Performance Characterization of Data Store Event Trigger Mechanisms for Serverless Computing

Ritul Satish, Sacheendra Talluri, Sudarsan Sivakumar, Matthijs Jansen, Alexandru Iosup
Vrije Universiteit Amsterdam
The Netherlands
ritul.satish@student.vu.nl, s.talluri@vu.nl

***Abstract*—Serverless applications are composed of functions triggered by events. Data stores are a common source of event triggers in the cloud, even beyond serverless, such as in Kubernetes. We find trigger latency, the time from event generation to function invocation, to take up to 62% of execution time for common serverless applications. Even though event triggers play a crucial role in serverless performance, the mechanisms driving these triggers are ill-understood. In this paper, we analyze data store trigger mechanisms, define the features that make up these mechanisms, and characterize their performance with TriggerPerf, a benchmarking tool for data store triggers. We implement TriggerPerf on three AWS data stores with built-in trigger support: S3, DynamoDB, and AuroraDB. With TriggerPerf, we demonstrate significant latency, scalability, and elasticity bottlenecks across these data stores. We observe that the trigger latency of AWS data stores is up to 100x higher compared to a reference etcd data store. Moreover, the median tail latency of S3 and AuroraDB is 10x higher when under high load, unlike DynamoDB. The observed variability in performance patterns significantly impacts the reliability of serverless and distributed systems that depend on them, highlighting the critical need for further research into the underlying mechanisms. The tool is open-sourced and is available at https://github.com/atlarge-research/trigger-perf.**

***Index Terms*—serverless, triggers, data store, performance**

I. Introduction

Serverless computing is ubiquitous in the cloud computing landscape, driven by the rise of Function-as-a-Service (FaaS) platforms [1]. These platforms abstract server management entirely, enabling developers to compose applications from stateless functions triggered by external events such as HTTP requests, timers, or updates to cloud data stores. Data store events play a pivotal role among these triggers, as they enable seamless integration between cloud storage and compute. For example, when a data object is created or updated, the corresponding event can invoke serverless functions to process the changes (Figure 1). Such triggers account for 6.8% of all events in the Azure Functions workload trace [2], underscoring their importance. The latency of data store triggers can significantly impact the overall responsiveness of serverless applications. As serverless functions achieve millisecond level execution times [2], the delay introduced by event triggers becomes a critical bottleneck. To demonstrate this impact, we gather serverless applications with data store triggers [3], [4] and infer the trigger latency's share in event response time (Figure 2). We define Trigger latency as the time delay between the occurrence of an event in a data store and

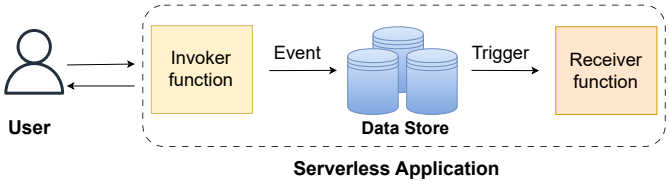

Fig. 1. Data store trigger execution flow.

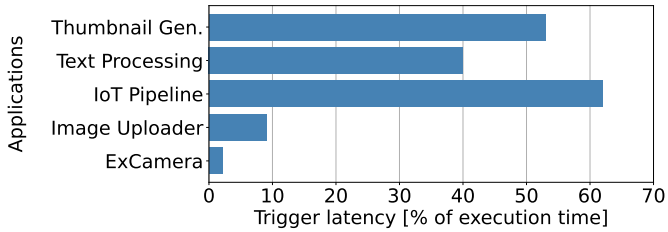

Fig. 2. Share of data store trigger latency in total execution time for serverless applications.

the invocation of a corresponding serverless function. We observed that trigger latency accounts for anywhere from 2% to 62% of total execution time between use cases, showing the data store trigger's profound impact on serverless application performance.

Despite their critical role in serverless ecosystems, the mechanisms and performance of data store triggers remain poorly understood. While prior work has benchmarked serverless platforms [5], [6], studied workflow platforms [7], and compared trigger types [8], these efforts have largely overlooked the detailed analysis of data store trigger mechanisms. Furthermore, cloud data stores have limited documentation, limiting research to black-box evaluations. Cloud data store triggers also have limited configurability: The AWS DynamoDB data store [9], for example, polls for trigger events only four times per second, which burdens dependent serverless applications with an unavoidable and significant overhead.

This lack of transparency and configurability motivates the need for a deeper analysis of data store triggers.

In this work, we analyze data store event trigger mechanisms and characterize the performance of serverless datastore triggers in the AWS ecosystem. We select three widely used serverless-compatible data stores: S3, DynamoDB, and AuroraDB. To broaden our analysis, we include etcd, a widely used, non-serverless distributed key-value store to serve as a baseline. Based on this implementation, we expand our

TABLE I
DATA STORE TRIGGER MECHANISMS OVERVIEW.

Data Store	Built-in Trigger	Data Model	Trigger Event Contents	Supported Trigger Events
Etcd	Watch	Key Value Store	Updated Key-Value, Event type, Key version	Put, Update, Delete
S3	Event Notifications	Object Storage	Updated Key, Event type, Source name	Put, Update, Delete
DynamoDB	DynamoDB Streams	NoSQL DB	Old & Updated key-values, Key Version	PUD, TTL, Table Create/Delete
AuroraDB	SQL UDF	Relational DB	Modifiable within UDF	SQL Filters

analysis of trigger mechanisms to include scalability (*what is the latency of concurrent triggers?*) and elasticity (*how does scalability change with varying concurrency levels?*).These evaluations are essential to ensure that triggers maintain low latency and adapt efficiently under real-world workloads.

