \{2, 4, 8, 16, 32, 48, 64, 96, 128\}$$
$$\text{Memory-Optimized: } \{2, 4, 8, 16, 20, 32, 48, 64, 96, 128\}$$

Burstable, General Purpose, and Memory-Optimized VMs make up 5%, 49%, and 46% of the user database, respectively. Throughout this work, all statistics and performance metrics describe the global average across all three server offerings. Since provisioning preferences vary significantly across the three offerings, we train a distinct parameter set per offering for all recommendation models. Lorentz also assumes that server offering is pre-selected, since users select their desired offering before choosing resource capacity.

2.2 Data

Lorentz requires three main data categories to recommend DB SKUs: (1) Utilization telemetry for similar existing provisioned DBs, (2) profile data tags for existing and new DBs, and (3) signals of customer satisfaction. This section details each of the required data, and summarizes the Azure PostgreSQL DB (flexible server) l0dataset we use to demonstrate Lorentz. Our method is generalizable to any DB, VM, or other system that surfaces the necessary data described here.

Telemetry for existing DBs. Existing DBs in the system will be used as reference points when provisioning for new DBs. For each resource dimension (e.g., CPU, memory, IOPS), Lorentz requires the following descriptive data: (1) the available capacity options (i.e., SKUs) for the service, (2) the currently selected capacity for each existing DB, and (3) telemetry tracking resource consumption for each existing DB. In sum, Lorentz requires:

- **Capacity/SKU options.** We use a table containing regularly-updated SKU selections for each Azure PostgreSQL DB to derive capacity recommendation options and current resource capacities.
- **Currently-selected SKUs.** Lorentz references existing DBs' SKU choices when recommending new configurations.
- **Resource utilization telemetry.** For existing DBs, we require compute utilization telemetry with at a high resolution to validate (or adjust) chosen SKUs.

Profile Data. *Profile data* refers to any categorical variable describing a customer or DB instances. At Microsoft, the billing team records such data using identifiable tags for every chargeable cloud resource created. This can include software versions, localization tags (e.g., region or country in which the compute instance resides), or development/test/production tags.

The profile data consists of: (1) individual database-level details, including the database ID, resource group name, and subscription ID. This information is essential for billing, ensuring that

usage is attributed to a specific account; (2) Comprehensive details about each customer, such as industry and segment names. This data serves multiple purposes, aiding in marketing efforts and facilitating customer management. Establishing detailed user profiles is a common industry practice for supporting various analyses. For example, companies like Uber and Lyft [32] utilize detailed customer profiles, including regional information, interviews, and behavior patterns, to inform promotion strategies and pricing regimes. Similarly, Facebook automatically tags individual users with learned preferences, such as political stances [20].

Note that these preferences are often acquired through automated processes rather than manual logging, various methods, including machine learning, can be leveraged. Database providers like Salesforce [31], AWS [1], and Oracle [23] offer professional software solutions to manage such customer profile data. At Microsoft, this information is stored in an internal database, ensuring secure access through authentication, and facilitating downstream analysis.

For resource provisioning on Azure PostgreSQL DB, Lorentz utilizes the requested DB's SKU family (i.e., server offering) and resource group (usually created to support a particular application or project), along with the hierarchy of customer profile data, from granular unique service account IDs (i.e., subscription ID) to broad customer segmentation tags like industry name (e.g., Food and Drink).

In sum, Lorentz leverages the following profile data:

- **Resource Tags.** Lorentz pulls resource-specific tags like software version or selected server offering from the same tables as capacity and utilization.
- **Customer Profile Data.** Low-level customer profile data (e.g., service account or subscription ID and resource groups) are inferred from DB instance ID paths in the utilization table. Higher-level profile data comes from the account's metadata table from the Billing department.

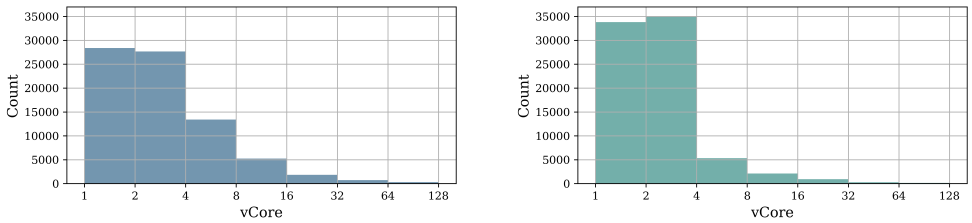
Customer Satisfaction Signals. Customer satisfaction signals comprise anything that indicates (1) a cost sensitivity—a customer prefers cheaper offerings and is willing to take slight performance hits to reduce cost—or (2) a performance sensitivity—a customer prefers more performant offerings and is willing to pay more to ensure they never experience any throttling. Examples include support tickets and manual scale actions on resources within the service.

For Azure PostgreSQL DB's resource provisioning, Lorentz utilizes performance and cost-related Customer Report Incidents (CRIs), which are currently labeled via a manually-crafted keyword search. We envision that more advanced CRI processing should use Large Language Models (such as a member of the OpenAI GPT family of models [7]) to more accurately assess the strength of perf/cost signals from each ticket.

The CRI data serve as indicators providing insights into customer preferences. Additional signals may be derived from the feedback loop, incorporating real usage and error data, as well as a sensitivity analysis comparing customer performance to cost. This analysis considers the actions of existing and new customers in adjusting capacity for this workload and similar cohorts, with these user operations being routinely recorded.

In our preliminary dataset comprising approximately 4,400 CRI tickets submitted by Azure PostgreSQL DB users, the algorithm discerns approximately 2,400 instances of neutral sentiment (0), around 2,000 cases of performance-sensitive sentiment (1), and 5 instances of price-sensitive sentiment (-1). This aligns with our expectations, indicating that a significant portion of these CRIs are expressions of dissatisfaction related to throttling via CRI data—a more prevalent scenario compared to capacity reduction.

Dataset Summary. For each DB provisioned on Azure PostgreSQL DB with a specific SKU, we notate the following information:



(a) Distribution of user-selected vCore capacities (b) Distribution of rightsized vCore capacities

Fig. 2. Rightsizing focuses the capacity distribution to prevent over- and under-provisioning.

- $\mathbf{u}(t)$: usage data for all the resource dimensions of interest (e.g., vCores or memory), accessed from telemetry; formatted as a high-resolution time series of vectors, and each entry $u_r(t)$ corresponds to one resource dimension r;
- $\mathbf{x}$: a feature vector describing the DB and its customer (e.g., by the customer ID, industry name, etc.);
- $\mathbf{c}^0$: the DB's customer-selected SKU, i.e., the SKU of the VM that hosts the service, depicted as a vector indicating the amount of resources provisioned, e.g., [4vCores, 5GB] for vCores and memory respectively.

Note that for any target (new) DBs to be provisioned, only profile data is available while for existing provisioned DBs, both usage data and profile data is available. We let the matrix $\mathbf{X}$ denote all the profile data records across DBs, where each row contains the profile data $\mathbf{x}$ for a particular DB.

The dataset contains 77,584 DBs, each corresponds to one VM. And we select 7 of the profile data features derived in Section 5. Workloads $\mathbf{u}(t)$ describe at most 7 days of resource usage, sampled approximately once per minute.

For the analyzed system, we found that:

- Users select the ideal resource capacity only 43% of the time: 19% over-provision and 38% of the time under-provision, relative to Lorentz's rightsized capacities (see Section 3.2).
- Users are approximately equally good at selecting capacities for development and production DBs.
- For development DBs:
  - Users under-provision for 54% of DBs, but over-provision only 6%.
  - Users choose the minimum (default) capacity for 80% of dev DBs, but it is only appropriate for 38% of them. Note that this is likely because the default choice presented to users is the minimum capacity; across all servers, users select this choice 63% of the time.
- For production DBs:
  - Users make balanced capacity guesses for production DBs Users are correct for 45%, under-provision for 25%, and over-provision for 30% of prod DBs.

3 LORENTZ FRAMEWORK

In this section, we describe the Lorentz framework, including the architecture of and interactions between each stage. To address the challenges listed in Section 1, we introduce a 3-stage pipeline.

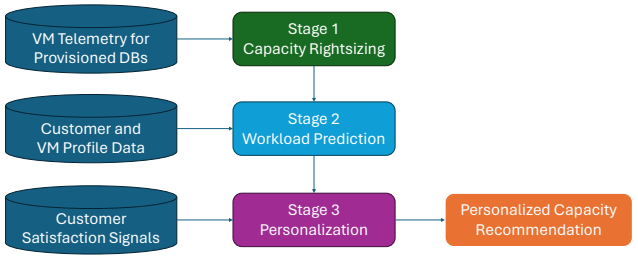


Fig. 3. Inputs and outputs of each stage of the Lorentz capacity recommendation framework.

