Comparative Analysis of Three IoT Data Storage System Architectures on AWS Cloud

Dominik Rohal*, Lucy Ellen Lwakatare†, Yusheng Wu*, Jesse Haataja*, Jukka K Nurminen†, and Juha Kangasluoma*
*Institute for Atmospheric and Earth System Research, University of Helsinki, Helsinki, Finland
†Dept. of Computer Science, University of Helsinki, Helsinki, Finland
Email: dominik.rohal@helsinki.fi, lucy.lwakatare@helsinki.fi, yusheng.wu@helsinki.fi, jesse.haataja@helsinki.fi, jukka.k.nurminen@helsinki.fi, juha.kangasluoma@helsinki.fi

***Abstract*—Internet of Things (IoT) devices generate large amounts of data, creating the challenge of designing efficient IoT cloud storage solutions. This study focuses on an IoT application managing air quality measurement data, which requires frequent retrieval of recent data for near-real-time monitoring and access to long segments of time-series data.**

The study performs a comparative analysis of three IoT storage system architecture options on Amazon Web Services (AWS) cloud. In contrast to existing literature, the options take into account data access and retrieval patterns specific to the use cases of the IoT application under study, where only data produced by a particular device is accessed at a time. The first option utilizes a NoSQL database, the second employs data streams and a data lake, and the third uses a time-series database. Each option was tested by performing retrievals of different lengths of time segments, focusing on retrieval times and query capabilities.

Results show that a time-series database is suitable for applications benefiting from query options or time-series functions, such as interpolation. A Key-Value NoSQL database is suitable for applications retrieving short time segments without extensive queries. Conversely, a data lake excels in storing and retrieving large volumes of raw data, while data streams are unsuitable as a direct source of data for near-real-time applications.

The study provides a framework for comparing alternative architectures to analyze their suitability, guiding decisions in selecting the most appropriate storage solution based on specific IoT application requirements.

***Index Terms*—AWS, IoT, DynamoDB, Timestream, S3, Kinesis, data streams, data lake, time-series**

I. Introduction

In the settings of scientific research and monitoring of outdoor or indoor environment, various measurement devices and sensors are used. If connected to internet, they compose an interconnected network matching definition of Internet of Things (IoT) [1]. In the context of IoT, measurement devices and sensors are referred to as things.

This study aims to address questions that arose from designing a cloud-based IoT infrastructure for measurement devices used at the Institute for Atmospheric and Earth System Research (INAR) at the University of Helsinki. During the experiments, data is analyzed and visualized in real-time, and ad-hoc retrievals of days-long measurements are performed for further analysis.

In the current setup, measurements are published by devices individually as Message Queuing Telemetry Transport (MQTT) messages. As the message broker, IoT Core services provided by Amazon Web Services (AWS) are used which direct the measurements and save them in DyanmoDB which is used as the only data storage from which data is retrieved for real-time monitoring and analysis of large volumes of data. Retrieval of large volumes of data from DynamoDB was proven to be challenging as the response hits the limit of the maximum response size. Thus, the employment of retrieval strategies such as pagination or subsequent queries is needed to retrieve all needed data. Additionally, to be able to perform increased number of read operations, depending on the service configuration, DynamoDB read capabilities must be scaled up and later scaled down to avoid unnecessary costs.

The aim is to suggest an alternative IoT data storage architecture which due to business decision must be built in AWS cloud. There is a lack of existing studies comparing specifically suitability of different architectures composed of AWS services for storing IoT data for applications which require capability for both, enabling data to be used by real-time or near-real-time applications and extract additional information from large volumes of data recorded over long periods of time. Additionally, few existing studies [2], [3] that compared and analyzed different solutions for storing IoT data did not consider specifics of IoT applications, and access patterns and use-cases for which different AWS services were optimized and intended to be used for. Also, services in this study are provisioned directly from the cloud provider unlike in [4] where a virtual machine was used to run the databases. This paper is a comparative analysis of data retrieval performance that bridges that gap.

The analysis is conducted using metrics collected from a simulation of a network of 100 simultaneously running scientific devices. In addition to the Key-value NoSQL database and the time-series database, a third option utilizing data streams and a data lake is also included. In this option, possibility of reading real-time data directly from data streams is studied.

This paper begins by providing context and an introduction to data storage options in Section II. The IoT data storage system architectures selected for comparison, along with the evaluation criteria and experiment implementation, are described in Section III. Experiment results are presented in Section IV, followed by an evaluation of these results and suggestions for alternative solutions in Section V. Finally, conclusions are provided in Section VI.

II. Background and Related Work

This section provides background information on IoT system architectures and, with the study's focus on IoT cloud-based data storage systems, includes a summary of different storage types commonly used in IoT systems. The different storage types and technologies introduce the main components considered in experiment evaluations in Section III.

The general IoT system architecture presented in literature is composed of several key layers, including a device layer, network layer, middleware layer, cloud layer and application layer [5]–[7]. The cloud layer facilitates storage, processing and analysis of data from the IoT devices for use in upstream applications. There is no one-size-fits-all architectural style for IoT systems, and the architectural options vary in each layer. For example, a summary of architecture options for the IoT device layer is provided in [8], highlighting that design choices are influenced by their intended use, power budget, and the necessity to support third-party development or dynamic programming. Furthermore, the feasibility of certain IoT system architectures also depends on the available capabilities of an IoT device [8].