We make the following contributions:

1) **Qualitative Analysis:** We analyze the data store trigger mechanisms of AWS S3, DynamoDB, and AuroraDB, characterizing them across five key dimensions. We also compare them to etcd to highlight differences between serverless and non-serverless systems (Section II).
2) **Microbenchmarking tool:** We design and implement *TriggerPerf*, a specialized benchmarking framework designed to evaluate the performance of data store triggers across latency, scalability and elasticity dimensions (Section III and Section IV). The tool is open-sourced and is available at https://github.com/atlarge-research/trigger-perf.
3) **Performance Characterization:** Using *TriggerPerf*, we systematically evaluate the latency, scalability, and elasticity of data store triggers. Our results reveal key bottlenecks and trade-offs in the design of serverless-compatible data stores and highlight the performance gaps relative to non-serverless systems like etcd (Section VI).

II. QUALITATIVE ANALYSIS OF DATA STORE TRIGGER SYSTEMS

To understand and compare data store trigger mechanisms, we perform a qualitative analysis of triggers in the AWS data stores S3, DynamoDB and AuroraDB. To guide our comparison, we define five key feature categories and organise them into two distinct phases: setup *(configuring when and how triggers are launched)* and activation *(managing how triggered events are delivered and consumed)*. For context To reason about data store trigger mechanisms and compare existing implementations, we qualitatively analyze data store triggers in the AWS data stores S3, DynamoDB, and AuroraDB and define five feature categories to compare them in. We present our qualitative comparison of these data stores with the non-serverless trigger providing data store etcd in Table I.

*a) **Setup:**:* The setup phase focuses on the configurations that determine when a trigger should launch and how the associated trigger mechanism operates. We categorize the setup phase configurations into the following categories:

1) **Event selection:** This category defines the granularity of events that can initiate a trigger. Common events include the creation, update, or deletion of data objects. AuroraDB offers a high degree of flexibility by supporting custom trigger rules defined using User-Defined Functions (UDFs) in SQL, enabling fine-grained event selection. In contrast, S3 supports only basic events, such as create, update, and delete, with no options for further customization.
2) **Filtering:** Filtering allows users to apply triggers selectively to specific data objects or groups of objects. For instance, S3 and etcd support filtering based on key prefix or suffix rules, enabling targeted triggers. This feature is particularly useful for reducing unnecessary trigger invocations and improving system efficiency.
3) **Execution model:** Most data stores offer asynchronous triggers to optimise for low latency. This helps reduce latency but introduces complex error handling to prevent cascading downstream failures.

*b) **Activation:**:* The activation phase governs how triggers are activated and how their events are sent to downstream systems. We categorize the activation phase characteristics into the following categories.

4) **Event content:** The payload sent to the destination varies significantly between data stores. For example, S3 provides the updated key and event type, while DynamoDB Streams include detailed snapshots of both old and new key-value pairs. These differences directly affect how downstream applications can process and respond to events. We list these contents in Table I.
5) **Destination:** The options for routing triggered events to downstream services are often constrained by the data store. AuroraDB's triggers, for example, can only route events to AWS Lambda functions, limiting integration flexibility. In contrast, S3 and DynamoDB offer broader destination support, enabling integration with multiple serverless frameworks and external services.

A. DynamoDB Streams Feature Mapping

To verify our qualitative analysis, we examine DynamoDB's trigger mechanism (DynamoDB Streams) and map it to our five feature categories. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores them as records in its logs.

In the setup phase, it supports a wide range of event types, including item creation, updates, deletions, Time-to-Live (TTL) expirations, and table-level events such as table creation or deletion. Filtering is supported at the payload level, allowing triggers to act on specific attributes or conditions

TABLE II
MAPPING OF DYNAMODB STREAMS TO FEATURE CATEGORIES

Category	DynamoDB Streams Characteristics
Event Selection	Supports creation, updates, deletions, TTL expirations, and table-level events.
Filtering	Allows payload-based filtering for fine-grained control.
Execution Model	Asynchronous, responding to writes before firing triggers.
Event Content	Customizable payloads with old/new item snapshots.
Destination	Routes events to AWS Lambda and other consumers.

within the event data, thereby reducing unnecessary event handling and improving efficiency. The execution model is asynchronous by default, ensuring that triggers respond to the originating write operation with minimal latency before firing the event.

In the activation phase, DynamoDB Streams offers customizable payloads, allowing developers to include snapshots of both the old and new values of modified items. This adaptability ensures that triggers can be tailored to specific use cases. Triggered events can be routed to various consumers, including AWS Lambda functions, enabling seamless integration with serverless frameworks. This flexibility supports diverse processing patterns, from real-time event handling to batch-oriented workflows. Table II shows this mapping.

B. etcd as a Reference

We use etcd, a widely used non-serverless data store with trigger mechanisms, as a reference system to compare against AWS' serverless data stores. etcd, along with its trigger capabilities, is used by complex software such as Kubernetes [10], Vitess [11], and OpenStack [12]. etcd is a distributed key-value store for coordination and state management in distributed systems. It is an integral component of Kubernetes, as it stores critical cluster configuration data [13]. The Watch mechanism in etcd provides its trigger functionality, enabling applications to monitor changes to specific keys or key prefixes dynamically. This capability supports responsive and adaptive behavior in systems that depend on etcd for real-time configuration and state updates. As a non-serverless trigger system, etcd establishes a latency baseline for serverless alternatives, setting the lower bound they must achieve to compete with traditional, performance-optimized solutions.

III. TRIGGERPERF: HIGH-LEVEL DESIGN

To experimentally characterize the performance of data store trigger systems, we design and implement TriggerPerf, a custom benchmarking tool tailored for serverless environments. The core objective of TriggerPerf is to provide precise, reproducible, and scalable evaluations of data store trigger latency and performance. By establishing a controlled sandbox environment, TriggerPerf facilitates interactions with selected data stores as the system under test (SUT), enabling the measurement of critical performance metrics. TriggerPerf focuses on three major performance metrics:

1) Unloaded Latency: Measures trigger latency under minimal load, providing a baseline for comparison.
2) Latency Under Load: Evaluates trigger behavior during high-throughput scenarios, identifying bottlenecks and tail latency (e.g., p90).
3) Latency on Elastic Scaling: Assesses the system's ability to maintain performance when scaling dynamically to handle increased workloads.

