MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud

Brian Kroth
bpkroth@microsoft.com

Sergiy Matusevych
sergiym@microsoft.com

Rana Alotaibi
ranaalotaibi@microsoft.com

Yiwen Zhu
yiwzh@microsoft.com

Anja Gruenheid
agruenheid@microsoft.com

Yuanyuan Tian
yuanyuantian@microsoft.com

ABSTRACT

This paper presents MLOS (ML Optimized Systems), a flexible framework that bridges the gap between benchmarking, experimentation, and optimization of software systems. It allows users to create one-click benchmarking and experimentation scenarios for multi-VM setups in the cloud with optional standard and custom metrics collection and data management of the results. MLOS provides a collection of pluggable optimizers (ML or otherwise) for efficiently exploring the configuration space and finding optimal values for parameters across the entire software stack, including VM, OS kernel, and userland applications. It has a convenient lightweight interface for data storage, access, and visualization for a user-friendly notebook experience. These features make it a useful platform for both systems developers and auto-tuning researchers. MLOS is an active open-source project and is being used within Azure Data. A video demonstrating MLOS is available at https://aka.ms/MLOS/VLDB-2024-demo-video.

PVLDB Reference Format:
Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Yuanyuan Tian, Anja Gruenheid. MLOS in Action. PVLDB, 17(12): 4269-4272, 2024.
doi:10.14778/3685800.3685852

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/microsoft/MLOS.

1 INTRODUCTION

The ever-increasing complexity of modern software stacks makes it hard, if not impossible, to tune all system components by hand. However, the ability to do so can have a significant impact on both performance and cost [3]. This challenge intensifies as the diversity of the cloud introduces a growing number of systems, workloads, and a heterogeneous mix of hardware [4]. For instance, Linux kernel alone has more than 1200 tunable parameters; there are over 300 configuration parameters in PostgreSQL and over 600 in MySQL, with these numbers growing with each release [18]. Given many different usage scenarios and workloads for (a combination of) these systems, optimizing their parameters specifically to a user's workload, while desirable both for customers and service providers, is a significant challenge. As a result, we witness a growing array of auto-tuning products like OtterTune [18] and DBTune [2], research efforts like DBBert [19] and GPTuner [14], as well as general optimization services such as Google Vizier [11] and Facebook Ax [15]. These frameworks leverage machine learning (ML) models to navigate the configuration space efficiently, relying on data points collected from often externally managed experimentation scripts or are tightly coupled to a specific domain like ML model training or online DB tuning. While these frameworks may allow users to feed basic performance values into their optimizer, they typically lack integration of standard repeatable offline benchmarking infrastructure provided by dedicated suites such as BenchBase [8], Dike [21], and LSTBench [6]. Moreover, they are often limited to tuning a small subset of systems targets.

We argue that a more generic infrastructure for systems benchmarking, optimization, *and* data management is essential for the following reasons: (a) aside from the classic developer-oriented scenarios, internal user studies indicate many customers and support teams are not yet comfortable with an *online* tuning approach (e.g., due to unpredictable behavior), so offline benchmarking of *similar* workloads and transferred to production continues to hold significant importance; (b) the ever-evolving and varied systems and their workloads supported in the cloud and limited engineering resources require more flexibility and automation in defining new benchmarking and customizing existing ones to represent real-world scenarios for services; (c) new approaches are needed for auto-tuning different software systems since despite sharing a similar overall architecture, each comes with its own peculiarities that require adjustments, and (d) data management of these experiments is often bespoke and can benefit from standard organization to facilitate reusable analysis and visualization across domains.

In this paper, we present MLOS [7], a flexible open-source framework that bridges this gap between *(a) experimentation, (b) benchmarking, (c) optimization, and associated (d) data management* of arbitrary systems software stacks. MLOS serves as a *flexible benchmarking automation tool* that supports easy-to-use *experimentation* with user-specified, tunable parameters that can generically span the entire software stack, including, but not limited to, VM, OS kernel, and userland applications. With pluggable optimizers (both ML and not), MLOS serves as a platform for researchers to explore new *auto-tuning optimization* and search strategies for different systems. Systems can be deployed to test these parameters by running benchmarks or other arbitrary scripts with its generic and modular automation framework. Finally, MLOS includes a *data management* component that helps track and analyze each aspect of the benchmark and optimization experiment. Although all MLOS

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685852

components work together by design, they can also be used in isolation or integrated with other software systems.