Focusing on storage in the cloud layer, IoT data storage systems on cloud platforms employ different kinds of databases (relational and NoSQL) or file-based systems (Hadoop) to store and retrieve data collected from IoT devices. Generally, data is stored either in a single or multiple data stores, with the structure of the data being one of the factors informing the decision. A related work [4] has compared different storage approaches on three leading cloud platforms (AWS, Google Cloud Platform, and Microsoft Azure) based on an IoT environmental monitoring application. Like this study, the evaluations in the study [4] measure the response time of a request to a specific storage system deployment on the AWS cloud. Similar studies focusing on AWS cloud and IoT settings performed architecture evaluations by assessing data ingestion rates and analysing the performance metrics of AWS serveless functions utilized [3]. Authors in [6], [7] observed both extreme variation in architecture evaluation metrics and a lack of practical experiments to verify their accuracy, even when studies focus on specific application areas, such as an indoor positioning system [6]. According to the authors [6], practical experiments are necessary for verification because some IoT system architectures are highly dependent on the application contexts used, which also strongly influence performance criteria. For example, latency is a key requirement for an indoor positioning system [6]. This study provides an in-depth evaluation of storage design options that consider the requirements and use cases of IoT application for managing air quality measurement data presented in Section (III) part B.

A. Relational and No-SQL Databases

NoSQL databases have been shown to have superior performance compared to relational databases when managing massive unstructured IoT data [9], [10]. NoSQL databases do not require a fixed data schema, and the models used for storing data, such as key-value model, provides horizontal scalability and high availability achieved through data replication and partitioning over many nodes [11]. A drawback is that complex queries are difficult to perform on NoSQL databases [9]. NoSQL databases can be classified into key-values store, document databases, column-oriented databases. Data in key-value stores is stored in the form of key-value pair. Document databases also use key-value pairs, but the values are stored in document structures, such as XML. Column-oriented databases store data tables in columns rather than rows, making them efficient for applications that require intensive column-based calculations, such as aggregation.

Since data collected from IoT devices is largely in the form of time series, best practices for storing and processing time series data in both relational and NoSQL databases are discussed by Namiot [12]. According to the author [12], for optimal performance, a NoSQL solution should only be considered in scenarios involving big IoT time series data where high data velocity is a key factor. On the other hand, a comparison between a cloud-optimized general-purpose relational database (BigQuery) and a specialized database for time series data (InfluxDB) showed that time series databases are an excellent solution for all application contexts involving extensive use of time series data [4].

B. Time Series Databases

Time series databases are non-relational, specialized databases optimized for handling time series data, where each entry is associated with a timestamp [13]. Specifically on the AWS cloud platform, authors in [3] compared a traditional relational database (PostgreSQL), various NoSQL databases (DynamoDB, MongoDB), and a specialized time-series database (InfluxDB). Results focusing on querying a time segment showed InfluxDB and MongoDB databases to generally be the best performing options. However, in operations concerning a single data item, DynamoDB performed better. In contrast, authors in [14] found that the DynamoDB database demonstrated poor performance primarily due to data partitioning across multiple nodes, which slowed operations on time series data.

[15] [2]

III. Architecture Evaluations

This section describes three data storage system architectures selected for comparison analysis, as well as the evaluation criteria and implementation details of experiment setup.

A. IoT Storage System Design Options

This subsection describes IoT data storage system architectures selected for comparative analysis. All options adopt the data flow of the current solution, where scientific measurement devices publish their measurements individually as MQTT messages. These messages are then routed to appropriate services using AWS IoT Core services. The specific service to which the data is published depends on the specific option. In the experimental setup, data from each MQTT message is sent to multiple services, with each service belonging to a different storage system design option. Figure 1 visualizes all the evaluated architectural options and the corresponding data flows.

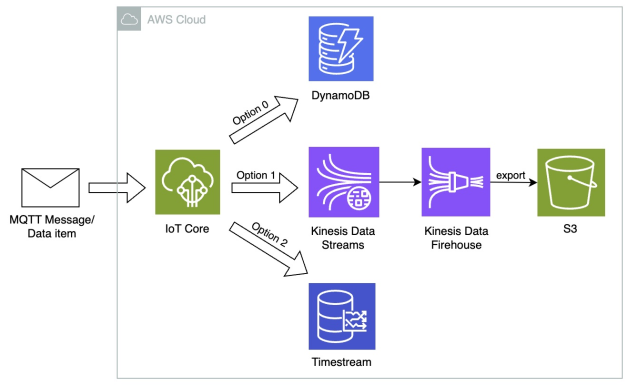

Fig. 1. Data storage system architecture options

1) Option 0: ***Key-value NoSQL database*** DynamoDB, utilized by the current solution, is also included in the comparison. The configuration of the service is optimized for saving time-series data. This is achieved by using timestamp as a sort key, keeping data items indexed by time and improving time segment query performance.

Performance of data retrieval by specific device is optimized following recommendations in [16]. Device identifier is used as partition key, keeping all data of the same device stored in the same node, avoiding use of the scan operation. Instead, data from the same device is retrieved by faster query operation which retrieves data by partition key, in our case device identifier.

In this option, all measurement parameters are saved in a single column as a data object to support heterogeneous data.