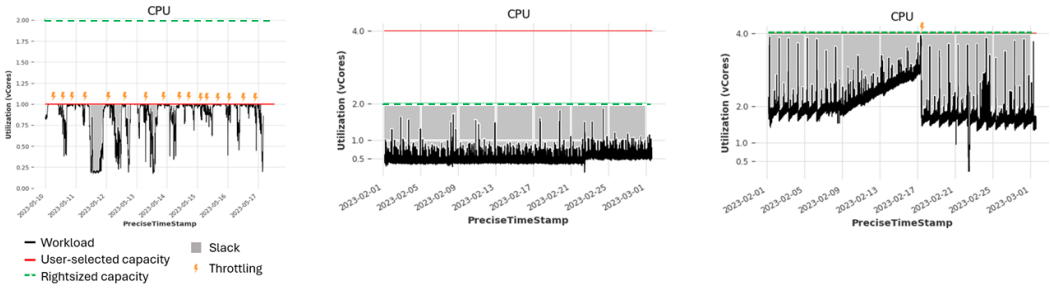


Fig. 4. CPU slack and throttling for under-provisioned (left), over-provisioned (center), and well-provisioned (right) Azure PostgreSQL DB database VMs. The dashed line shows the rightsized SKU.

3.1 Overview

Figure 3 illustrates the three stages of Lorentz: Stage 1 of Lorentz computes the best-fit capacity for an existing VM instance, based on its usage telemetry; Stage 2 leverages these rightsized capacities as "labels", along with customer profile data characterized as "features", to recommend resource capacities for new workloads; and, finally, Stage 3 takes arbitrary signals of customer satisfaction to continuously learn each customer's relative cost/performance sensitivity, enabling Lorentz to provide personalized capacity recommendations for each customer's resource provisioning requests.

3.2 Stage 1: Capacity rightsizing

Stage 1 begins with rightsizing the SKUs for existing users, especially for the ones that are over- or under-provisioned (see Figure 4). In the telemetry data collected for existing workloads, let $\mathbf{u}(t)$ denote the irregularly sampled time series that measures the raw resource utilization for a DB hosted on a VM (up to the user's selected capacity, denoted as $\mathbf{c}^0$). We have thus:

$$u_r(t) \leq c_r^0, \qquad \forall t, r, \tag{1}$$

where t is the time stamp and r the resource dimension index, such as CPU, memory, and IOPS.

First, to standardize its temporal resolution, we group the irregularly-spaced samples $\mathbf{u}(t)$ into T-minute bins and aggregate each bin to a single observation, producing a regular workload resource usage signal $\mathbf{w}[n]$ with fixed time intervals. Our use case usually samples utilization approximately every minute, and we select $\max(\cdot)$ as our aggregator to measure worst-case performance thus avoiding under-provisioning.

$$\mathbf{w}[n] = \max(\{\mathbf{u}(t) | n \leq \frac{t}{T} < n+1\}). \tag{2}$$

To compute the rightsized capacity of existing DBs using the regularly sampled workload trace $\mathbf{w}[n]$, we design a selection criterion that selects the optimal SKU candidate from a set of candidate capacities C, based solely on the workload resource usage. We first compute two opposing statistics describing the resource utilization: *throttling* and *slack* for each SKU candidate $\mathbf{c}$, and then define an optimizer that selects the capacity that best balances the two statistics for the rightsizing.

Throttling. When the selected capacity is too small to accommodate the workload, *throttling* occurs; we call this resource *under-provisioned* (see left of Figure 4). For some resources (e.g., memory), this can result in the cancellation of jobs (e.g., out-of-memory errors); this is nearly always an undesirable behavior. For other resources like CPU, jobs are simply delayed until the requisite resource space is available, which can more often be accommodated. Tolerance for throttling may vary across use cases and resource dimensions; for example, production services with tight reliability and latency requirements may be particularly sensitive to throttling, while minor throttling for CPU can be more tolerable than memory.

To compute throttling for SKU candidate $\mathbf{c}$ given the workload with resource usage $\mathbf{w}[n]$, we first set a ratio threshold η_r of resource utilization for each resource dimension, above which performance degradation can be expected. This value should be less than 1. This threshold also indicates the optimal utilization level that users hope to achieve while minimizing the amount of unused resources without impacting performance. Given resource usage of a workload, $\mathbf{w}[n]$, we define the probability of throttling $T_w(\mathbf{c})$ under candidate capacity $\mathbf{c}$ as:

$$t_w[n] = \begin{cases} 1 & \text{if } \exists r \text{ s.t., } w_r[n] > \eta_r c_r \\ 0 & \text{otherwise} \end{cases}, \tag{3}$$

$$T_w(\mathbf{c}) = \frac{1}{n}\sum_n t[n], \tag{4}$$

where $T_w(c)$ captures the percentage of time frames where the resource is under-provisioned (there exists one resource dimension r whose utilization is above η_r).

Slack. Whenever a resource capacity is larger than necessary, some of the available resources go unused; we refer to this unused capacity as *slack* and call the resource *over-provisioned* (Figure 4, middle). In practice, the presence of some slack is unavoidable, since it is not feasible or efficient to scale resource capacity to exactly match the needs of upcoming jobs, even if those needs could be perfectly predicted. Still, users aim to minimize slack, since an excess of this wasted space incurs costs that would have been avoidable with better capacity selection.

We define the average slack ratio vector $\mathbf{S}_w(\mathbf{c})$ induced by picking candidate capacity $\mathbf{c}$ on workload $\mathbf{w}[n]$ as follows, where each entry corresponds to one resource dimension r:

$$s_{w,r}[n] = \frac{c_r - w_r[n]}{c_r}, \tag{5}$$

$$S_{w,r}(\mathbf{c}) = \frac{1}{n}\sum_n s_{w,r}[n], \tag{6}$$

where $\mathbf{S}_w(\mathbf{c})$ captures the percentage of resources that will be unused given the provisioned resource amount of $\mathbf{c}$, averaged over time, and can be estimated for each resource dimension respectively.

Note that the observed resource usage is always bounded by the original user-selected capacity $\mathbf{c}^0$ from the telemetry data, and $0 \le w_r[n] \le c_r^0\ \forall r, n$. Therefore, in instances of throttling, the true requested quantity of the resource is censored by the capacity chosen by the user. In such cases where $\mathbf{c} > \mathbf{c}^0$, $\mathbf{S}_w(\mathbf{c})$ as computed in Equation (6) is thus an overestimation of the true slack. To account for this situation, slack computed using Equation (6) is no longer the critical

selection criterion in the process of rightsizing when the observed workload has been throttled (see Equation (8)).

Basic rightsizing optimizer. Slack is directly proportional to resource capacity, while throttling is inversely related to capacity; it is rarely possible to achieve both near-zero slack and zero throttling. We cast the navigation of this slack/throttling trade-off as an optimization problem.

Our optimizer must select the best capacity choice from the SKU candidate set for workload $\mathbf{w}$: $\hat{\mathbf{c}}^0(\mathbf{w}) \in C$. Users generally have stricter requirements for throttling (e.g., four-nines reliability) than for slack, so we construct our optimization function as a slack target $\mathbf{s}^*$ subject to a throttling constraint (upper bound) of τ. Especially for un-censored data (where the workload was not throttled), the rightsized capacity $\hat{c}_r$ for each resource dimension r is as follows:

$$\hat{c}_r^0(\mathbf{w}) = \underset{\mathbf{c}\in C}{\operatorname{argmin}} |S_{w,r}(\mathbf{c}) - s_r^*|, \quad \text{s.t.}, T_w(\mathbf{c}) \leq \tau, \forall r, \tag{7}$$

where s_r^* represents the optimal slack we hope to preserve for each resource dimension and the maximum percentage of throttling $\tau = 0$ in this work as a conservative, throttling-averse starting point that can be calibrated through personalization. Combined with Equation (3), we provide two alternatives for resource reprioritization. In Equation (3) where we define throttling as a ratio η_r of total capacity for each resource dimension r independently, if throttling e.g. disk, is undesirable, its μ_r can be lower. Second, the slack target s_r^* in Equation (7) can be tuned, enforcing low targets for costly resources like memory and higher targets for cheaper ones like disk space.

Rightsizing throttled workloads. An additional effect of the workload censoring described previously is that both the true slack and the throttling probability for candidate capacities $\mathbf{c}$ larger than the user-provisioned capacity $\mathbf{c}^0$ cannot be exactly computed, due to the censored observed resource usage level. Resource requests larger than $\mathbf{c}^0$ are censored to equal $\mathbf{c}^0$, and $T_w(\mathbf{c})$ will always be $\mathbf{0}$ when $\mathbf{c} \geq \mathbf{c}^0$, and the slack is overestimated.