The ability to combine manual and automated experimentation with efficient auto-tuning optimization and easy-to-use data management and visualization of the results sets MLOS apart from other frameworks and makes it useful for many (intersecting) cohorts of users, most notably: (1) Software Engineers who look to vet or tune their code changes in a principled and automated way; (2) Platform Engineers who explore ways of configuring the deployment platform; and (3) Researchers to develop optimization, and auto-tuning strategies for improved performance and cost of operation.

In summary, MLOS makes the following contributions:

- An open-source framework that supports full-stack auto-tuning with easy-to-use benchmarking automation tools.
- A convenient platform to explore new auto-tuning algorithms and search strategies for various systems.
- A comprehensive data management component that facilitates tracking and analysis of experiments.
- Deployed in production, MLOS has been widely used to manage the launch of a large number of experiments to verify new setups, configurations, workloads, and hardware.

In this demo proposal, we will first showcase the need for cloud-centric experimentation and auto-tuning platform and give examples of real-life scenarios where MLOS has been successfully applied to optimize the performance of various software systems. Then, we will briefly describe the MLOS architecture and outline the user experience with the framework. Finally, we will describe our planned demonstration and review the related work.

2 MOTIVATION

Numerous research projects have demonstrated the benefits of auto-tuning for systems at cloud scale due their potential to improve application performance and thereby cost efficiencies [5, 23, 24]. However, current state-of-the-art auto-tuning software is not easily adaptable to leverage the existing standard or custom benchmarks, or apply to new tunable systems. Users then have to repeatedly write custom, often ad-hoc, scripts for parsing and managing the results and turn the benchmark data into machine- or human-consumable formats. Worse yet, neither benchmarking tools nor auto-tuners provide a convenient way to tweak the configuration parameters across the entire software stack (i.e., VM, OS, and application) and quickly experiment with new or modified setups. Such a holistic approach is essential. We have found, for instance, that co-tuning kernel and application parameters can yield higher performance improvements than either of them alone. We also find that different systems targets require different auto-tuning techniques (i.e., no silver bullets).

These considerations motivated us to develop MLOS, a framework that allows users to define and integrate new and existing benchmarks while exposing a set of tunable configuration parameters that can be adjusted for experimentation automation. The flexibility of the MLOS approach allows human experts and software optimizers (ML-based or otherwise) to work together to find the best configuration and best search algorithm for a given workload and easily validate new configurations in various system environments.

2.1 Use Cases

We now outline just a few use cases of MLOS within Microsoft.

2.1.1 *Linux kernel tuning for Redis.*

Our studies of Azure usage show that there is typically a single main application per VM. Moreover, while customers may spend significant time and effort tuning the application parameters, they often neglect the OS kernel configuration. Linux kernel has over 1200 tunable parameters that almost always stay at their default, general purpose, values for the lifetime of the VM. Internal research [5] has shown that tuning even a small number of kernel parameters can significantly improve the application's performance. For instance we used MLOS to benchmark and optimize Redis on a Linux VM on Azure. We find that after just a few hours of trials, it can produce a new Linux kernel configuration that decreases the P95 tail latency of Redis' GET requests by 68%.

2.1.2 *MySQL optimization on Azure.*

We have also looked at optimizing application services, e.g., MySQL. Specifically, we were able to reuse many MLOS configuration components from the Redis optimization scenario, including Azure VM provisioning, remote VM script execution, and Linux kernel configuration, to optimize the MySQL InnoDB parameter for Azure MySQL Flexible Server. Our preliminary results for this setup show a 49% decrease in P95 query tail latency and a 51% increase in throughput for TPCC.

Note that the high variance in some of these initial results makes it more difficult for ML optimizers to learn an accurate signal from the data. However, with MLOS's flexibility, we can explore alternative trial scheduling and search policies for tuning for *robust* configs [10], and use the insights to improve the baseline service noise.