These metrics are evaluated across three AWS serverless data stores, S3, DynamoDB, and AuroraDB and compared to etcd, a non-serverless distributed key-value store, which serves as a baseline reference.

A. Design Principles

TriggerPerf is guided by the following key principles:

1) **Modularity [P1]:** TriggerPerf's architecture is designed to be modular, with independent components handling setup, event generation, and trigger handling. This modularity allows developers to integrate new data stores or modify configurations with minimal effort. This modular design is critical in the rapidly evolving landscape of cloud computing, where trigger mechanisms are continually advancing.
2) **Reproducibility [P2]:** Reproducibility is fundamental to TriggerPerf's design, ensuring consistent results across multiple experiment runs. TriggerPerf introduces unique identifiers, including a run ID and event ID, to precisely track individual events and mitigate issues caused by imprecise log filtering in AWS CloudWatch. These identifiers enable accurate mapping of event creation and receipt timestamps, even under high-concurrency conditions. This includes consistent use of AWS resources (e.g., Lambda memory and EC2 instance types)
3) **Flexibility [P3]:** Recognizing the diversity of workloads and system configurations in serverless environments, TriggerPerf offers extensive configurability across multiple parameters, including throughput, batch size, and event filtering. This flexibility allows users to simulate a wide range of scenarios, from low-latency single-event triggers to high-throughput workloads that stress data store scalability.

B. Architecture Overview

The core architecture of TriggerPerf is built around a modular chain of AWS Lambda functions designed to orchestrate the setup, event generation, and trigger handling in the system under test (SUT). This chain includes three key lambda functions: the setup lambda (λ_i), the write lambda (λ_w), and the receive lambda (λ_r). Each function plays a distinct role, with (λ_i) initializing the experimental environment and configuring the system under test (SUT), (λ_w) generating events (e.g., create or update) at timestamp (t_2), ensuring isolation between event generation and subsequent processing. Finally, the receive lambda acts as the trigger destination, capturing the receipt timestamp (t_4) and collecting metadata specific to the SUT for post-experiment analysis. The benchmarking

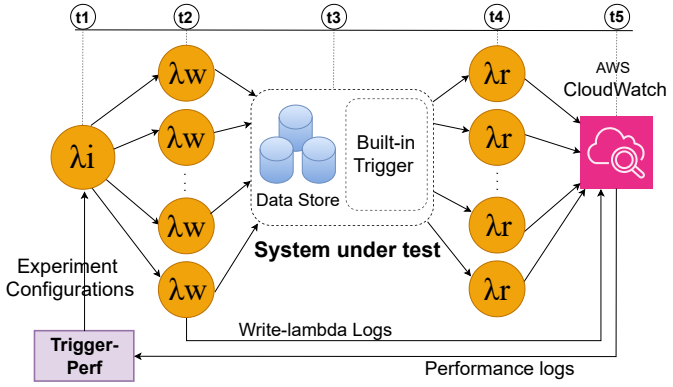

Fig. 3. TriggerPerf system design.

process is carefully instrumented to capture timestamps at key phases of the trigger lifecycle, including event creation (t_2), processing within the SUT (t_3), and event receipt (t_4). These timestamps are recorded in Unix epoch time, ensuring high precision (**P2**) and enabling detailed analysis of trigger latency. Alongside the timestamps, metadata such as batch size and event filters are logged in AWS CloudWatch. After the experiment completes, TriggerPerf extracts these logs to compute latencies and evaluates performance metrics.

These functions collectively orchestrate the benchmarking process while maintaining separation of responsibilities, ensuring the architecture can adapt to evolving trigger mechanisms and data store technologies (**P1**). Additionally, TriggerPerf supports extensive configurability (**P3**), including parameters such as throughput, batch size, and event filtering. This flexibility allows users to simulate a wide range of scenarios, from single low-latency triggers to high-throughput workloads that stress scalability.

C. Trigger Latency vs. End-to-end Trigger Latency

We define trigger latency as the time elapsed between an event occurring in the data store (t_3) and the receive lambda being invoked (t_4). However, the data stores we investigate are black-box services provided by the cloud provider whose internals we do not have access to. Therefore, we cannot measure t_3. We investigate trigger latency by measuring the time elapsed between event generation in the write lambda (t_2) and the receive lambda being invoked (t_4). We term this measurement *end-to-end trigger latency*.

The end-to-end trigger latency includes the latency of the writing to the data store. We measure the write round-trip-time (RTT) latency of data stores and find it to be much lower than the trigger latency. For example, writing 1 KB objects to AWS S3, we find a median write latency of 32 ms, which is much lower than the 1,300 ms median end-to-end trigger latency.

Therefore, we conjecture that the write part is only a small fraction of the end-to-end trigger latency that we measure. The rest of the latency is due to the trigger itself. In subsequent sections, we use trigger latency to mean end-to-end trigger latency.

IV. TriggerPerf: Detailed Design

This section details the core components of TriggerPerf. We first outline the Lambda function chain that orchestrates setup, event generation, and logging (§IV-A). Next, we describe TriggerPerf's configurability and logging capabilities, which enable precise measurement and dynamic experimentation (§IV-B). Finally, we discuss TriggerPerf's modular architecture, which facilitates extensibility and adaptability across diverse data store environments (§IV-C).