When throttled, we assume that the rightsized capacity should be at least 2^K times the current capacity. The calibration of K can be done by examining the telemetry data of formerly throttled servers whose capacities have been scaled up. To rightsize censored workloads, we use the following optimizer instead, $\forall r$:

$$\hat{c}_r^0(\mathbf{w}) = \underset{\mathbf{c}\in C}{\operatorname{argmin}} |S_{w,r}(\mathbf{c}) - s_r^*|, \quad \text{s.t.}, c_r \geq 2^K c_r^0. \tag{8}$$

Complete rightsizing optimizer. Combining the censored and uncensored cases, we define the following complete optimizer, the rightsized SKU $\hat{\mathbf{c}}^0(\mathbf{w})$ is computed as:

$$\hat{c}_r^0(\mathbf{w}) = \begin{cases} \underset{\mathbf{c}\in C}{\operatorname{argmin}} |S_{w,r}(\mathbf{c}) - s_r^*|, & \text{s.t.}, T_w(\mathbf{c}) \leq \tau \quad \text{if } T_w(\mathbf{c}^0) = 0, \\ \underset{\mathbf{c}\in C}{\operatorname{argmin}} |S_{w,r}(\mathbf{c}) - s_r^*|, & \text{s.t.}, c_r \geq 2^K c_r^0 \quad \text{otherwise.} \end{cases} \tag{9}$$

As we describe in the following section, the rightsized capacities that this optimizer computes become useful labels for training models that recommend resource capacities for new workloads. For Azure PostgreSQL DB, we focus on the analysis for the *virtual cores* (vCores, i.e., CPU) resource as the capacity for memory is provisioned proportional to the number of vCores, e.g., 4GB per core. And the estimation can be simplified as CPU constraints mostly dominate.

3.3 Stage 2: Capacity recommenders

Lorentz's second stage recommends resource capacities for a newly-requested VM for the database, based on resource capacities of VMs performing similar duties. We provide two alternative models for this task, which we refer to as *provisioners*. Since the VMs that Lorentz provisions for are new,

they have no telemetry data for workload traces upon which to compute VM/workload similarity; instead, we use customer- and server-level profile data as input features. We leverage existing users' provisioned SKUs and their profile data as the training dataset. For labels of this training dataset, we use the rightsized capacities computed in Stage 1 from provisioned resources $\hat{\mathbf{c}}^0$ (as opposed to user-selected SKUs $\mathbf{c}^0$), since they are conceptually and empirically an improvement over user-selected capacities.

Notation. Let $\mathbf{x} \in \mathbb{R}^{M_c+M_s}$ be a vector with M_c customer-level and M_s server-level profile data features for a newly-requested VM. The goal of provisioner models is to provide a function mapping $f : \mathbb{R}^{M_c+M_s} \rightarrow \mathbf{c} \in C$ for the recommendation. Let $\mathbf{c}^*$ denote the recommended SKU candidate for Stage 2, we have: $\mathbf{c}^* = f(\mathbf{x})$.

Transformations. For regression methods, the exponential scale of many compute resources (e.g., vCores $\in \{1, 2, 4, 8, 16, ...\}$) can lead to undesirable statistical properties, including heteroskedastic noise [36], which in turn hurts prediction efficiency. We thus fit our models in a transformed space, e.g., $\tilde{c_r} = \log_2(c_r)$, then invert the transform for our predictions. We generally denote this transform and its inverse as $\xi(\cdot)$ and $\xi^{-1}(\cdot)$.

Hierarchical provisioner. Our first provisioner model is a heuristic method that explicitly leverages hierarchical relationships in the input feature to make recommendations. For instance, in the observed profile data, one customer account (referred to as CloudCustomerGuid) can have multiple subscription IDs, which can further have multiple resource groups, etc. We want to learn the hierarchy of all the entries (including both customer- and server-level features) in the profile data automatically. Thus we train the hierarchical provisioner with two steps:

(1) Compute the hierarchical structure of the service's observed profile data features;
(2) Sort each observed resource capacity into "buckets" along the computed hierarchy's feature values, similar to building a decision tree for the inference.

Inference is then performed by selecting the bucket best fitting the new cloud resource's profile data, and recommending a capacity based on samples resigned in that bucket.

To compute the profile data hierarchy, we first leverage the HALO entropy-based approach [37] to identify the strength of hierarchical relationships between each pair of profile features. A Python implementation of the paper can be found in [21]. Although the HALO approach was designed with strict hierarchies (where each node except the root has exactly one parent node) in mind, we find it effective for our non-strict case as well.

To compute the hierarchy chain $\mathbf{h}$, we construct a weighted directed acyclic graph (DAG), using the thresholded table as the adjacency matrix; each edge points toward the coarser node. We then select the node with the highest out-degree as the root $\mathbf{h}_0$ of the hierarchy and traverse the graph by selecting the current node's highest out-degree neighbor until a node with out-degree 0 is reached [37]. In the DAG, lower granularity features always point towards higher granularity features. Figure 5 shows an example of the learned hierarchy, i.e., SegmentName > IndustryName > VerticalName > VerticalCategoryName > ... > ServerName, where for samples share the same value for the child node features (e.g., both have h_2='Insurance'), they also share the same value for the corresponding parent node (e.g., both have h_1='Financial Services'.

Based on the learned hierarchy of profile features, the next step is to populate a set of buckets B along the computed hierarchy. For each feature $\mathbf{h}_j, 0 \leq j < |\mathbf{h}|$, we define one level of buckets $B_{\mathbf{h}_j}$ (see Figure 5); within each level, we initialize one bucket $B_{\mathbf{h}_j,v}$ for each unique value v of feature $\mathbf{h}_j$. We populate each bucket with the capacities of existing provisioned VMs based on the profile data

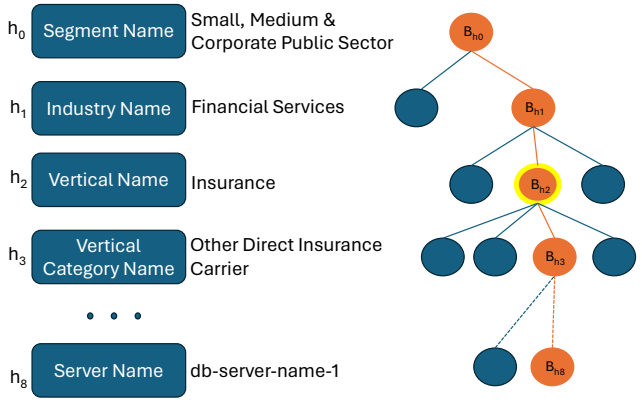


Fig. 5. Example of the learned hierarchy and buckets.

for each with that bucket's feature value[1] such that:

$$B_{\mathbf{h}_j,v} = \{\hat{\mathbf{c}}_i^0 | x_{i,\mathbf{h}_j} = v\}. \tag{10}$$

To recommend the capacity for a newly-provisioned VM resource with profile feature vector $\mathbf{x}$, we first select its most granular feature $\mathbf{h}_k$ whose feature-value pair has a sufficient bucket size at the corresponding level, i.e., $\geq N$. We then return the p^{th} percentile of the observed capacities in that bucket as the recommendation $\mathbf{c}^*$:

$$k = \underset{l}{\text{argmax}}(|B_{\mathbf{h}_l,\mathbf{x}_l}| \geq N), \tag{11}$$

$$\mathbf{c}^* = f(\mathbf{x}) = \%\text{ile}(B_{\mathbf{h}_k,\mathbf{x}_k}, p), \tag{12}$$

where $\mathbf{c}^*$ is the recommended SKU candidate from Stage 2 using the mapping function f.

Figure 5 provides an example of inference. The profile data of a hypothetical VM indicates a particular sequence of buckets it belongs to, following the computed hierarchy $\mathbf{h}$. Starting from $\mathbf{h}_8$ (ServerName), we might find that bucket $B_{\mathbf{h}_8,\text{db-server-name-1}}$ is not sufficiently large to make a recommendation. In this case, we traverse up the hierarchy until reaching $B_{\mathbf{h}_2,\text{Insurance}}$, the first sufficiently-large bucket. We observe from this bucket the distribution of rightsized capacities $\hat{\mathbf{c}}^0$ (computed in Stage 1) for all servers in the "Insurance" vertical, taking the p^{th} percentile as the capacity recommendation.

Target encoding provisioner. Our second provisioner model uses target encoding (TE) to admit non-parametric regression or classification ML models like tree-based models which are best on tabular data [13], such as LightGBM [17]. This approach is suited to (i) data with non-hierarchical relationships between profile data tags and (ii) large enough data size that arbitrary relationships between variables can be learned by non-parametric models. While the framework technically admits arbitrary categorical variable encodings and regression/classification methods, we choose tree-based predictors and TE because (i) tree-based predictors offer best-in-class performance on tabular data and (ii) alternative encodings–such as one-hot encoding–often behave poorly with trees and high-cardinality profile data tags.