Currently, MLOS is part of the daily workflow of the MySQL production team, which uses the tool for benchmarking as well as optimization by running a large number of optimization experiments for a variety of workloads and VM configurations.

A similar effort has also begun with Azure PostgreSQL.

2.1.3 *Cost constants tuning for SQL Server Query Optimizer.*

Like many other Query Optimizers (QO), SQL Server QO has a number of predefined, hard-coded values to estimate operation costs such as predicate evaluation, HashJoin, and operator exchange. Our experiments show that the *one-size-fits-all* nature of hard-coded values can lead to sub-optimal performance, particularly in diverse query and hardware environments. We utilize MLOS to re-calibrate these constants, tailoring them to the specific characteristics of the workload and hardware. Initial results show that tuned values can lead to 2.6x speedup for a subset of TPC-H queries thru improved plans, and even more for internal workloads.

3 MLOS OVERVIEW

We have designed MLOS to be modular and extensible, with a focus on flexibility and automation. All MLOS components are pluggable; the user can replace any with provided alternatives (e.g., Grid Search vs. SMAC BO) or even custom implementations adhering to the core APIs (e.g., `suggest`, `register`, etc.). This architecture makes the framework applicable to many scenarios, from simple application benchmarking to complex full stack multi-VM optimization, and allows users and researchers to reuse and compose configs for new use cases to try different setups and optimization algorithms easily.

3.1 MLOS Architecture

MLOS architecture (Figure 1) consists of three main components: a) *Optimizer* module suggests new values for the various *tunable parameters* - a description of the configuration space provided by the user including hints on sampling distribution, quantization, etc.; b) *Scheduler* module assigns the config to one or more *trial workers*, which, optionally acting in parallel, plug the suggested values into the configuration templates and, using a combination of built-in and user-provided scripts and MLOS configuration files, sets up and runs the experiments, deploying VMs if necessary, and collects the results; and c) *Storage* module preserves the exercised configurations and the results of the experiments. An optional *Visualization* module provides a collection of stock visualization and analysis routines for the data collected. MLOS connects these components in a main optimization cycle that runs repeatedly until it meets a specified stopping criterion. The collected data can be reused to inform the optimizer's choices in compatible experiments, i.e., to warm up the optimizer after restarts or transfer learn. Users can also replace or turn off the optimizer as needed. In those cases, the scheduler can still load explicit tunable values from a JSON file or the storage module, thus working as a benchmarking and manual exploration framework. For example, some of our customers initially use MLOS only for benchmarking, gradually enabling the optimizer to experiment with different system configurations offline while they gain confidence in the system and the values the optimizer produces.

MLOS Environments. An *Environment* is a key abstraction used within the MLOS scheduler. It encapsulates a benchmark setup's logic, configuration, and runtime state. A benchmarking environment has three phases: *setup* creates the system-under-test (SUT), setting the tunable parameters' values as part of the configurations to instantiate the system stack; *run* executes the benchmark and collects its results; *teardown* cleans up the resources acquired.

As shown in Figure 2, environments can be stacked to chain up the setup, run, and teardown phases. This allows users to compose smaller reusable environment configurations into complex benchmarking scenarios. MLOS has a growing open-source library of such configuration files available in the GitHub repository. Such reusable configuration modules not only lower the barrier to entry, but also allow users to continually benefit from additional improvements made by others, rather than each implementing their own proprietary non-reusable and non-repeatable systems.

MLOS Services. Another key abstraction in MLOS configs is *Services*, which implement pluggable functionality required by different Environments in the stack (e.g., deploy resource, remote exec, etc.). Services allow MLOS users to reuse portions of the configurations across different Environments and experiments. As a result, MLOS users can quickly reconfigure their experiments from one cloud provider to another, run on CloudLab [9], or locally via SSH.