A. Lambda Function Chain.

TriggerPerf employs a chain of AWS Lambda functions (depicted in Figure 3) to orchestrate trigger setup, event generation, and latency measurement. This design ensures minimal operational overhead and allows precise timestamping of key trigger phases. The functions are described below:

- **Setup Lambda (λ_i):** Configures the trigger mechanism in the SUT. This includes setting the destination Lambda (λ_r) to handle events and generating unique identifiers (run ID and event ID) for tracking logs.
- **Write Lambda (λ_w):** Generates an event (e.g., create or update) in the SUT at timestamp t_2. By isolating the event generation into a dedicated function, TriggerPerf minimizes interference between event creation and processing, ensuring accurate latency measurements.
- **Receive Lambda (λ_r):** Acts as the trigger destination. Upon receiving the event, it logs the receipt timestamp (t_2) and extracts metadata specific to the SUT. This function ensures consistent data collection for all experiments, regardless of the underlying data store.

B. Configurability and Logging.

TriggerPerf is designed to support diverse experimental setups by providing extensive configurability and robust logging mechanisms. Users can adjust key parameters, such as:

- Throughput: Defines the number of events per second, enabling stress-testing across a wide range of workload intensities.
- Batch Size: Controls the number of events processed per trigger, allowing users to evaluate the trade-offs between latency and resource efficiency.
- Event Filtering: Applies custom filters to the workload, enabling the simulation of real-world scenarios where only specific events trigger downstream processing.

These configurable parameters make TriggerPerf adaptable to test various scenarios, from low-latency single-event triggers to high-throughput batch workloads that stress scalability limits.

Precise timestamping and logging is central to TriggerPerf's design, with logs capturing key phases of the trigger lifecycle:

- Event creation (t_2): Recorded by the Write Lambda after write/update event.
- Event receipt (t_4): Logged by the Receive Lambda upon trigger execution.

These timestamps are supplemented with metadata, such as trigger processing time (t_3, where available) and system-specific details. All data is recorded in AWS CloudWatch

and associated with unique run and event IDs. This ensures reproducibility and traceability, even under high-concurrency conditions. TriggerPerf extracts the logs from CloudWatch to local after completion of the run for further analysis.

C. Modular Design.

TriggerPerf's modular architecture is a key design feature, enabling extensibility and adaptability to a wide range of systems. Each component in the Lambda function chain operates independently, ensuring that new data stores or trigger mechanisms can be integrated without disrupting the core benchmarking framework. For example, integrating a new system under test (SUT) requires modifications only to the Setup Lambda, where the trigger mechanism is configured, while the Write Lambda and Receive Lambda remain unchanged. Additionally, system-specific workload parameters such as throughput, batch size, and event filtering can be dynamically adjusted without altering the underlying logic of the Lambda chain. This flexibility ensures that TriggerPerf can easily adapt to different benchmarking requirements. To add a new trigger mechanism for testing, users need to implement the following in a driver file:

1) A function to configure the trigger mechanism in the SUT.
2) Functions to perform CRUD operations on the SUT.
3) Configuration to set the Receive Lambda as the trigger destination.

This structured approach simplifies the process of extending TriggerPerf, enabling researchers and developers to efficiently benchmark a wide range of systems and trigger mechanisms with minimal effort.

V. Experimental Setup

We conducted all experiments on Amazon Web Services (AWS) in the 'us-east-1' region using TriggerPerf, our custom benchmarking tool. TriggerPerf leverages the AWS Boto3 and Go-v2 SDKs to interact with AWS services and facilitate the benchmarking process. For experiments involving latency measurements, we used AWS Lambda configured with 1024MB of memory and i4i.2xlarge instances from Amazon EC2 to ensure sufficient compute resources and consistent performance.

TriggerPerf executes experiments through a chain of Lambda functions: Setup Lambda (λ_i), Write Lambda (λ_w), the System Under Test (SUT), and Receive Lambda (λ_r), as shown in Figure 3. Each data store benchmarked is supported by a dedicated driver program that sets up the trigger mechanism, configures λ_r as the destination, and performs CRUD operations on the data store. The lambda functions log the timestamps to AWS Cloudwatch along with unique identifiers (RunID & EventID) and some datastore specific metadata.

To establish a baseline for comparison, we used etcd, a non-serverless distributed key-value store, as a reference system. We set up an etcd-backed Kubernetes cluster with a single worker node and deployed a containerized workload designed to perform periodic updates. Each container was configured to idle for one minute before terminating. To ensure the experiments reflected realistic usage patterns, we instrumented the etcd data store and analyzed the distribution of key and value sizes stored in the cluster. This analysis helped align our experiments with the characteristics of Kubernetes workloads. For consistency, we used a median key size of 59 bytes and a median value size of 889 bytes across all experiments.

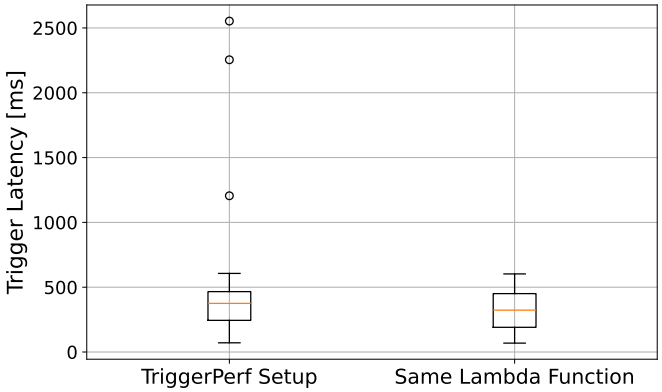

Fig. 4. AWS clock synchronization validation.

A. AWS Clock Synchronization Validation

We measure the trigger start and end times on different lambda functions and define their difference as the trigger latency. Any drift between the clocks of the two lambda functions threatens the validity of our measurements. AWS claims clocks across lambda functions are synchronized using NTP (Network Time Protocol) and across EC2 instances with the Amazon Time Sync Service.