[1]Due to user mis-entry of profile features, the hierarchies in our dataset are nearly, but not perfectly strict (i.e., HI values slightly less than 1). As a result, it is most appropriate to index bucket $B_{\mathbf{h}_j,v}$ by a single hierarchy level $\mathbf{h}_j$ (instead of including all features coarser than $\mathbf{h}_j$), since this eliminates noise in the coarser features.

Target encoding converts every categorical feature into a numeric using statistics from the whole training data. For instance, in the column/profile data entry of SegmentName, one particular sample has the value "Beverage". This value is replaced with the average vCores (i.e., the label) from the subset of samples whose SegmentName equals "Beverage". The average vCore values can be later used as ML prediction features to predict the same label. This approach can be broken into two steps:

(1) For each row (i.e., one VM) in the training data X, for each attribute at entry h, x_h, based on the SKU label $\hat{\mathbf{c}}^0$ (after rightsizing) of this VM and feature value $x_h = v$ such as "Beverage", we map $\text{TE}(x_h) = \psi_{\{n|X_{n,h}=v\}}(\mathbf{c}_n)$, where $\psi(\cdot)$ is some aggregation function, such as a mean or percentile. The set $\{n|X_{n,h} = v\}$ includes all the observations of VMs whose profile feature at entry h equals the observed value v. $\text{TE}(x_i)$ estimates the statistics of the labels from the subset of samples given a particular attribute value.
(2) After transforming all the features in all observations $\mathbf{x} \in X$ (i.e., the ones we learned in the hierarchy model) with the mapping $\text{TE}(x_h)$, use a traditional regression method to estimate $\mathbf{c}^* = f(\mathbf{x})$.

Missing data. The handling of missing profile data tags with a TE approach is non-trivial. We explored the options of replacing the tag "missing" in each attribute with an invalid numerical value, e.g., -999, but found both random forest [34] and gradient-boosted trees [33] unable to differentiate this case from the other encoded values, resulting in severe underestimation in the presence of missing data. This problem vanishes when replacing the tag "missing" with the average of $\hat{\mathbf{c}}^0$ over the whole training set.

Extension to include trace data. The method can be extended to incorporate additional data types. For instance, if trace data becomes accessible, both the hierarchical and target encoding models can utilize more features as inputs, thereby enhancing prediction accuracy. Specifically, the inclusion of more features contributes to improving the variability of SKU choices within each bucket, leading to a more powerful predictor. In this context, Lorentz can serve as a predictive tool to assist in decision-making for autoscaling after resources have been provisioned.

3.4 Stage 3: Personalization

By training Stage 2 of Lorentz on the rightsized capacities from Stage 1, the system is able to predict a capacity fitted to the unseen workload of a new resource based on profile features $\mathbf{x}$. However, no matter how reasonable the definition of "rightsized" may be to a given user, different users have different needs; in fact, even a single customer may have different needs across their various resource groups within a service.

Consider two types of databases for a beverage company: trucking logs and a small marketing team's database. Faults in trucking logs may result in delayed shipments, meaning the company is willing to pay significantly more to ensure their trucking logs never experience any faults. Meanwhile, the small marketing team may never see heavy database traffic, and thus may prioritize a good deal with cheaper price.

Stage 3 of Lorentz treats the rightsizing recommendations from the Stage 1 algorithm as a *baseline recommendation*; that is, a typical customer may be fine with that fit of capacity to workload. We then use customer *cost/performance sensitivity scores*, denoted by λ, to further adjust the output of Stages 2. A performance-sensitive customer will receive a higher capacity than the Stage 2 recommendation, while a cost-sensitive customer will receive a smaller recommendation.

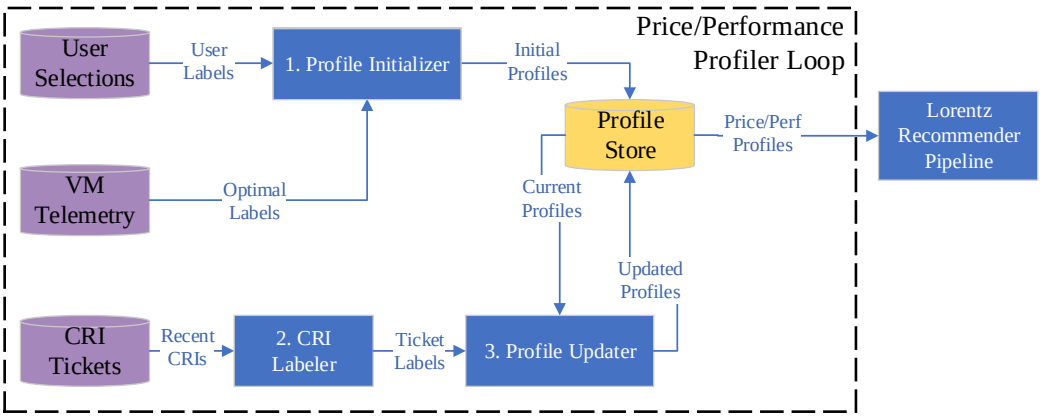


Fig. 6. The personalization loop completes Lorentz.

We develop the Lorentz's personalization algorithm to keep a profile of each individual user (or even finer granularity) about their sensitivity to price or performance. At initialization time, such profiles can be created based on their current selection of SKUs and the resource utilization level (see Figure 6). A profile store is thus created and maintained to store detailed information. Whenever a learning signal is triggered, such as a customer-reported incident (CRI) is issued, we will (1) determine if such a signal is related to the performance or cost of a customer, and (2) update the customer profile accordingly. Based on the constantly learned customer profile, the output recommendation from Stage 2 will be further adjusted to accommodate the different preferences of price versus performance accordingly. As the signal can be very sparse, we develop an innovative profile update algorithm similar to message propagation in social networks [25] to amplify the impacts of each received signal.

3.4.1 λ-adjusted Recommendations. Stage 3 uses the recommendations from Stage 2 along with a learned cost-performance sensitivity score of λ to compute adjusted scores in a two-stage approach. We remain aware of the log-transformation $\xi(\cdot)$ in this process. We get the adjusted capacity $\mathbf{c}^{**} = g(\mathbf{c}^*; \lambda)$ from the Stage 2's recommended capacity $\mathbf{c}^*$ as:

$$\mathbf{c}^{**} = g(\mathbf{c}^*; \lambda) = \xi^{-1}\Big(\xi(\mathbf{c}^*) + \lambda\Big). \tag{13}$$

In the case of $\xi = \log_b$, λ is easily interpreted as "How many powers of b should we increase or decrease the Stage 2's predictions by?":

$$\mathbf{c}^{**} = g(\mathbf{c}^*; \lambda) = b^{\left(\log_b(\mathbf{c}^*)+\lambda\right)} = b^{\lambda}\mathbf{c}^*. \tag{14}$$

The larger λ is the more costly offers will be recommended indicating that the user prefer a more performant service.

3.4.2 Learning customized λ. If a customer generates cost-sensitive signals (e.g., submits tickets to complain about cost or manually scale down resource capacity for their VMs), their cost-performance sensitivity score λ will decrease, adjusting future recommendations down. If they complain about performance and the mitigation is to provision a larger SKU, their λ will increase.

Structuring λ. We consider the hierarchy that can be learned from Stage 1 as the structure to compute the λ values, such as CloudCustomerGUID > SubscriptionID > ResourceGroup > Stratification for individual service or product. We can record the cost-perf sensitivity score λ at any of these granularities. Because predictions are mostly made at the Resource Group (RG) level in Stage 2, and because RGs are typically grouped with similar purposes (and thus often similar workload patterns), we will structure λ as a vector $\lambda_{C,S,R}$ whose dimension equals the number of possible stratification values and who is indexed by CustomerGUID C, subscription ID S, and RG R. Note that, each RG has its own vector of preferences across stratification variables; thus, if there are 3 stratification variables, there will be 3 sets of λ values in each RG. For new user profiles, λ defaults to 0.

Customer Satisfaction Signals. Stage 3 operates by adapting predictions to feedback on whether a given customer needs higher performance or lower costs. In order to capture this information, we broadly define customer satisfaction signals as any signal indicating a preference for more performant or lower-cost resources. These can include Customer Report Incidents (CRIs) with performance or cost as primary subjects, manual scale actions, or any other observable signal that can reliably indicate such a preference. We map these signals to the range $\gamma \in [-1, 1]$, where -1 indicates a strong cost sensitivity (prefer cheaper solutions), +1 indicates a strong performance sensitivity (prefer performant solutions), and anything in between indicates a lesser lean in that direction. We integrated a simple strategy for sentiment detection by checking if keywords such as “scale-up”, “increase CPU”, etc., exist in the ticket. Specifically, we focus on three ticket fields—“symptom", “title", and “resolution"—to formulate distinct sets of keywords for discerning the intent and resolution of the ticket. An example of such keywords for identifying complaints related to throttling can be found in Table 1. In the future, a more comprehensive analysis could be conducted, leveraging tools such as Large Language Models (LLM) or machine learning classifiers [40].