4 DEMONSTRATION

Although MLOS can be used on cloud resources, due to time constraints and setup simplicity, in this demo we use MLOS to tune a local self-contained SQLite [1] example DB in conjunction with BenchBase [8], a SQL benchmarking framework. We have created a DevContainer with the dependencies preinstalled and several simple scripts to setup and configure the database, benchmark it, and parse the results. We also describe the SQLite tunable parameters and define the benchmark orchestration using simple JSON configs for MLOS. Working examples of the config files and scripts are available at https://github.com/Microsoft-CISL/sqlite-autotuning.

4.1 Setup

Workload. We use the TPC-C benchmark to measure OLTP system performance.

Tunable Parameters. In this demo, the tuning process focuses on 9 parameters, each contributing to the database's performance and reliability for OLTP workloads, and resulting in a config space size of $O(10^{20})$ combinations. In other scenarios we see even larger numbers, though intentionally limited it for a short demo. Examples for the tunable parameters used in SQLite (amongst others) are:

Name	Values/Range	Short Description
synchronous	{'off', 'normal', 'full', 'extra'}	Disk synchronization mode
locking-mode	{'normal','exclusive'}	Database file locking strategies
cache-size	[1, 2147483647]	Cache size
temp-store	{'default', 'file', 'memory'}	Storage mode of temporary database files

4.2 User Interaction

Demo attendees will interact with the demo as follows:

MLOS as a One-shot Benchmark. We start the demo with MLOS executing a single TPC-C benchmark against a sqlite DB and recording the results in a local DB in order to expose users to the MLOS configs and CLI. Users will examine the MLOS configs that control the tunables parameters, experiment name, environment setup, etc. The entire process is flexible by design, allowing demo attendees to provide different configs, thereby customizing the experiment according to user interactions and acclimating to the system.

MLOS as an Optimization Loop. In this scenario, users will use the MLOS tool to run an optimization cycle. Before they start tuning a setup, they can (*i*) decide how many times the optimization loop will run, (*ii*) choose which optimizer to use for tuning (e.g., FLAML, SMAC, GridSearch) and optional space adapter (e.g., LlamaTune [13]), (*iii*) specify the optimization objective metric (e.g., throughput or P99 latency) and (*iv*) the preferred optimization direction such as minimizing or maximizing the chosen objective. The optimization loop runs several trials, each time updating the tunable values based on the results of the preceding one. Figure 3 (lower left) shows a snippet of the configuration file needed for the optimization phase.

Analyzing the Results. The users will run the provided Jupyter starter notebook and MLOS *visualization* module to analyze the data obtained from running the experiment. This analysis will help the users to identify a better SQLite configuration and to gain insights into the optimizer's findings regarding the performance of that configuration. Figure 3 shows some of the results analysis APIs.

5 RELATED WORK

Machine learning has been widely used for parameter optimization [12, 17]. For database management, OtterTune [3, 20] uses machine learning models to identify key configuration parameters for tuning based on workload analysis and is offered as a service [18]. Similarly, DBTune [2] offers continuous tuning services for databases like PostgreSQL, MySQL, RocksDB, and FoundationDB. CBDTune [22] employs deep reinforcement learning with