2) Option 1: ***Data streams and data lake*** are used in the first alternative data storage system architecture. This options in inspired by architecture used in [2] and pattern 1 in [15]. It utilizes Kinesis Data Streams and S3 service is used as long-term storage for historical data. Inability to save data in S3 data item by data item requires Kinesis Data Firehouse service to be used as a data buffer that flushes batch of grouped data to S3 every 15 minutes which is the maximum buffer time. Real-time data is retrieved from the Kinesis Data Streams service as it takes up to 15 minutes for data to be present in S3. Unlike solution described in [2] which uses a separate data store for real-time data or pattern 1 in [15] which uses separate data pipelines for real-time data and batch data, our solution retrieves real-time data directly from Kinesis Data Streams service.

In Kinesis Data Streams, data is partitioned by device identifier, resulting in data from the same device being queued in the same shard. In S3, each file that contains data items has a prefix that consists of device identifier, year, month, day and hour, enabling querying data by device and time with granularity of one hour. Since data is exported every 15 minutes or less, multiple files can share the same prefix.

Including this option in the comparison aims to verify viability of querying real-time data directly from Kinesis Data Streams service.

3) Option 2: ***Time-series database*** is chosen as the second alternative solution. Including it in comparison aims to explore benefits of the specialized time-series database in an IoT application. Querying by device is enabled by using device identifier as a dimension.

B. Evaluation Criteria

The options included in the comparison are evaluated based on experiment simulating 100 devices; therefore, real use of the system. There is no difference between simulated device and real device from the perspective of the storage system.

In this study, we consider the access patterns observed from our IoT application, which served as the motivation for our research. It is reasonable to assume that other IoT applications exhibit similar or identical data access patterns. The access patterns identified in our observations are as follows:

1) Write workload is predictable and depends on the number of running devices.
2) The read workload varies over time with periodic or ad-hoc retrievals of hours or days long data segments causing spikes in the read workload.
3) Data is always retrieved for specific device.
4) First real-time data retrieval queries a longer segment of the most recent data (30 minutes) which is periodically (every 1 minute) updated with the newer data.
5) Longer segments (hours or days) of older data are retrieved for extracting information from longer data sets.

Three data retrieval types, derived from IoT data access patterns, are evaluated separately using descriptive statistical methods. Each type retrieves a specific segment of time-series data produced by a single device in a one second resolution:

- **Long recent data:** A 30 minutes long segment of the most recent time-series data. The usual use case is the first data fetch by real-time application.
- **Short recent data:** It differs from long recent data only in its length, being limited to 1 minute. This data segment is typically used for periodic updates to fetch newly created items in real-time applications.
- **Historical data:** One day long segment of older data. Typically retrieved for data analysis.

All data retrieval types are performed in two modes, returning the following data items:

- **Full data items:** It returns a set of data items in which each data item consists of all parameters measured by given device at certain time.
- **Specific parameter:** It requires data items to contain only a single given parameter.

In the context of scientific measurements, a parameter refers to a value of a measurement, such as temperature or humidity.

In the experiment, a full data item in JavaScript Object Notation (JSON) format, cleared of unnecessary characters such as white spaces, has at most 511 characters, and therefore bytes. It consists of a device identifier, a timestamp, and 27 parameters. On the other hand, a data item containing a specific parameter includes only one parameter along with the timestamp and device identifier.

Each retrieval type and mode is evaluated separately using metrics from the experiment and qualitative observations of the retrieved data and retrieval process.

A total of 1800 time-series data segments are retrieved during the experiments (3 evaluated options × 3 retrieval types × 2 modes × 100 devices) to measure the following metrics for evaluation:

- **data retrieval execution time**
- **number of requests**
- **number of retrieved data items**

The main metric is data retrieval execution time, which reflects retrieval performance. The remaining metrics offer insights into potential factors contributing to good or poor performance.

Additional insights into the performance and usability of the AWS services used for data retrieval are provided through observed qualitative measures:

- **query options:** support of querying by device, querying by time and querying of specific parameter
- **data structure:** how retrieved data items are grouped or split into multiple records in data retrieval request response
- **need to adjust service configuration:** for example, DynamoDB in provisioned requires adjusting write and read capacity units to support changes in write and read workloads [17]

C. Experiment Implementation and Analysis

Experiment is conducted in four steps, repeated for each retrieval type. The code for all steps is located in a GitHub repository [18].

1) Step 1: ***Infrastructure provisioning*** is done with help of Infrastructure as a Code (IaC) tool Terraform [19], enabling repetitive and seamless provisioning and deprovisioning of identical infrastructure setup.

2) Step 2: ***Device simulation*** for the recent data retrieval part of the experiment is executed from the local machine while for historical data retrieval part it is executed from Amazon Elastic Compute Cloud (EC2) instance.

The simulation script is written in Python and simulates 100 devices concurrently, with each device generating one data item per second. Each data item is sent to the cloud individually as MQTT message.

3) Step 3: ***Data retrieval*** script is written in Python and uses client from AWS software development kit (SDK) for Python - Boto3 library [20]. An overview of the services from which recent and historical data are retrieved for each evaluated option is presented in Table I. In the retrieval code, data is retrieved in all cases by device identifier and within a specific time range. Each retrieval run for specific option performs 100 data retrievals, with data being retrieved from a different device in each retrieval. If all data cannot be retrieved in a single request due to exceeding the response body size limit, subsequent requests are made.