Table 1. Throttle Filters

Column	Keywords
Symptoms	[‘high cpu’, ‘high cpu usage’, ‘high cpu utilization’, ‘high cpu utilisation’]
Subject	[‘high cpu’, ‘high cpu usage’, ‘high cpu utilization’, ‘high cpu utilisation’]
Subject	[‘100%’, ‘99%’, ‘95%’, ‘90%’, ‘0%’, ‘9%’]
Resolution	[‘increas’]
Resolution	[‘throttl’]
Resolution	[‘scale up’, ‘scaling up’, ‘scaled up’]

Algorithm. As the satisfaction signals can be very sparse, in this paper, we develop a message propagation algorithm to capture the network impact and dynamically update the related profile across services/resource groups. Each time Lorentz receives a customer satisfaction signal, it propagates it into the associated customer profiles via weighted addition. For example, if a customer files a complaint about performance in a given resource group, the signal will still have a (reduced) impact on other resource groups across that customer’s subscriptions, but *will not impact scores for other customers.* Algorithm 1 details the process. The signal is first multiplied by a learning rate l_r, which controls the overall speed with which Lorentz reacts to signals. Relative impact in other buckets is controlled by multiplicative decay parameters ρ_R (decay across stratifications for different server offerings, e.g., Bustable, General Purpose), ρ_S (decay across Resource Groups) and ρ_C (decay across Subscriptions).

These decays can be set to control the impact of a customer satisfaction signal in one RG to λ-profiles in other RGs and across strata, or even across customers if needed. When signals are

Algorithm 1: Message Propagation Algorithm for Customer Sensitivity Score Update

Input: Satisfaction signal γ for customer CU, subscription SU, resource group RG, stratification ST, learning rate l_r, decay parameters ρ_R, ρ_S, ρ_C, initial sensitivity scores λ

for each signal γ **do**

Multiply signal by learning rate: $s = l_r \times \gamma$;

Step 1: Update score for the same resource group and stratification: $\lambda_{\text{RG,ST}} \leftarrow \lambda_{\text{RG,ST}} + s$;

Step 2: Decay signal and add to other stratification dimensions in the same resource group: $\delta = \rho_R \times s$, $\lambda_{\text{RG},x} \leftarrow \lambda_{\text{RG},x} + \delta, \forall x \in \text{RG} \setminus \text{ST}$;

Step 3: Accumulate updates in other resource groups in the same subscription: $\lambda_{x,y} \leftarrow \lambda_{x,y} + \rho_S \times \delta,\ \forall x \in \text{SU} \setminus \text{RG}, y \neq \text{ST}$; or $\lambda_{x,\text{ST}} \leftarrow \lambda_{x,\text{ST}} + \rho_S \times s,\ \forall x \in \text{SU} \setminus \text{RG}$;

Step 4: Add decayed updates to other subscriptions under the same customer: $\lambda_{x,y} \leftarrow \lambda_{x,y} + \rho_C \times \delta,\ \forall x \neq \text{SU}, y \neq \text{ST}$; or $\lambda_{x,\text{ST}} \leftarrow \lambda_{x,\text{ST}} + \rho_C \times s,\ \forall x \neq \text{SU}$;

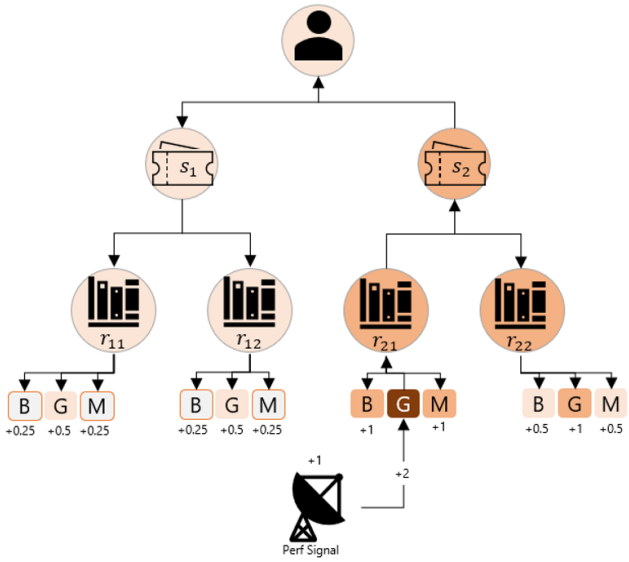


Fig. 7. Example of updating λ with exaggerated numbers: signal $\gamma = 1$, learning rate $l_r = 2$, and decays $\rho_R = 1/2$, $\rho_S = 1/2$, and $\rho_C = 1/4$.

rare, it can be beneficial to share these signals (Steps 3 and 4), as customers may have overarching preferences in addition to RG-specific preferences. However, as signals become more common, it may be preferable to set $\rho_S = 0$ to prevent sharing of signals across RGs, allowing better convergence of λ to the true preference in each RG. An example of the update computation is given in Figure 7. The customer has two subscriptions and each has two resource groups. Under each resource group, there is a λ-profile score for Burstable (B), General Purpose (G), and Memory-Optimized (M) service (see Section 2). When a signal is triggered for General Purpose database for resource group r_{21}, a chain-impact of updates will be made to all the other related services based on the defied decay rate l_r. The same propagation mechanism can be generalized to cross-customer learning where calibration for more propagation rates is required.

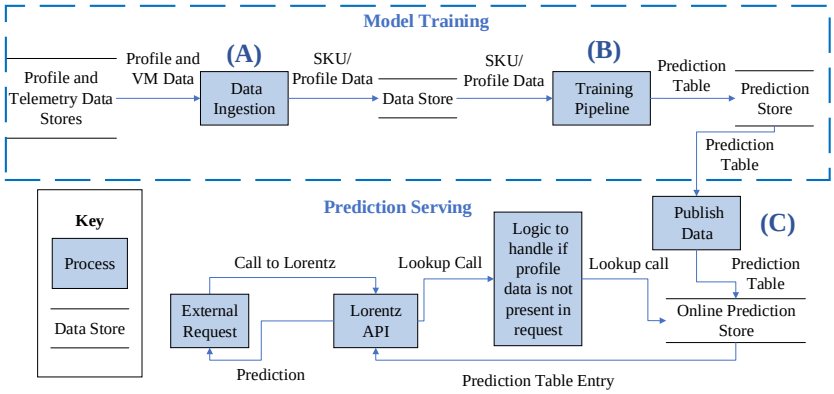


Fig. 8. Lorentz leverages an offline pipeline for frequent retraining and prediction pre-computation.

4 IMPLEMENTATION

To apply Lorentz to Azure PostgreSQL DB customer data in production, we prioritize low latency and compute price. We therefore perform daily batch inference offline to generate predictions for all customers with existing Azure PostgreSQL DB databases, using the following architecture (see Figure 8):

- Data Integration (A): Each batch of profile data and VM usage/health data is ingested via big data processing pipelines from profile data and telemetry data stores onto the compute resource used to train the model.
- Training Pipeline (B): We use this data as the input into our training and prediction pipeline. The pipeline executes data cleaning and validation, retrains Lorentz on the fresh data, and confirms that the new model's performance on a validation dataset is acceptable. The training and prediction pipeline also stores the trained model and its performance metrics for offline experimentation and model improvement. We then use the retrained Lorentz model to make predictions in batch, computing an SKU recommendation for each unique [hierarchy level, feature value, server offering] key represented in our dataset.
- Publish Data (C): We use a cloud-based ETL service to copy the new predictions to an online prediction store which provides low-latency lookup with built-in authentication and data versioning.

For inference, Lorentz receives SKU prediction requests from downstream services for all Azure PostgreSQL DB customers; each request contains the proposed instance's feature vector $\mathbf{x}$. From the online prediction store, we return the prediction corresponding to the most granular hierarchy level in $\mathbf{x}$ whose value is present in the store; if no values in $\mathbf{x}$ are present, we return a default prediction.

The final output presented to the user includes not only the recommended SKU but also the full rationale behind the choice for explainability. This includes information about both the DB instances identified as "similar" to this user's new request and those instances' chosen SKUs. We also provide the user's current value of cost-performance sensitivity score λ (i.e., their perceived preference on the performance-cost trade-off), allowing them to adjust this value to their liking.

Alternate implementations. It's important to address that in this design we prioritize low latency and compute price over other potentially desirable attributes requiring different architecture approaches. For example, if cloud operators instead prioritized prediction freshness or robustness to newly-created customer accounts, they could implement Lorentz using an online architecture

where the model is served via an inference endpoint, rather than pre-computing predictions on a schedule.