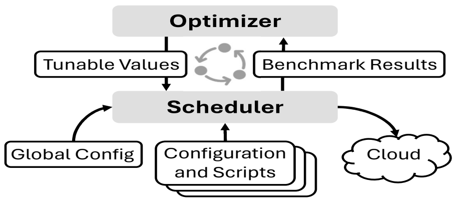


Figure 1: MLOS Architecture

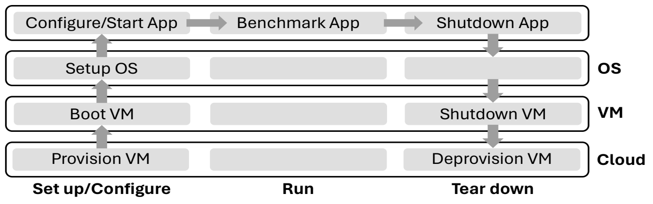
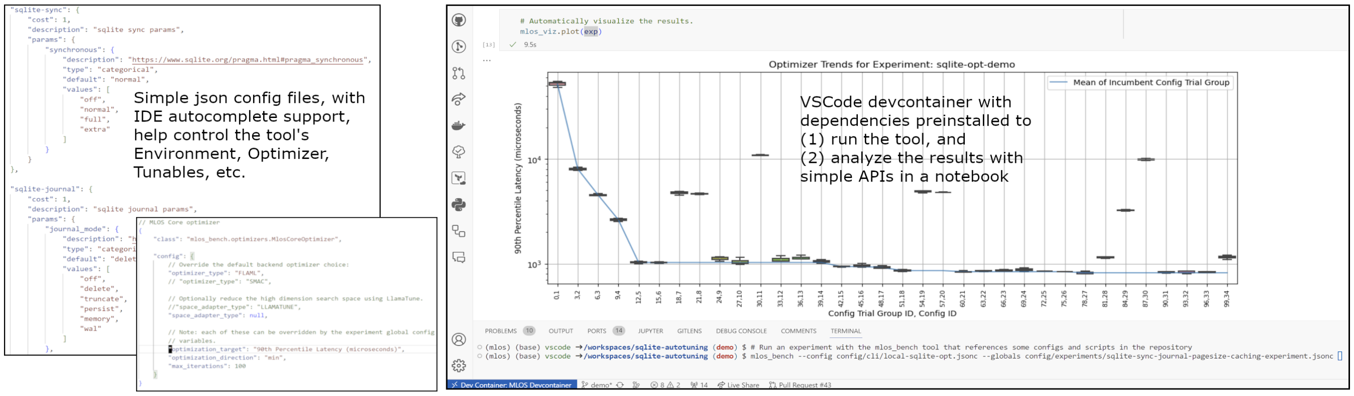


Figure 2: Stacking MLOS Environments



```
"sqlite-sync": {
    "cost": 1,
    "description": "sqlite sync params",
    "params": {
        "synchronous": {
            "description": "https://www.sqlite.org/pragma.html#pragma_synchronous",
            "type": "categorical",
            "default": "normal",
            "values": [
                "off",
                "normal",
                "full",
                "extra"
            ]
        }
    }
},

"sqlite-journal": {
    "cost": 1,
    "description": "sqlite journal params",
    "params": {
        "journal_mode": {
            "description": "h
            "type": "categori
            "default": "delet
            "values": [
                "off",
                "delete",
                "truncate",
                "persist",
                "memory",
                "wal"
            ]
        },
```

Simple json config files, with IDE autocomplete support, help control the tool's Environment, Optimizer, Tunables, etc.

```
// MLOS Core optimizer
{
    "class": "mlos_bench.optimizers.MlosCoreOptimizer",

    "config": {
        // Override the default backend optimizer choice:
        "optimizer_type": "FLAML",
        // "optimizer_type": "SMAC",

        // Optionally reduce the high dimension search space using LlamaTune.
        //"space_adapter_type": "LLAMATUNE",
        "space_adapter_type": null,

        // Note: each of these can be overridden by the experiment global config
        // variables.
        "optimization_target": "90th Percentile Latency (microseconds)",
        "optimization_direction": "min",
        "max_iterations": 100
    }
}
```

```
# Automatically visualize the results.
mlos_viz.plot(exp)
```

VSCode devcontainer with dependencies preinstalled to (1) run the tool, and (2) analyze the results with simple APIs in a notebook

```
○ (mlos) (base) vscode →/workspaces/sqlite-autotuning (demo) $ # Run an experiment with the mlos_bench tool that references some configs and scripts in the repository
○ (mlos) (base) vscode →/workspaces/sqlite-autotuning (demo) $ mlos_bench --config config/cli/local-sqlite-opt.jsonc --globals config/experiments/sqlite-sync-journal-pagesize-caching-experiment.jsonc
```

Figure 3: The MLOS demo interface: JSON config files (left) help control the command line tool running in a terminal (lower right) of the VSCode devcontainer (right) while simple MLOS APIs are used to visualize results in a Jupyter notebook.