To ensure the validity of our experiments, we assess the time drift in AWS and ensure that it is within bounds. We set up an experiment such that the lambda that receives the trigger event is the same lambda that initiated the write. This is possible because AWS keeps functions warm after they are invoked. We ensure only the trigger event invokes the lambda after a write [14]. We evaluate the trigger latency in this setup for 200 iterations. We compare the obtained latency distribution to the latency distribution when the write and trigger target are different lambdas (TriggerPerf Setup). We use DynamoDB Streams configured with 100 Write Capacity Units as our data store trigger mechanism for this experiment.

Our results are depicted in Figure 4. We observe that the distributions for the same lambda case and the **TriggerPerf** setup are approximately the same. However, we observe a few outliers in the different lambda case. We consider these results evidence that time is synchronized across different lambda functions. Additionally, we perform the Kolmogorov-Smirnov test on the distributions and get a value of 0.14; this confirms the distributions are similar [15].

VI. Experimental Characterization of Data Store Trigger Systems

Data store triggers are a critical component of severless applications, yet their performance under different conditions

remains underexplored. Understanding how these triggers behave under varying workloads is essential for designing responsive and efficient serverless systems. In this section, we systematically evaluate the latency, scalability and elasticity of data store triggers using TriggerPerf, a specialized benchmarking tool.

TriggerPerf provides a controlled benchmarking environment, allowing us to systematically evaluate trigger latency and performance across different operational conditions. Using this framework, we focus on three key performance metrics:

1) **Unloaded latency:** What is the baseline (unloaded) latency of different data store triggers?
2) **Latency under load:** How do these triggers perform under high load?
3) **Latency during elastic scaling:** How do they respond to dynamic scaling events?

We benchmark these metrics across three AWS serverless data stores, S3 Standard, DynamoDB, and Aurora Postgres and compare them to etcd, a widely used non-serverless distributed key-value store, as a reference system. Table III summarizes the experimental setup for each scenario, detailing the constant and varying parameters. Our experiments yielded the following key observations:

O-1: **High Tail Latencies Under Load:** Except for DynamoDB Streams, all evaluated trigger mechanisms exhibit significant tail latency (p90) under load, with values up to 10x higher than the median latency. This indicates challenges in maintaining consistent performance as throughput increases.

O-2: **Resilience of DynamoDB Stream:** DynamoDB Streams demonstrated robust performance, showing resilience under both high load and elastic scaling scenarios. Unlike other SUTs, its tail latencies remained within acceptable bounds even as workload intensity increased.

O-3: **S3 Event Notification Stabilization:** While S3 Event Notification initially exhibited elevated latencies during scaling, its performance stabilized at a higher steady-state latency after the scaling event. This behavior highlights S3's scaling mechanism but also indicates potential limitations in achieving low-latency triggers post-scaling.

O-4: **Aurora Serverless UDF Latency Spikes:** Aurora Serverless UDFs showcased significant performance degradation during elastic scaling, with p90 latencies increasing by up to 9x compared to baseline values. This result highlights the challenges of maintaining trigger performance in relational database environments with scaling events

A. Unloaded Latency: Baseline Performance Analysis

Unloaded latency serves as a critical baseline for measuring system performance under minimal load. This metric provides insights into the fundamental efficiency of the system without the influence of workload-induced variability [4].

TABLE III
EXPERIMENTS OVERVIEW. SUT = SYSTEM UNDER TEST. FUNCTION = TRIGGER TARGET FUNCTION.

Section	Constant parameters	Varying parameters	Metrics
§V-A	Concurrency: 1, Iters: 200	Function	Latency
§VI-A	Concurrency: 1, Iters: 200	SUT	Latency
§VI-B	Throughput: 3000 req/s	SUT	Latency
§VI-C	Throughput: 2,000 (before); 4,000 (after)	SUT	Latency

Setup. Using TriggerPerf, we evaluated the unloaded trigger latencies of the selected systems. DynamoDB was tested with autoscaling enabled, S3 with event versioning activated, and Aurora PostgreSQL configured with engine version 15.4 on a db.r6g.2xlarge (memory-optimized) instance type. For etcd, we deployed a three-node cluster on EC2 instances, with the experiments executed from another EC2 instance in the same VPC. The etcd benchmark utilized the etcd/clientv3 Golang API. To ensure stability, we initiated 200 warm-up put events for all systems before performing the primary benchmark of 200 trigger events, each separated by a one-second interval.

Results. The results, visualised in Figure 5, reveal substantial differences in the unloaded latency performance of the tested data stores. Among all systems, etcd exhibited the lowest latency, with a median value of just 2.6ms. Its latency distribution was tightly concentrated, indicating minimal variability and highly efficient event propagation. AuroraDB performed significantly worse than etcd, with a median latency of 103ms. Despite the higher latency, its response times were relatively consistent, as indicated by a steep ECDF curve with low variance. DynamoDB Streams exhibited slightly higher latency, with a median of 170ms. While its performance was still better than S3, DynamoDB showed a somewhat broader distribution, indicating occasional variations in trigger responsiveness. This behavior may be attributed to internal polling mechanisms and batching strategies used within the DynamoDB Streams architecture. S3 Event Notifications performed the worst, with a median latency of 1,157ms and a tail latency (p90) exceeding 1,700ms. Unlike other systems, S3 displayed high variability, with a gradual ECDF curve indicating unpredictable delays in event propagation. This inconsistency suggests that S3's trigger mechanism is less optimized for low-latency workloads.

The results demonstrate that serverless triggers introduce significantly more latency than traditional event-driven mechanisms such as etcd. Even in an unloaded state, AWS-managed triggers show a 10-100× increase in latency compared to a non-serverless alternative. This highlights the inherent overhead associated with cloud-managed event propagation and suggests that serverless architectures must account for these delays when designing latency-sensitive applications.