5 EXPERIMENTS

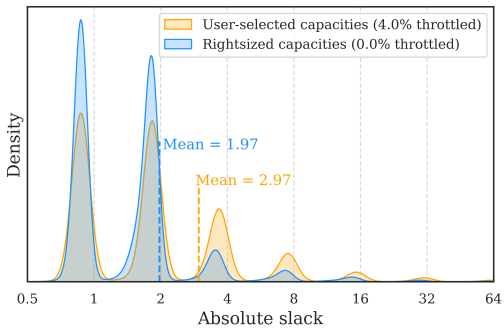


Fig. 9. Capacity rightsizing reduces slack and throttling over user selections based on empirical data.

This section describes the experiments we use to evaluate each stage of Lorentz. Table 2 describes the hyperparameters used for each stage's experiments.

Table 2. Hyperparameters

Model	Hyperparameters
Stage 1: Rightsizer	$T = 5$ min, $\eta = 0.95$, $s^*_{\text{CPU}} = 0.5$, $\tau = 0$, $K = 1$
Stage 2: Capacity recommenders	Train/val/test=80/10/10
Hierarchical model	$p = 50$, $\gamma = 0.6$
Target encoder	# trees = 100, ξ =$\log_2$
Stage 3: Personalizer	learning rate = 0.3, signal decay = 0.25

In the hierarchical model, $p = 50^{\text{th}}$ percentile serves as a balanced, outlier-robust choice. The hierarchy threshold γ is empirically selected to include only the observed group of strong hierarchies, and ξ =$\log_2$ maps the capacity options to linear space.

5.1 Stage 1: Rightsizing

The goal of the rightsizing stage is to compute new resource capacities that minimize slack and throttling for existing workloads given that many have been over/under-provisioned. In our Azure PostgreSQL DB dataset, we find significant disparities between user-selected and rightsized capacities; see Section 2.2 and Figures 1 & 2 for a complete summary.

We further evaluate the effectiveness of this stage on the Azure PostgreSQL DB workloads W by comparing the observed slack and throttling between user-selected capacities and our rightsized capacities, computed extremely conservatively using $s^* = 0.5$, i.e., 50% CPU utilization, and $\tau = 0$, i.e., 0% throttling limit as in Equation (7). Here, we use similar definitions for slack and throttling as in Stage 1. In Figure 9, we show the distribution of *absolute slack*, i.e., $S_{w,r}(c_r) \cdot c$, to better align with the business goal of minimizing the global resource volume provisioned, and compute each capacity set's *throttling ratio* as the ratio of workloads $\mathbf{w} \in W$ with $T_w(\mathbf{c}) > \tau$. That absolute slack weights higher-capacity servers higher is desirable in our business context, since resource costs generally scale linearly with capacity.

Figure 9 shows that our rightsized capacities eliminate throttling entirely, while still reducing slack by 34%. Note that absolute slack distributions are modal around powers of 2 because the candidate capacities are, for the most part, powers of 2.

The challenge, of course, is predicting these rightsized capacity labels before VMs are created; the following section describes Lorentz's solution to this task.

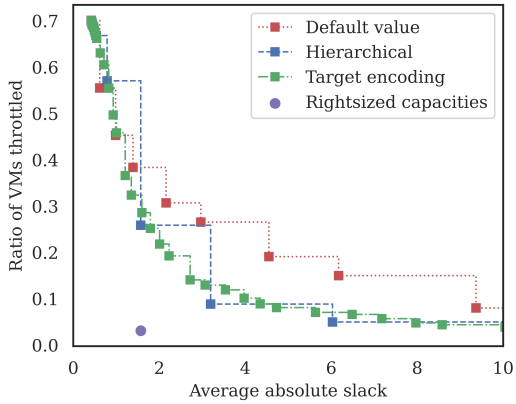


Fig. 10. Lorentz provisioners improve the slack/throttling Pareto frontier over baselines.

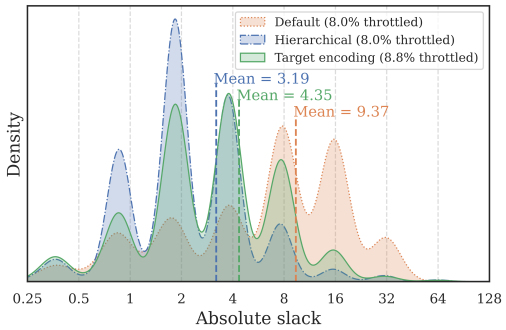


Fig. 11. Lorentz provisioners improve slack while keeping average throttling<10% over baseline.

5.2 Evaluating end-to-end provisioners

In this section, we evaluate the ability of Lorentz provisioners to recover the rightsized capacity labels following Stages 1 and 2, given useful features (computed only based on profile data) and diverse workloads. The dataset of Azure PostgreSQL DB resources is extremely left-skewed, containing many low-usage workloads: the mean maximum utilization is only 1.2 vCores. Further, our rightsizing algorithm selects the minimum capacity choice for 86% of DBs, and one of the two smallest choices 95% of the time. As a result, our provisioners achieve similar average slack to default-value baselines at most throttling ratios, with a maximum possible slack reduction of 33%. We also evaluated the provisioner models based on their aggregate vCores provisioned and hours throttled, extrapolated from the test set to a count of 67k servers (the total number of one DB offer for Azure PostgreSQL DB), achieving 27% (Hierarchical) and 8% (Target Encoding) reduction in cost compared to user selection.

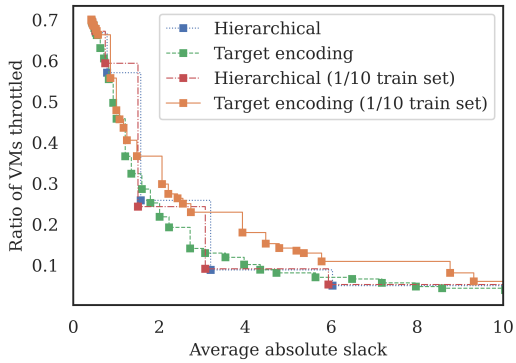


Fig. 12. The hierarchical provisioner is robust to data quantity.

Synthetic workload upscaling. To better demonstrate the effectiveness of our provisioner models, we therefore generate synthetic usage telemetry, increasing the diversity of our label set by scaling up some of the workloads according to their profile data.

To upscale the Azure PostgreSQL DB workloads, we first compute the hierarchy present in their profile features using Step 1 of the hierarchical provisioner, from most to least granular. We then upscale workloads with the following steps: (1) Select three features throughout the hierarchy, and assign them global scale factors–ResourceGroup: 1, CloudCustomerGuid: 1, VerticalName: 3; (2) Per feature, assign each unique value a scale factor: either that feature's global scale factor or 0, with equal likelihood; (3) Compute each workload's total scale factor χ_w by applying its feature values' assigned scale factors in combination, producing a result between 0 and $1 + 1 + 3 = 5$; (4) Upscale each workload as $2^{\chi_w} \cdot w[n]$; (5) Recompute the rightsized capacities. After upscaling, the dataset is much more diverse: the mean maximum utilization is 9.0 vCores and the rightsized capacity is one of the lowest two choices for only 55% of workloads.

Provisioner evaluation. We train the provisioner models on a 80%/10%/10% train/validation/test split using the hyperparameters described in Table 2. As described in Section 2.2, we fit distinct models to each stratified server offering; each of the stratified models uses the same hyperparameters, and all results are averaged across the offerings.

We evaluate the provisioner models on the upscaled workloads by computing their absolute slack and throttling ratio, as for Stage 1. Since users have variable sensitivities to slack and throttling, we cannot produce a single metric defining model performance; instead, we observe the Pareto frontiers of our models. We compute the Pareto curve for each model to evaluate the optimality of the recommended SKUs. For the hierarchical and target encoding provisioners, we build the Pareto curves by scaling all recommendations up and down by varying powers of two. Each scale factor produces one Pareto curve point: for example, scaling all predictions by $2^{-2.5}$ produces a low-slack, high-throttling point. We construct our default-value baseline by evaluating different sets of default values, where each set defines one default value for each of the three server offerings (e.g., Burstable, General-Purpose, MemoryOptimized). Note that since many users (89%) in the original observation select either the default vCore value or the next larger option, we consider this default baseline a reasonable approximation of user behavior after the scaling.

Figure 10 shows that the Pareto frontiers of both provisioner models notably exceed those of the baselines along both the slack and throttling axes using the synthetic workload. The hierarchical model performs particularly well, although its Pareto curve is coarser since its predictions are discrete. On the other hand, the target encoder exchanges a small retraction in performance for

incredible flexibility to be tuned along the slack/throttling spectrum. Target encoder yields detailed Pareto curves with ample data, while the hierarchical model delivers more robust performance in situations with limited data. For both methods, relying solely on profile data doesn't reach a perfect solution due to the variation of SKU choices within the same bucket. In this paper, we aimed to propose a general framework that can be extended with the availability of more data, potentially reducing the variability within each bucket.