TABLE I
RECENT AND HISTORICAL DATA SOURCE PER STORAGE OPTION

Option	Recent data	Historical data
Option 0	DynamoDB	DynamoDB
Option 1	Kinesis Data Streams	S3
Option 2	Timestream	Timestream

TABLE II
REQUESTS USED FOR DATA RETRIEVAL

DynamoDB	
query*:	returns data items
Kinesis Data Stream	
describe_stream:	returns shards with hash key ranges
get_shard_iterator:	returns shard iterator
get_records*:	returns data items
S3	
list_objects*:	lists objects with the specified prefix
get_object*:	returns object content - data items
Timestream	
query*:	returns data items
** multiple requests in a data retrieval*	

Retrieving data items from Kinesis Data Streams and S3 requires additional information to be fetched from the service first. Therefore, a sequence of different types of requests is preformed. If a single request is not sufficient to retrieve all needed data, subsequent requests are performed. The specific request types for each service are listed in Table II.

4) Step 4: ***Data retrieval analysis*** is done by processing a file containing metrics recorded during data retrieval and performing descriptive statistical analysis over it. In a file, each row contains metrics recorded from retrieval of data from different device.

IV. Experiment Results

This section presents the experiment results for each data retrieval type.

A. Short recent data retrieval

Figure 2 summarizes the retrieval times for short recent data. In the full data item retrieval mode, the best-performing option was Option 0, using DynamoDB, with a median retrieval time of 0.26 seconds. This was followed by Option 2, utilizing Timestream, with a median time of 0.57 seconds. The worst-performing option was Option 1, which used Kinesis Data Streams and had a median retrieval time of 1.08 seconds.

In specific parameter retrieval mode, only Option 2 (using Timestream) supported querying specific parameters, while Option 0 and Option 1 continued to retrieve full data items. The benefits of Timestream's query options (used in Option

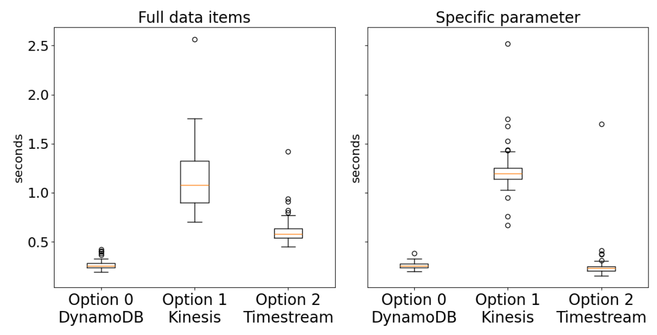

Fig. 2. Short recent data retrieval times

TABLE III
NUMBER OF REQUESTS PERFORMED TO RETRIEVE SHORT RECENT DATA

	Option 0 *DynamoDB*	Option 1 *Kinesis*	Option 2 *Timestream*
full data items	1	3 - 14 (median: 4)	1
spec parameter	1	3 - 14 (median: 8)	1

2) become evident in Figure 2 when comparing retrieval times between full data item and specific parameter modes. In specific parameter retrieval mode, Timestream outperformed DynamoDB, achieving a median retrieval time of only 0.23 seconds. As expected, Options 0 and 1 showed similar performance between specific parameter and full data item retrieval modes. In specific parameter mode the median retrieval times were 0.25 and 1.20 seconds for Option 0 and Option 1 respectively.

Table III summarizes number of requests needed to be performed to retrieve short recent data. The big difference in the number of requests to retrieve data from Kinesis Data Streams (3-14 requests) is caused by the retrieval strategy used to retrieve data from Kinesis Data Streams service. Due to query limitation, subsequent data retrieval requests are called until a more recent data item than expected is returned. It might take time until such item is available in the service for retrieval and returned and thus this results in increased number of requests which varies between retrievals.

The median numbers of items returned in short recent data retrievals are summarized in Table IV. The expected number of data items was 60, one for each second in a minute.

Median value suggesting one missing data item in data retrievals from DynamoDB and Timestream is explained by the fact, that in many cases, the data item from the most recent second was not yet available for retrieval at the time the request was performed and missing data item due to nor

TABLE IV
MEDIAN NUMBERS OF RETRIEVED DATA ITEMS FOR SHORT RECENT DATA RETRIEVALS

	Option 0 *DynamoDB*	Option 1 *Kinesis*	Option 2 *Timestream*
full data items	59	1680	59
specific parameter	59	1681	59

TABLE V
NUMBER OF REQUESTS PERFORMED TO RETRIEVE LONG RECENT DATA

	Option 0 *DynamoDB*	Option 1 *Kinesis*	Option 2 *Timestream*
full data items	1	6 - 8 (median: 8)	7
specific parameter	1	6 - 8 (median: 8)	1

being created by device at that specific second or network connection problem.

Kinesis Data Streams returned too many data items due to inability to query data items by specific device and returning data items from all devices that shared the same shard.

B. Long recent data retrieval

The box plot in Figure 3 shows an overview of retrieval times for long recent data. These results are consistent with the retrieval times observed for short recent data.

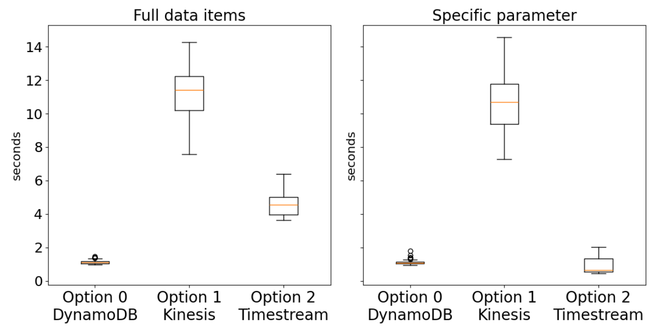

Fig. 3. Long recent data retrieval times

In full data item mode, the best-performing option was Option 0 (using DynamoDB) with a median retrieval time of 1.11 seconds. This was followed by Option 2 (using Timestream) with a median retrieval time of 4.55 seconds. The worst-performing option was Option 1 (using Kinesis Data Streams), with a median retrieval time of 11.41 seconds.