B. Latency Under Load

While unloaded latency provides a baseline measurement, real-world serverless applications often experience sustained

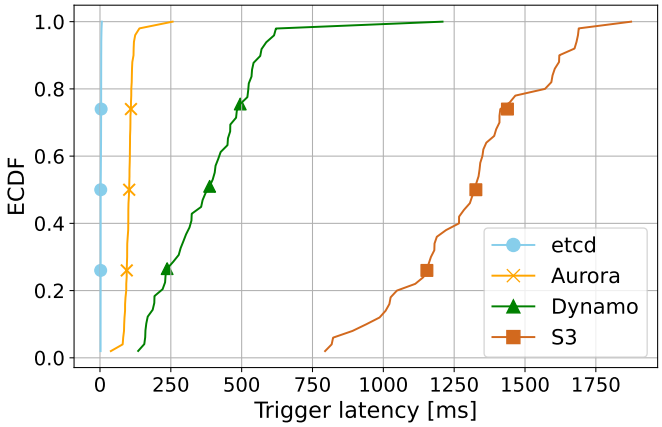

Fig. 5. Unloaded trigger latency across systems.

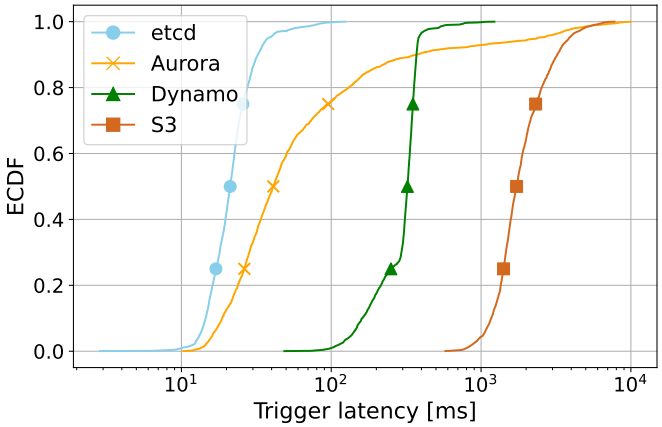

Fig. 6. Trigger latency under load across systems.

workloads with high concurrency. Under such conditions, the efficiency of data store triggers can degrade due to increased contention, queuing delays, and resource limitations. This experiment examines how trigger latency behaves under sustained load

Setup. For this evaluation, we generated put/update events at increasing request rates until each system reached approximately 80% utilization of its key resources. For etcd and Aurora PostgreSQL, this threshold was defined as 80% CPU utilization across the cluster. For DynamoDB Streams, the load was set to 80% of the provisioned write capacity units (WCU) to simulate peak usage. S3, which lacks configurable resource utilization parameters, was benchmarked under the same request intensity. All experiments were conducted using a dedicated EC2 instance to ensure a consistent testing environment.

The optimal requests per second (req/s) necessary to maintain stable 80% utilization was determined through iterative trials for each system. Once the req/s was identified, the systems were subjected to sustained load for 5 minutes, with put/update events continuously dispatched at the determined intensity. To ensure meaningful results, latency measurements were collected exclusively during the third minute of the experiment, discarding data from the warm-up and cool-down phases.

Results. Figure 6 illustrates the impact of high load on trigger performance across the tested data stores. Under load, significant performance differences emerge, with some systems exhibiting severe latency spikes and others maintaining relatively stable response times.

DynamoDB Streams demonstrated the most resilience under high load, maintaining relatively stable latencies across all percentiles (**O-1**). The system's internal event processing model appears to efficiently distribute workload pressure, resulting in only a 10% increase in p90 latency compared to unloaded conditions. This suggests that DynamoDB's trigger architecture effectively handles high-frequency updates without substantial degradation in response time. AuroraDB exhibited moderate performance degradation, with its p90 latency increasing by 3.5-4x compared to its unloaded state. While the system remained functional, the increased tail latency indicates that contention within the database's internal event-handling mechanisms may introduce additional delays under stress. S3 Event Notifications struggled the most under load, experiencing severe degradation in both median and tail latency. At peak stress levels, p90 latencies reached up to 10 times the median value, highlighting fundamental scalability limitations in its event notification system.

The high tail latencies observed in etcd, Aurora, and S3 under load highlight limitations in their resource management strategies when handling high request concurrency. In contrast, DynamoDB Streams maintained tail latencies comparable to its unloaded performance, indicating robust resource allocation and effective workload handling. This consistency makes DynamoDB Streams well-suited for latency-sensitive applications where minimizing variability is critical.

C. Latency After Elastic Scaling

The bursty nature of serverless workloads [2] requires that data stores and trigger mechanisms elastically scale in response to sudden fluctuations in application demand. Elastic scaling performance is critical for maintaining responsiveness and preventing downstream processing bottlenecks [16]. To evaluate the elasticity of trigger mechanisms, we measured latency during and after scaling events.

Setup. For this experiment, we benchmarked Aurora PostgreSQL Serverless v2 [17], DynamoDB with autoscaling, and standard S3. The warm-up phase involved stabilizing Aurora and DynamoDB at 2000 requests per second (req/s) for 15 minutes, while S3 required a 40-minute warm-up due to its longer initialization time. Following this, we executed put/update events over a 6-minute period with the following pattern: 2000 req/s for the first 3 minutes (stable phase), followed by 4000 req/s for the next 3 minutes (scale-up phase). Latency data from the third (pre-scale-up) and fourth (post-scale-up) minutes were collected for analysis.