We compare the single point on each model's Pareto curve as in Figure 10 that minimizes slack with a throttling ratio < 10%, a middle ground in the slack/throttling tradeoff. The probability density function plot in Figure 11 describes the models' slack distributions at these points; we again observe here that both provisioner models outperform the baselines. While achieving a similar level of throttling, the hierarchical provisioner reduces mean slack by 66%, while the target encoder provides a 54% reduction. We observe that workloads with greater variation are more throttling-prone, where cost improvements outweigh perfformance degradation. Lorentz also performs consistently across all capacity needs, skewed towards under-provisioning for large outliers (>32 vCores).

Smaller training set. To evaluate the provisioners' robustness to data quantity, we provide in Figure 12 their performance on a training set containing 10% of the samples in the full training set, randomly sampled, after the workload upscaling. We find that although the performance of the target encoder suffers with fewer training samples, the hierarchical model yields nearly equivalent performance despite the drastically smaller training set. Those deploying Lorentz should therefore select the most appropriate provisioner based on both the size of their training set and the flexibility required for their use case: if data is plentiful, take advantage of the target encoder's very granular Pareto curve, while if data is scarce, the hierarchical model will offer more robust performance.

The Lorentz provisioner models are effective despite being lightweight, and require relatively few features and parameters. This makes them especially feasible for deployment to realize even minor improvements over baselines. As a result, though, it's important that any VM profile data used as inputs is verified to be informative of their corresponding workloads.

5.3 Evaluating Stage 3: Personalization

While the evaluation of the personalization requires labled data (true sensitivity scores) and a longer time frame of observation for feedbacks, in order to test the convergence of our algorithm, we simulate the behavior of three customers assuming a ground truth of cost/performance sensitivity: Alice ($\lambda_A = 0$), Bob ($\lambda_B = 1.5$), and Charlie ($\lambda_C = -1.5$). Each of these customers has three subscriptions with their own associated cost/performance sensitivity, "Dev" ($\lambda_{\text{dev}} = -1$), "Prod1" ($\lambda_{\text{prod1}} = 0.5$), and "Prod2" ($\lambda_{\text{prod2}} = 1.5$). Thus, the true cost/performance sensitivity for Bob's Prod2 subscription is $\lambda_{B,\text{prod2}} = \lambda_B + \lambda_{\text{prod2}} = 3$, and so on. Each subscription has three resource groups, and within each resource group, we randomly provision $n_R \in \{1, ..., 5\}$ resources. For simplicity, we draw a random $\mathbf{c}^*$ from the set of C=$\{1, 2, 4, ..., 128\}$ representing the recommendation from Stage 2 for each resource. We assume that the error of such generated recommendation follows a log-normal distribution, $\log_2(\epsilon) \sim N(0, \sigma^2)$, and the optimal candidate (before customization) should be: $\bar{\mathbf{c}}^* = \mathbf{c}^* + \epsilon$. For each resource, considering the customer preference, the optimal capacity that the corresponding customer will be most satisfied with can be calculated as:

$$\bar{\mathbf{c}}^{**} = 2^{\lambda_{\{\text{A,B,C}\},\{\text{dev,prod1,prod2}\}}} \bar{\mathbf{c}}^* \tag{15}$$

For each resource, we initialize the personalization parameters $\widehat{\lambda_0} = 0$, making the initial SKU recommendation $\mathbf{c}_0^{**} = 2^{\widehat{\lambda_0}} \mathbf{c}^* = \mathbf{c}^*$.

We then simulate the profile learning from this initial data set in a three-step loop: (Step 1) Generate signals, (Step 2) Update profiles, (Step 3) Recompute predictions.

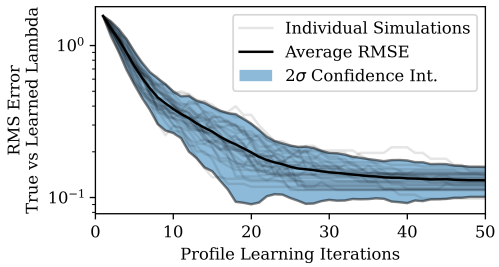


Fig. 13. Lorentz's Stage 3 converges rapidly in all simulations.

Step 1: Generate signals. Ideally, at time t, each customer gives us a signal for each poorly-provisioned resource: if the resource is over-provisioned ($\mathbf{c}_t^{**} > \bar{\mathbf{c}}^{**}$), we get a -1 signal, and if the resource is under-provisioned ($\mathbf{c}_t^{**} < \bar{\mathbf{c}}^{**}$), we get a $+1$ signal. We introduce two more parameters: signal rate (the probability that a signal is generated at all, simulating our ability to collect signals from customers) and signal noise (the probability that we collect an *incorrect* signal, flipping the sign). These two controls allow us to confirm the convergence of the profiles over time when signal collection is imperfect.

Step 2: Update Profiles. After signal generation, we update the stored profiles for each resource group using the algorithm outlined in Section 3.4.

Step 3: Recompute Predictions. The predictions at time $t > 0$ are given by $\mathbf{c}_t^{**} = 2^{\widehat{\lambda}_i}\mathbf{c}^*$, discretized to C. The difference between the personalized recommendation and the optimal capacity is:

$$\begin{aligned}\Delta &= \mathbf{c}_t^{**} - \bar{\mathbf{c}}^{**} \\ &= 2^{\widehat{\lambda}_t}\mathbf{c}^* - 2^{\lambda_{\{\text{A,B,C}\},\{\text{dev,prod1,prod2}\}}}\bar{\mathbf{c}}^* \\ &= 2^{\widehat{\lambda}_t}\mathbf{c}^* - 2^{\lambda_{\{\text{A,B,C}\},\{\text{dev,prod1,prod2}\}}}(\mathbf{c}^* + \epsilon) \\ &= (2^{\widehat{\lambda}_t} - 2^{\lambda_{\{\text{A,B,C}\},\{\text{dev,prod1,prod2}\}}})\mathbf{c}^* - M\epsilon, \end{aligned} \tag{16}$$

where M is a constant. When $|\lambda_{\{\text{A,B,C}\},\{\text{dev,prod1,prod2}\}} - \widehat{\lambda}_i|$ is small enough, the recommended capacity will be close to the optimal while only the error of Stage 2's prediction dominates.

5.3.1 Results. We run two main experiments using the above settings. First, we demonstrate that the Stage 3 algorithm converges reliably for a given setting. We set the signal rate to 40% (40% of misconfigured systems generate a signal at each iteration), the signal noise to 13% (13% of geenrated signals are multiplied by -1), the Stage 2 error $\sigma = 0.1$. In Figure 13, we plot individual simulations (gray lines), the average simulation (black line), and an estimated point-wise 95% confidence interval (2 standard errors). On average, customer profiles reach a root mean squared error (RMSE) of 0.15 within 30 iterations. Profile learning ceases once each customer's system is accurately provisioned.

Second, we evaluate the personalizer at each element in the Cartesian product of signal error values $\{0\%, 13\%, 26\%, 40\%\}$ and Stage 2 error (σ) values $\{0.0, 0.1, 0.25\}$. At each element, we run multiple simulations at signal rates of $\{10\%, 40\%, 70\%, 100\%\}$ respectively, averaging the convergence times over these experiments to produce one point in Figure 14. We define "convergence" here as the first iteration that the 80^{th} percentile of customer profiles have errors below 0.5, or $|\lambda_* - \widehat{\lambda}| \leq 0.5$. At this point, 80% of customers are receiving recommendations within half a step of their true preference, typically rounding to their preferred capacity. In general, these results indicate that (i) Lorentz rapidly personalizes results to individual customers if signals can be accurately classified and

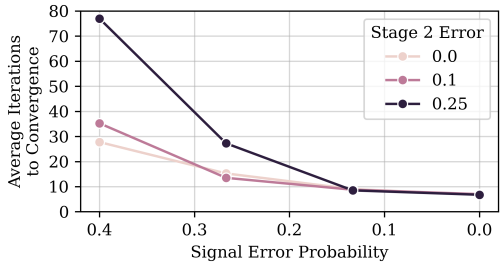


Fig. 14. Stage 3 converges rapidly when customer signals are accurate.

Stage 2 results are accurate, and (ii) accurate cost/performance signal classification can overcome both sparse or rare signals and inaccurate Stage 2 predictions.