The advantages of the query options in Option 2 (utilizing Timestream) are also evident in the context of long recent data retrieval. In the specific parameter mode, Option 2 consistently outperformed Option 1, achieving a median retrieval time of just 0.65 seconds. As anticipated, both Option 0 and Option 1 exhibited similar retrieval times to those observed in the full data item mode, as they returned full data items in both modes. Specifically, in the specific parameter mode, the retrieval times were 1.08 seconds for Option 0 (using DynamoDB) and 10.68 seconds for Option 1 (using Kinesis Data Streams).

Table V summarizes numbers of requests needed to be performed to retrieve long recent data. The reason behind the big difference in the number of requests needed to be performed to retrieve long recent data from Kinesis Data Streams is the same as in retrieval of short recent data.

The median numbers of retrieved data items in long recent data retrievals are summarized in Table VI. The values in the table differ from the expected number of retrieved data items,

TABLE VI
MEDIAN NUMBERS OF RETRIEVED DATA ITEMS FOR LONG RECENT DATA RETRIEVALS

	Option 0 *DynamoDB*	Option 1 *Kinesis*	Option 2 *Timestream*
full data items	1792	50473	1793
specific parameter	1792	50456	1792

which was 1800, due to the same reasons as in recent data retrievals - missing the most recent data item and occasionally not created data item by device.

C. Historical data retrieval

In contrast to recent data retrievals, where data in Option 1 was obtained from Kinesis Data Streams, historical data retrievals fetch data from the S3 service in that option. The retrieval times of historical data are summarized in Figure 4.

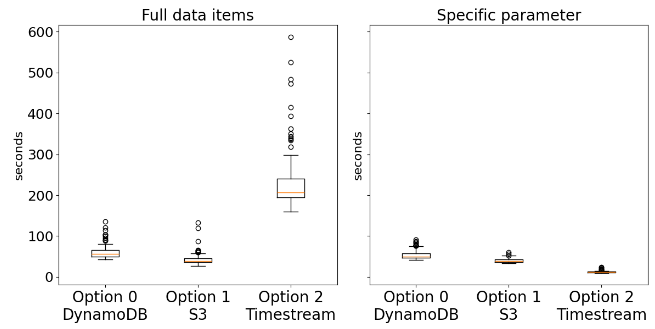

Fig. 4. Historical data retrieval times

In historical data retrieval, the shortest retrieval times were achieved by Option 1 (using S3), with a median retrieval time of 38.34 seconds. The next best option is Option 0 (using DynamoDB), which has a median retrieval time of 56.87 seconds. The slowest option is Option 2 (using Timestream), with a median retrieval time of 207.35 seconds.

In the specific parameter retrieval mode, Option 2 (using Timestream) performs the best with a retrieval time of only 11.681 seconds. Options 0 (using DynamoDB) and Option 1 (using S3) achieved similar results to those in the full data item retrieval mode because these options do not support querying specific parameters. The median retrieval times of Option 0 and Option 1 in specific parameter mode were 48.98 and 38.17 respectively.

Table VII summarizes the number of requests needed to be performed to retrieve historical data. There is a visible difference in number of requests performed to retrieve data from Timestream between full data item and specific parameter modes. This is attributed to the Timestream's support of querying specific parameter. This factor is also linked to improvement in Timestream's query times between those two modes.

The median numbers of data items retrieved in historical data retrievals are summarized in Table VIII. The expected number of data items was 86400. The values in the table differ

TABLE VII
NUMBER OF REQUESTS PERFORMED TO RETRIEVE HISTORICAL DATA

	Option 0 *DynamoDB*	Option 1 *S3*	Option 2 *Timestream*
full data items	32	120 - 121	255
spec parameters	32	120 - 121	11

TABLE VIII
MEDIAN NUMBERS OF RETRIEVED DATA ITEMS IN HISTORICAL DATA RETRIEVALS

	Option 0 *DynamoDB*	Option 1 *S3*	Option 2 *Timestream*
full data items	86241	86242	86241
spec parameters	86241	86242	86241

due to occasionally not created data item by a device during specific second. Retrieval by time from S3 is possible only with granularity of one hour, but this limitation did not affect this result as the queried time segment began at the whole hour and ended by the last second of an hour. Therefore data from full hours was retrieved.

D. Result summary

The best-performing option for retrieving full data items was Option 0, which used DynamoDB. However, in historical data retrievals, it was outperformed by Option 1, which utilized the S3 service. Based on the numbers of requests required for recent data retrieval and the associated retrieval times, it can be concluded that DynamoDB can return more data items in a single response and that executing a single query request is faster compared to Timestream. On the other hand, in all scenarios in which query options were needed, Timestream demonstrated the best performance as its support of querying specific parameter reduced response size.

The poor performance of Kinesis Data Streams is primarily due to the high number of required requests and its inability to query data from specific devices, resulting in an unnecessarily large number of returned items.

The observed qualitative measures are summarized in Table IX. In both use cases, for recent data and historical data retrieval, Option 1 could query data within a specific time range only partially. When retrieving data in from Kinesis Data Streams (Option 1), additional requests were made until an item outside the desired time range was returned, leading to more recent data items being retrieved than expected. Retrieval from S3 from specific time range was possible only with hourly granularity.