Results. Figure 6 presents the latencies observed during these experiments. Across all systems, a notable increase in latency was observed during the scale-up phase. For S3, the median trigger latency surged by 30-40%, increasing from 1,157 ms (pre-scale-up) to a new stabilized baseline of

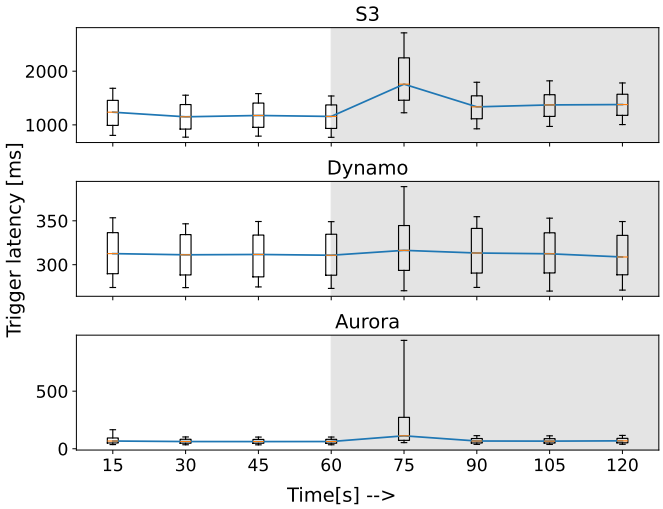

Fig. 7. Trigger latency on elastic scaling from 2000 req/s to 4000 req/s. The greyscale region of the graph represents the duration after the scale-up event.

1,335 ms after the scale-up (**O-3**). This behavior highlights the overhead associated with scaling its event notification pipeline. DynamoDB Streams demonstrated the most consistent performance, with minimal impact on latency and interquartile range (IQR) after scaling (**O-2**). This resilience is indicative of efficient resource allocation and autoscaling mechanisms designed to handle bursty workloads without significant degradation. In contrast, Aurora PostgreSQL exhibited a pronounced increase in variability during the scale-up phase. Its IQR nearly doubled, reflecting a broader spread of latencies during this period. Although the median latency returned to pre-scale-up levels, tail latencies reached as high as 9x the median at the 90th percentile (**O-4**). These elevated tail latencies highlight challenges in managing resource contention and maintaining performance during rapid scaling events.

The elevated tail latencies observed for both Aurora and S3 during scale-up events underscore a fundamental limitation in their design for handling rapid elasticity. Such behavior underscores the need for serverless systems to better integrate elastic scaling mechanisms that minimize latency spikes, even under bursty and unpredictable workloads.

VII. Discussion

We discuss the limitations of serverless trigger mechanisms and our experiments in this section. We end with a call for more configurability in serverless data store triggers.

Different experiment protocols for each service. Comparing data store trigger mechanisms head-on presents challenges due to varying service specific parameters and implementation opaqueness. This necessitates experimental protocols specifically tailored for each service, complicating head-on evaluations.

First, warm-up and cool-down times differ widely between services, which affects their readiness to handle experimental workloads. For example, DynamoDB retains historical load data for limited and unspecified perios, allowing it to handle load changes faster [18]. In contrast, S3 requires up to 30 minutes of warm-up to sustain the desired request rate without rejecting writes. However, S3 also cools down rapidly after an experiment, unlike Aurora and DynamoDB. These inconsistencies make standardized benchmarking complex.

Second, database event complexity adds another layer of variability. Our experiments focus exclusively on put/update events for consistency across systems. However, data store triggers often handle a variety of operations and queries with differing complexities. Evaluating performance across these broader scenarios would provide a more comprehensive characterization of trigger mechanisms and their real-world applicability.

Finally, distributed clock error bounds pose challenges in accurately measuring trigger latencies across cloud environments. Clock synchronization in distributed systems is inherently difficult, leading to minor inaccuracies in latency measurements. To address this, we designed experiments to minimize synchronization issues and validated the impact of these errors in Section 4. We show that these errors are negligible and do not affect the validity of our analysis.

These factors highlight the intricacies of benchmarking data store trigger mechanisms, underscoring the need for tailored protocols to address the unique characteristics of each service while ensuring meaningful comparisons.

Trigger customization limitations. Current data store trigger mechanisms often lack the necessary level of customizability, posing a significant challenge for systems designers. This inflexibility restricts the ability to tailor triggers to the specific needs of individual applications. Batch trigger requests are absent in S3 Event notifications and AuroraSQL Lambda Invoke UDFs, unlike DynamoDB Streams. This can result in the excessive triggering of destinations(e.g., Lambda functions), potentially cascading downstream and increasing costs. In the activation phase of AuroraSQL UDFs, the only supported destination is Lambda functions. Such constraints limit the possible design space of serverless applications. The observed latencies increase 10x under load compared to the unloaded case. The increase is 100x if we consider the tail (p90). This makes application performance highly variable.

Towards a data store trigger mechanism for serverless. A good data store trigger solution for serverless architectures should offer sufficient customization options to optimize performance, cost-efficiency, and flexibility. This can be achieved by enhancing customizability during both the setup and activation phases of triggers. In the setup phase, programmable triggers with User-Defined Functions (UDFs) allow embedding event selection, filtering, transformation, or validation logic directly within the trigger. This potentially eliminates separate serverless functions, simplifying workflows and reducing overhead. For the activation phase, batching and polling features will allow system designers to strike the required balance between responsiveness (frequent polling/small batch size) and cost-effectiveness (less frequent polling/large batch size).

VIII. Related Work

Event Trigger Benchmarks and Systems. The most closely related works to our study are TriggerBench, Unum, and Triggerflow. Among these, TriggerBench [8] focuses solely on evaluating unloaded latency (Section VI-A) across a limited subset of serverless triggers. While it includes S3 in its evaluation, it does not extend to DynamoDB or Aurora, leaving significant gaps in coverage. Additionally, TriggerBench lacks the configurability and modularity needed to fully explore diverse workload patterns and trigger mechanisms, limiting its applicability in broader benchmarking scenarios. Unum [3] implements a polling-based trigger mechanism and compares to AWS Step Functions across multiple serverless workflows. Triggerflow [7] implements a trigger-based serverless workflow coordinator and compares it with workflow coordinators from cloud providers. With TriggerPerf, we provide the first comprehensive benchmark of all three serverless data stores offered by AWS (S3, Dynamo, Aurora). We are also the first to characterize their performance in an unloaded, loaded, and elastic scaling scenario.