policy gradient to recommend optimal configurations. Recently, GPTuner [14] integrates domain knowledge with LLM to refine the tuning domain and initial configs to improve the knob selection per workload. Due to its pluggable APIs it is possible for MLOS to incorporate LLMs for this purpose. General black-box optimization services like Google Vizier [11] are implemented as a managed service with an RPC API, relying on a centralized persistent database for optimization status. Ax [16] provides a simple Python API and employs an optimizer suitable for noisy experiments. Using the flexibility of our framework, we also have work investigating trial scheduling policies for improved noise handling. However, most of the existing tools are too infrastructure heavy for our purpose or hard to integrate (e.g., Ax, Vizier) or run a very tight online optimization loop that makes safe exploratory or manual experimentation hard (e.g., DBTune, OtterTune). Benchmark suites such as BenchBase [8], LSTBench [6] and Dike [21] provide flexibility in defining new and customizing existing benchmarks to represent the users' workloads. In this work, MLOS aims at closing this gap by providing a flexible framework for benchmarking and experimentation in the cloud while also serving as an auto-tuning platform, leveraging state-of-the-art optimizers and providing a platform to continue their development for systems tuning applications.

REFERENCES

[1] 2024. *SQLite*. Retrieved Jan 22, 2024 from https://www.sqlite.org
[2] DBT Solutions AB. 2024. *DBTune*. https://www.dbtune.com
[3] Van Aken et al. 2017. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*. 1009–1024.
[4] Van Aken et al. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.
[5] Anand Bonde. 2022. Kernel Parameter tuning for Memcached. *GitHub/preprint* (2022). https://github.com/anandbonde/anandbonde.github.io/blob/main/docs/kernel_parameter_tuning_for_memcached.pdf
[6] Jesús Camacho-Rodríguez et al. 2023. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *arXiv preprint arXiv:2305.01120* (2023).
[7] Carlo Curino et al. 2020. MLOS: An Infrastructure for Automated Software Performance Engineering *(DEEM'20)*. Article 3, 5 pages.
[8] Djellel Eddine Difallah et al. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.
[9] Dmitry Duplyakin et al. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14.
[10] Johannes Freischuetz et al. 2023. Performance Roulette: How Cloud Weather Affects ML-Based System Optimization. In *ML for Systems Workshop at NeurIPS*.
[11] Daniel Golovin et al. 2017. Google Vizier: A Service for Black-Box Optimization *(KDD '17)*. 1487–1495.
[12] Xu Huang et al. 2022. A Novel Reinforcement Learning Approach for Spark Configuration Parameter Optimization. *Sensors* 22, 15 (2022), 5930.
[13] Konstantinos Kanellis et al. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *arXiv preprint arXiv:2203.05128* (2022).
[14] Jiale Lao et al. 2023. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *arXiv preprint arXiv:2311.03157* (2023).
[15] Benjamin Letham et al. 2019. Constrained Bayesian Optimization with Noisy Experiments. *Bayesian Analysis* 14, 2 (2019), 495 – 519.
[16] Meta Platforms, Inc. 2024. *Ax: Adaptive Experimentation Platform*. https://ax.dev
[17] Van Otterlo et al. 2012. Reinforcement learning and Markov decision processes. In *Reinforcement learning: State-of-the-art*. 3–42.
[18] OtterTune. 2024. *OtterTune*. Retrieved Jan 2, 2024 from https://ottertune.com
[19] Immanuel Trummer. 2022. Demonstrating DB-BERT: A Database Tuning Tool that" Reads" the Manual. In *Proceedings of the 2022 International Conference on Management of Data*. 2437–2440.
[20] Bohan Zhang et al. 2018. A demonstration of the OtterTune automatic database management system tuning service. *PVLDB* 11, 12 (2018), 1910–1913.
[21] Huidong Zhang et al. 2023. Dike: A Benchmark Suite for Distributed Transactional Databases. In *Companion of the 2023 International Conference on Management of Data (SIGMOD '23)*. 95–98.
[22] Ji Zhang et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning *(SIGMOD '19)*. 415–432.
[23] Yiwen Zhu et al. 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *Proceedings of the 2021 International Conference on Management of Data*.
[24] Yiwen Zhu et al. 2023. Towards Building Autonomous Data Services on Azure. In *Companion of the 2023 International Conference on Management of Data*. 217–224.