All options except Option 1 in recent data retrieval, where data was retrieved from Kinesis Data Streams, supported retrieval of data from specific device. In case of Kinesis Data Streams, data of all devices sharing shard were returned. Shards are assigned to devices by service based on device identifier used as a partition key.

Only Option 2 using Timestream supported retrieval of specific parameter. The format and structure of returned data varied across services. DynamoDB and Kinesis Data Streams

TABLE IX
QUALITATIVE MEASURES

Recent data			
	Option 0 *DynamoDB*	Option 1 *Kinesis*	Option 2 *Timestream*
query time range	yes	partially	yes
query device	yes	partially	yes
query parameters	no	no	yes
record per	data item	data item	parameter
config. adjustment	yes	no	no

Historical data			
	Option 0 *DynamoDB*	Option 1 *S3*	Option 2 *Timestream*
query time range	yes	partially	yes
query device	yes	yes	yes
query parameters	no	no	yes
record per	data item	data item	parameter
config. adjustment	yes	no	no

returned one data item per record, while S3 returned data objects with each line representing a data item. Timestream, on the other hand, returned one record per parameter, requiring multiple records to be grouped locally to reconstruct each data item. Only in case of DynamoDB there was a needed to adjust service configuration to handle changes in read or write workload.

V. Discussion

This section discusses the results, compares them with related literature, and provides recommendations. Unlike existing studies [4], which used a virtual machine to run databases, this study provisions services directly from the AWS cloud provider. This approach makes the comparison more representative of real cloud-based systems and limits other factors that could potentially affect the results. As shown in [21], data trasnfer rate in VM is higher than in serverless services, such as AWS Dynamo and Amazon S3.

A. Evaluation of storage system architecture options

This subsection discusses evaluation of IoT data storage system architectures included in the comparison.

1) Evaluation of Option 0: The data retrieval times recorded during the experiment did not align with results in [3] where DynamoDB showed poor performance and was outperformed by a specialized time-series database, in their case InfluxDB. In our experiment, in the retrieval of the full data items, DynamoDB demonstrated shorter data retrieval times than specialized time-series database offered by AWS.

The poor performance of DynamoDB compared to specialized time-series databases was also described in [14]. In this case it was Timestream. However, the authors pointed out that the poor DynamoDB performance in their case may have been caused by slow scan operations over multiple nodes. Our experiment which used faster query operation, that retrieves data from a single node based on provided partition key [16], proved their assumption to be correct.

Option 0, which uses DynamoDB, demonstrated to be the best among the evaluated options for retrieving recent data when full data items are needed. In full data item retrieval mode, this option was only outperformed by Option 1 in the retrieval of long and older segments of data. Option 1 retrieved data from a data lake built on the S3 service.

However, DynamoDB with our configuration suffered from the lack of extensive query options in cases when only specific parameter needed to be retrieved.

Our conclusion is that the authors of [3] and [14] did not fully leverage the advantages of DynamoDB. Their design choices for the DynamoDB table may not have been optimal, nor did they evaluate it in use cases best suited to DynamoDB's strengths, even within IoT applications.

In conclusion, DynamoDB's performance is highly dependent on its configuration and alignment with the intended use case. For optimal performance with time-series data, the timestamp should serve as a sort key, while the partition key should be carefully chosen to prevent data retrievals from multiple nodes. Ideally, the DynamoDB table should be structured to prioritize query operations, minimizing the need for scans. Query operations retrieve data by partition key, whereas scan operations involve scanning across all nodes (partitions) [16].

2) Evaluation of Option 1: The experiment in this study demonstrated possibility of retrieving recent data directly from Kinesis Data Streams. However, the retrieval times were high even for short time segments of time-series data. This was attributed to inability to query data from specific single device and data items of all devices that shared the same shard were retrieved. Therefore, approach that used a dedicated data store allowing efficient and quick access for real-time data used in [2] is recommended. An alternative solution could be a dedicated data pipelines for real-time data and batch data as in pattern 1 in [15].

On the other hand, choosing S3 as permanent storage for historical data was proven to be a good choice as it demonstrated the best performance in retrieval of large volumes of raw data.

Overall, this option in the current configuration did not prove to be suitable option for storing IoT data due to poor performance for real-time applications.

3) Evaluation of Option 2: A specialized time-series database Timestream, specifically optimized for handling time-series data, did not deliver the expected results when retrieving full data items. In all full data item retrieval scenarios, it was outperformed by at least one other option. The worst performance was demonstrated in the retrieval of long data sets of full data items (historical) data where it demonstrated multiple times slower performance than other options.

However, in all scenarios requiring only specific parameter to be queried, Timestream demonstrated its advantages, consistently outperforming other options. These benefits would likely be even more pronounced if time-series functions, such as interpolation, which are built into the Timestream service [22], were considered, as they reduce the processing time and resources needed locally to achieve the desired results.

For applications that require better performance in retrieving large sets of historical data, using the UNLOAD [23] command to export data to S3 should be considered, as this S3 service has demonstrated superior performance in such scenarios.

An additional benefit of Timestream is its strong support as a data source for popular analytics and visualization platforms like Grafana [24] and other real-time tools [25], [26].

B. Suggested alternative IoT Data Storage System Architectures

In this subsection, based on the found shortcomings of evaluated solutions and related literature, we suggest alternative solutions possibly fixing the found shortcomings.