Serverless Performance. Multiple works describe the impact of trigger performance on total application execution time [2]–[4]. SONIC [19], WISEFuse [20], and Pheromone [21] propose other mechanisms such as speculation and fusion to reduce function coordination overhead. Serverless storage systems such as Pocket [22], Cloudburst [23], Boki [24], and PolarDB [25] are elastic, but do not have built-in trigger mechanisms. State-of-the-practice event trigger databases such as Google Cloud Firestore [26] and Microsoft Azure SQL Database [27] have similar performance characteristics to their AWS counterparts discussed in this work, but their trigger performance still needs to be evaluated. Storage functions [28], Durable functions [29], and Lambda objects [30] are systems that run computation in the data store itself. Although related, they are a different kind of system and require a separate evaluation.

IX. Conclusion

Many modern data stores are equipped with trigger mechanisms to invoke serverless functions or connect distributed components in systems such as Kubernetes following data object create, update, or delete events. In this paper, we demonstrate that data store triggers are essential in serverless performance, but their underlying mechanisms are not well understood. We are the first to present a qualitative analysis of data store trigger mechanisms and present TriggerPerf, our tool for benchmarking trigger mechanisms. Between the AWS S3, DynamoDB, and AuroraDB data stores, we find a 92% difference in median trigger latency (latency), a 94% difference for concurrent event triggers (scalability), and an up to 40% increase in latency when abruptly doubling the number of concurrent triggers (elasticity). Moreover, we find limited trigger customizability and argue for the use of User-Defined Functions, batching, and polling to meet serverless system's needs.

Acknowledgements

This work was supported by the EU Horizon Graph Massivizer and the EU MSCA Cloudstars projects. This research was partly supported by a National Growth Fund through the Dutch 6G flagship project "Future Network Services".

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019.

[2] M. Shahrad, R. Fonseca, I. Goiri, G. I. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 2020, pp. 205–218.

[3] D. H. Liu, A. Levy, S. A. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 2023, pp. 1505–1519.

[4] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," 2022.

[5] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: a serverless benchmark suite for function-as-a-service computing," in *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*. ACM, 2021, pp. 64–78.

[6] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 2020, pp. 30–44.

[7] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: trigger-based orchestration of serverless workflows," in *14th ACM International Conference on Distributed and Event-based Systems, DEBS 2020, Montreal, Quebec, Canada, July 13-17, 2020*. ACM, 2020, pp. 3–14.

[8] J. Scheuner, M. Bertilsson, O. Grönqvist, H. Tao, H. Lagergren, J. Steghöfer, and P. Leitner, "Triggerbench: A performance benchmark for serverless function triggers," in *IEEE International Conference on Cloud Engineering, IC2E 2022, Pacific Grove, CA, USA, September 26-30, 2022*. IEEE, 2022, pp. 96–103.

[9] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. S. III, S. Sosothikul, D. Terry, and A. Vig, "Amazon dynamodb: A scalable, predictably performant, and fully managed nosql database service," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022, pp. 1037–1048.

[10] K. Contributors, "Operating etcd clusters for kubernetes," 2024. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

[11] V. Contributors, "Vitess - topology service," 2023. [Online]. Available: https://vitess.io/docs/17.0/concepts/topology-service/

[12] O. Contributors, "Openstack - base services," 2018. [Online]. Available: https://governance.openstack.org/tc/reference/base-services.html

[13] S. Sagkriotis and D. Pezaros, "Scalable data plane caching for kubernetes," in *18th International Conference on Network and Service Management, CNSM 2022, Thessaloniki, Greece, October 31 - Nov. 4, 2022*. IEEE, 2022, pp. 345–351.

[14] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 133–146.

[15] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[16] M. Copik, A. Calotoiu, G. Rethy, R. Böhringer, R. Bruno, and T. Hoefler, "Process-as-a-service: Unifying elastic and stateful clouds with serverless processes," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 223–242.

[17] AWS, "Aurora serverless v2," https://aws.amazon.com/rds/aurora/serverless/, 2024, accessed: 2024-04-30.

[18] ——, "Amazon dynamodb auto scaling: Performance and cost optimization at any scale," https://aws.amazon.com/blogs/database/amazon-dynamodb-auto-scaling-performance-and-cost-optimization-at-any-scale/, 2019, accessed: 2024-04-30.

[19] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 285–301.

[20] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "WISEFUSE: workload characterization and DAG transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 26:1–26:28, 2022.

[21] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 2023, pp. 1489–1504.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 2018, pp. 427–444.

[23] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2438–2452, 2020.

[24] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021, pp. 691–707.

[25] Y. Zhang, X. Yang, H. Chen, F. Li, J. Xu, J. Zhou, X. Wu, and Q. Zhang, "Towards a shared-storage-based serverless database achieving seamless scale-up and read scale-out," in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 5119–5131.

[26] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The nosql serverless database for the application developer," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 3376–3388.

[27] O. Poppe, Q. Guo, W. Lang, P. Arora, M. Oslake, S. Xu, and A. Kalhan, "Moneyball: Proactive auto-scaling in microsoft azure SQL database serverless," *Proc. VLDB Endow.*, vol. 15, no. 6, pp. 1279–1287, 2022.

[28] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 1–12.

[29] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[30] K. Mast, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Lambdaobjects: re-aggregating storage and execution for cloud computing," in *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*. ACM, 2022, pp. 15–22.