6 RELATED WORK

SKU selection has perpetually posed a challenge. Prior studies have delved into situations involving workload migration, tapping into existing traces to aid in decision-making. Examples are Cloud Target Selection (CTS) [19] and Cloudle [16] where systematic processes on the foundation of exhaustive questionnaires are created, enabling users to outline their requirements. However, these tools primarily aimed at streamlining the decision-making process but were not able to fully automate the process. REMICS [30] undertook an in-depth exploration of prevailing workloads, encompassing source code, configuration files, and execution traces. This culminated in the development of a model-driven tool to facilitate migration. A recent addition to this domain, Doppler [2], delved into harnessing machine learning to assess the advantages of selecting optimal SKUs. This approach was grounded in historical resource consumption and insights gleaned from existing customers, factoring in preferences for price-performance trade-offs. However, these methodologies all share a common trait–the need for extensive input regarding the workload, rendering them impractical for our particular scenario.

Other related lines of research revolve around resource scaling after provisioning, encompassing both reactive scaling [8–10] and proactive scaling [5, 11, 15, 26, 27]. Techniques such as machine learning [3, 15, 18, 26–28, 35], signal processing [11], and statistical methods [15] are employed to predict future usage while accounting for performance constraints such as throttling impact and QoS requirements [9, 24], aiding in the determination of optimal resource allocation.

7 CONCLUSION

In this study, we introduced Lorentz, a system to generate SKU recommendations for *newly-provisioned* cloud resources without relying on telemetry data or workload traces. Lorentz employs a three-stage approach: capacity rightsizing, capacity recommendation, and personalization, allowing it to accommodate diverse user preferences for price and performance across various services. Through a continuous learning pipeline, the system dynamically adjusts preference scores based on user feedback signals, such as Customer Reported Incident (CRI) information. Evaluation on production data demonstrated that Lorentz eliminates >60% of wasted COGS compared to our baseline, indicating its effectiveness in optimizing cloud resource allocation.

Lorentz can be extended to suggest capacities for multiple resources, offering comprehensive SKU recommendations. Additionally, incorporating more entries of profile data features could potentially enhance the recommendation accuracy as well as enable recommendations of suitable server offerings among different types, adding substantial value.

REFERENCES

[1] Amazon.com, Inc. 2024. Amazon Web Service. Retrieved Jan 4, 2024 from https://aws.amazon.com/

[2] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. 2022. Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud. Proc. VLDB Endow. 15, 12 (aug 2022), 3509–3521. https://doi.org/10.14778/3554821.3554840

[3] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. 2014. Workload prediction using ARIMA model and its impact on cloud applications' QoS. IEEE transactions on cloud computing 3, 4 (2014), 449–458.

[4] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In VLDB '07. 3–14.

[5] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In Proceedings of the 26th Symposium on Operating Systems Principles. 153–167.

[6] 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 DEEM. 1–5.

[7] Robert Dale. 2021. GPT-3: What's it good for? Natural Language Engineering 27, 1 (2021), 113–118.

[8] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. 2016. Automated demand-driven resource scaling in relational database-as-a-service. In Proceedings of the 2016 International Conference on Management of Data. 1923–1934.

[9] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices 49, 4 (2014), 127–144.

[10] 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.

[11] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. Press: Predictive elastic resource scaling for cloud systems. In 2010 International Conference on Network and Service Management. Ieee, 9–16.

[12] Vladimir Gorodetsky, Vladimir Samoylov, and Olga Tushkanova. 2014. Agent-based customer profile learning in 3G recommending systems. In Proceedings of 9-th International Workshop on Agent and Data Mining Interaction (ADMI-2014) Associated with International Conference on Autonomous Agents and Multi-agent Systems (AAMAS-2014), Paris.

[13] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data? Advances in Neural Information Processing Systems 35 (2022), 507–520.

[14] Michael Hammer and Arvola Chan. 1976. Index Selection in a Self-Adaptive Data Base Management System. In SIGMOD. 1–8.

[15] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. 2012. Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems 28, 1 (2012), 155–162.

[16] Jaeyong Kang and Kwang Mong Sim. 2010. Cloudle: a multi-criteria cloud service search engine. In 2010 IEEE Asia-Pacific Services Computing Conference. IEEE, 339–346.

[17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. Advances in neural information processing systems 30 (2017).

[18] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In 2012 IEEE Network Operations and Management Symposium. IEEE, 1287–1294.

[19] Aliki Kopaneli, George Kousiouris, Gorka Echevarria Velez, Athanasia Evangelinou, and Theodora Varvarigou. 2015. A model driven approach for supporting the Cloud target selection process. Procedia Computer Science 68 (2015), 89–102.

[20] Matthew J Kushin and Kelin Kitchener. 2009. Getting political on social network sites: Exploring online political discourse on Facebook. First Monday (2009).

[21] Lotcher. Bowaer. 2024. Python implementation of HALO. Retrieved Jan 4, 2024 from https://github.com/lotcher/HALO

[22] Oracle. 2006. Oracle Database 10g Release 2: The Self-Managing Database. Technical Report. Oracle.

[23] Oracle. 2021. Oracle Database. https://www.oracle.com/database/. Accessed: 2021-11-03.

[24] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control of multiple virtualized resources. In Proceedings of the 4th ACM European conference on Computer systems. 13–26.

[25] Sasa Petrovic, Miles Osborne, and Victor Lavrenko. 2011. Rt to win! predicting message propagation in twitter. In Proceedings of the international AAAI conference on web and social media, Vol. 5. 586–589.

[26] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, et al. 2020. Seagull: An infrastructure for load prediction and optimized resource

allocation. arXiv preprint arXiv:2009.12922 (2020).
[27] 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. Proceedings of the VLDB Endowment 15, 6 (2022), 1279–1287.
[28] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In 2011 IEEE 4th International Conference on Cloud Computing. IEEE, 500–507.
[29] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems. 1–16.
[30] Andrey Sadovykh, Christian Hein, Brice Morin, Parastoo Mohagheghi, and Arne J Berre. 2011. REMICS-REuse and Migration of legacy applications to Interoperable Cloud Services. In European Conference on a Service-Based Internet. Springer, 315–316.
[31] Salesforce, Inc. 2023. CRM 101: What is CRM? Retrieved Dec 24, 2023 from https://www.salesforce.com/crm/what-is-crm/
[32] Mohanbir Sawhney, Birju Shah, Ryan Yu, Evgeny Rubtsov, and Pallavi Goodman. 2019. Uber: Applying Machine Learning to Improve the Customer Pickup Experience. Kellogg School of Management Cases (2019), 1–21.
[33] Scikit-learn. 2023. GradientBoostingRegression. Retrieved Oct 4, 2023 from https://scikit-learn.org/stable/auto_examples/ensemble/plot_gradient_boosting_regression.html
[34] Scikit-learn. 2023. RandomForestRegressor. Retrieved Oct 4, 2023 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
[35] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-store: An elastic database system with predictive provisioning. In Proceedings of the 2018 International Conference on Management of Data. 205–219.
[36] Wikipedia. 2023. Amazon Web Service. Retrieved Oct 4, 2023 from https://en.wikipedia.org/wiki/Homoscedasticity_and_heteroscedasticity
[37] Xu Zhang, Chao Du, Yifan Li, Yong Xu, Hongyu Zhang, Si Qin, Ze Li, Qingwei Lin, Yingnong Dang, Andrew Zhou, et al. 2021. Halo: Hierarchy-aware fault localization for cloud systems. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 3948–3958.
[38] 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.
[39] Yiwen Zhu, Yuanyuan Tian, Joyce Cahoon, Subru Krishnan, Ankita Agarwal, Rana Alotaibi, Jesús Camacho-Rodriguez, Bibin Chundatt, Andrew Chung, Niharika Dutta, Andrew Fogarty, Anja Gruenheid, Brandon Haynes, Matteo Interlandi, Minu Iyer, Nick Jurgens, Sumeet Khushalani, Brian Kroth, Manoj Kumar, Jyoti Leeka, Sergiy Matusevych, Minni Mittal, Andreas Mueller, Kartheek Muthyala, Harsha Nagulapalli, Yoonjae Park, Hiren Patel, Anna Pavlenko, Olga Poppe, Santhosh Ravindran, Karla Saur, Rathijit Sen, Steve Suh, Arijit Tarafdar, Kunal Waghray, Demin Wang, Carlo Curino, and Raghu Ramakrishnan. 2023. Towards Building Autonomous Data Services on Azure. In Companion of the 2023 International Conference on Management of Data (Seattle, WA, USA) (SIGMOD '23). Association for Computing Machinery, New York, NY, USA, 217–224. https://doi.org/10.1145/3555041.3589674
[40] Paolo Zicari, Gianluigi Folino, Massimo Guarascio, and Luigi Pontieri. 2021. Discovering accurate deep learning based predictive models for automatic customer support ticket classification. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (Virtual Event, Republic of Korea) (SAC '21). Association for Computing Machinery, New York, NY, USA, 1098–1101. https://doi.org/10.1145/3412841.3442109

Received October 2023; revised January 2024; accepted February 2024