1) Alternative solution 1: Poor performance of Option 1 in the retrieval of recent data could be fixed by using a dedicated data store for real-time data as in [2]. Depending on the project specifics, DynamoDB or Timestream services could be a good choices to serve as a data store. Figure 5 pictures Option 1 extended with Timestream service used as a data store.

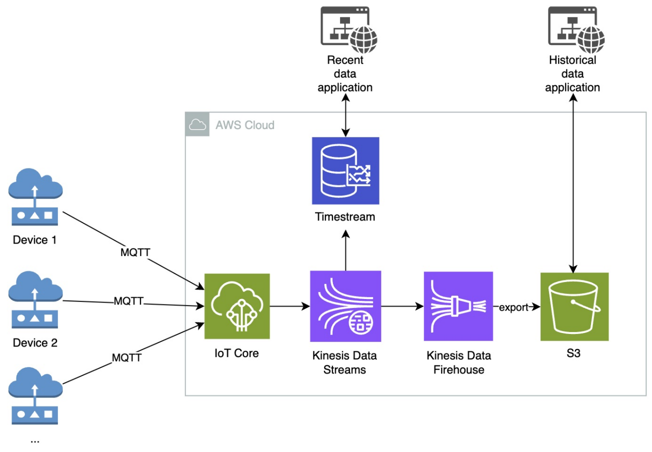

Fig. 5. Alternative 1: Option 1 with data store

2) Alternative solution 2: : Long retrieval times of full data items in Option 2 could be shortened by periodically unloading [23] data from Timestream to S3 service (data lake) which was demonstrated to be suitable for retrieval of big volumes of raw data. The architecture is pictured in Figure 6.

Setting Timestream's data retention properties [27] can achieve automatic data removal from the service after specific time passed from its arrival in the service and it is already unloaded to data lake. Furthermore, this can achieve cost saving as S3 is cheaper option for long term data storage than Timestream [28], [29].

C. Cost estimation and analysis

Initially, this study included a cost estimation for the AWS services used in the evaluated data storage system architectures. However, due to the lack of resources needed to verify the assumptions, on which the estimations were based, this part was excluded from the final version of the paper. The

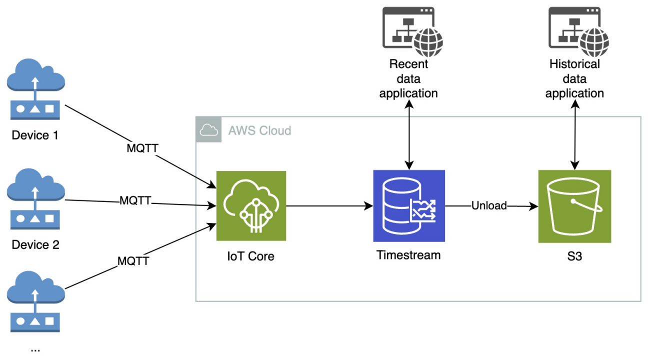

Fig. 6. Alternative 2: Option 2 with data lake

cost estimation and analysis could be detailed in a separate paper, as it is a sufficiently complex topic on its own. To verify the assumptions used in the cost estimation, we recommend provisioning the services form AWS and inspect the generated costs in AWS Cost Explorer [30].

Here is a couple of the unknowns needed to be answered to correctly estimate the cost of AWS services:

- data size calculation: As the base for the cost calculation for many services [28], [29], [31] data size is used. However, it was observed that each service stores data differently, resulting in variations in the size of the same data across services. It was unclear how to account for the difference between the size of the raw data and the size of the data stored in a service in the cost estimation. The same was true when data was retrieved and how the format, and therefore data size, of data retrieved from a specific service affects the data-size-based costs calculation.
- Calculation of the number of consumed units: The data retrieval cost of some services also depends on the consumed unit. As an example, in 2024, AWS introduced the Timestream Compute Unit (TCU) based on which, Timestream Cost is calculated [32]. TCU replaced data retrieval calculation which was based on the amount of scanned data. Precise estimation of needed TCU units, without running the service with given workload, was proven challenging or even impossible.

VI. Conclusion

This study identified the advantages and specific use-cases of the Key-value NoSQL database service DynamoDB and the specialized time-series database Timestream in IoT data storage applications. Timestream is preferable for applications that benefit from extensive query options, built-in time-series functions, and integration with real-time data analytics and visualization platforms like Grafana. DynamoDB provides the best performance for storing and retrieving recent data items, while S3 is ideal for storing large volumes of raw data that can be easily retrieved for data analysis. Kinesis Data Streams was found unsuitable as a direct source for real-time data retrieval using the boto3 AWS SDK for Python.

The results of this study can aid in designing future IoT data storage system architectures and selecting suitable AWS services. However, this study only compared three options, so none can be definitively identified as the best possible choice. Additionally, cost and other relevant metrics were not measured or taken into account.

Based on the results, alternative architectures that could potentially provide good performance for both real-time and batch data retrieval were suggested. Future work could involve studying the performance of alternative architectures and comparing them using additional metrics, such as cost.

When designing IoT data storage system architectures, choices should leverage the design and optimization of different AWS services to achieve desired performance. Specifically, horizontally scalable data storage services should be designed to minimize data operations across multiple nodes where data is stored.

Acknowledgments

This study was supported by Business Finland (6909/31/2022) and the Research Council of Finland (1346370, 1364223).

References

[1] S. Madakam, R. Ramaswamy, and S. Tripathi, "Internet of things (iot): A literature review," *Journal of Computer and Communications*, vol. 3, no. 5, pp. 164–173, 2015.

[2] K. Dineva and T. Atanasova, "Design of scalable iot architecture based on aws for smart livestock," *Animals*, vol. 11, no. 9, p. 2697, 2021. [Online]. Available: https://www.mdpi.com/2076-2615/11/9/2697

[3] P. John, J. Hynek, T. Hruska, and M. Valny, "Application of time series database for iot smart city platform," in *2023 Smart City Symposium Prague (SCSP)*, 2023, pp. 1–6.

[4] L. Calderoni, D. Maio, and L. Tullini, "Benchmarking cloud providers on serverless iot back-end infrastructures," *IEEE Internet of Things Journal*, vol. 9, no. 16, pp. 15 255–15 269, 2022.

[5] A.-E. Bouaouad, A. Cherradi, S. Assoul, and N. Souissi, "The key layers of iot architecture," in *2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech)*, 2020, pp. 1–4.

[6] L. F. Mendes, P. A. C. Aguilar, and C. I. M. Bezerra, "Software architecture for iot-based indoor positioning systems for ambient assisted living," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, 2023, pp. 93–104.

[7] H. Muccini and M. T. Moghaddam, "Iot architectural styles: A systematic mapping study," in *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*. Springer, 2018, pp. 68–85.

[8] A. Taivalsaari and T. Mikkonen, "A taxonomy of iot client architectures," *IEEE Software*, vol. 35, no. 3, pp. 83–88, 2018.

[9] A. Ali, S. Naeem, S. Anam, and M. Ahmed, "A state of art survey for big data processing and nosql database architecture," *International Journal of Computing and Digital Systems*, vol. 14, pp. 2210–142X, 05 2023.

[10] W. Ali, M. Majeed, A. Raza, and M. U. Shafique, "Comparison between sql and nosql databases and their relationship with big data analytics," *Asian Journal of Computer Science and Information Technology*, vol. 4, pp. 1–10, 10 2019.

[11] R. Cattell, "Scalable sql and nosql data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[12] D. Namiot, "Time series databases." *DAMDID/RCDL*, vol. 1536, pp. 132–137, 2015.

[13] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and influxdb," *Studienarbeit, Université Libre de Bruxelles*, vol. 12, pp. 1–44, 2017.

[14] B. Singh, R. Martyr, T. Medland, J. Astin, G. Hunter, and J.-C. Nebel, "Cloud based evaluation of databases for stock market data," *Journal of Cloud Computing*, vol. 11, no. 1, p. 53, 2022.

[15] A. Hakme, R. Gupta, and U. Kalaspurkar, "7 patterns for iot data ingestion and visualization- how to decide what works best for your use case," https://aws.amazon.com/blogs/iot/7-patterns-for-iot-data-ingestion-and-visualization-how-to-decide-what-works-best-for-your-use-case/, Accessed on 30th October 2024.

[16] "Best practices for querying and scanning data in dynamodb," https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-scan.html, Accessed on 1st November 2024.

[17] "Managing throughput capacity automatically with dynamodb auto scaling," https://docs.aws.amazon.com/amazondynamodb /latest/developerguide/AutoScaling.html, Accessed on 12th November 2024.

[18] D. Rohal, "Aws timeseries experiment," https://github.com/drohal3/aq-data-experiment , Accessed on 17th November 2024.

[19] "Terraform," https://developer.hashicorp.com/terraform , Accessed on 8th August 2024.

[20] "Aws sdk for python (boto3)," https://aws.amazon.com/sdk-for-python/ , Accessed on 17th April 2024.

[21] S. C. Palepu, D. Chahal, M. Ramesh, and R. Singhal, "Benchmarking the data layer across serverless platforms," in *Proceedings of the 2nd Workshop on High Performance Serverless Computing*. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–7. [Online]. Available: https://doi.org/10.1145/3526060.3535460

[22] "Interpolation functions," https://docs.aws.amazon.com/timestream/latest/developerguide/timeseries-specific-constructs.functions.interpolation.html, Accessed on 13th November 2024.

[23] "Unload," https://docs.aws.amazon.com/timestream/latest/developerguide/supported-sql-constructs.UNLOAD.html, Accessed on 13th November 2024.

[24] "Grafana," https://grafana.com/, Accessed on 24th October 2024.

[25] "Amazon timestream plugin for grafana," https://grafana.com/grafana/plugins/grafana-timestream-datasource/, Accessed on 24th October 2024.

[26] "What is amazon timestream for liveanalytics?" https://docs.aws.amazon.com/timestream/latest/developerguide/what-is-timestream.html, Accessed on 24th October 2024.

[27] "Retention properties," https://docs.aws.amazon.com/timestream/latest/developerguide/API_RetentionProperties.html, Accessed on 13th November 2024.

[28] "Amazon timestream pricing," https://aws.amazon.com/timestream/pricing/, Accessed on 24th October 2024.

[29] "Amazon s3 pricing," https://aws.amazon.com/s3/pricing/, Accessed on 24th October 2024.

[30] "Aws cost explorer features," https://aws.amazon.com/aws-cost-management/aws-cost-explorer/features/ , Accessed on 14th November 2024.

[31] "Aws pricing," https://aws.amazon.com/pricing, Accessed on 14th November 2024.

[32] "Announcing timestream compute unit (tcu) for amazon timestream for liveanalytics," https://aws.amazon.com/about-aws/whats-new/2024/04/tcu-amazon-timestream-liveanalytics/, Accessed on 14th November 2024.