DATA PIPELINING FOR INCOMING ASYNCHRONOUS STREAM

Project report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering

By

Aishani Pachauri (191393)

Under the supervision of

Dr. Rajni Mohana

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Certificate

I hereby declare that the work presented in this report entitled “Data Pipelining for Incoming Synchronous Stream” in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from January 2023 to May 2023 under the supervision of Mr.Aishwarya Agrawal, Software lead @Electorq Tech. The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Aishani Pachauri,

191393

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Rajni Mohana

Associate Professor
Department of Computer Science & Engineering
and Information Technology
Dated:

Plagiarism Certificate

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: .....

Type of Document (Tick): PhD Thesis M.Tech Dissertation/ Report B.Tech Project Report Paper

Name: _____ **Department:** _____ **Enrolment No** _____

Contact No. _____**E-mail.** _____

Name of the Supervisor: _____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): _____

_____

_____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

Total No. of Pages =

Total No. of Preliminary pages =

Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ……..(%). Therefore, we

are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

<table>
<tr><th>Copy Received on</th><th>Excluded</th><th>Similarity Index
(%)</th><th colspan="2">Generated Plagiarism Report Details
(Title, Abstract & Chapters)</th></tr>
<tr><td></td><td rowspan="4">All Preliminary
Pages
Bibliography/Ima
ges/Quotes
14 Words String</td><td rowspan="3"></td><td>Word Counts</td><td></td></tr>
<tr><td>Report Generated on</td><td>Character Counts</td><td></td></tr>
<tr><td rowspan="2"></td><td>Submission ID</td><td>Total Pages Scanned</td><td></td></tr>
<tr><td></td><td>File Size</td><td></td></tr>
</table>

Checked by

Name & Signature **Librarian**

……

……

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File)

through the supervisor at plagcheck.juit@gmail.com

ACKNOWLEDGEMENT

First, I express my heartiest thanks and gratefulness to Lord Shiva for His divine blessing to make it possible to complete the project work successfully.

I am really grateful and wish my profound indebtedness to **Dr. Rajni Mohana,** Associate Professor, Department of CSE & IT, Jaypee University of Information Technology, Waknaghat. Deep Knowledge & keen interest of my supervisor in the field of "Cloud Computing" to carry out this project. Her endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, and reading many inferior drafts and correcting them at all stages have made it possible to complete this project.

I would also generously welcome my friend Achyut Tiwari and each one of those individuals who have supported me straightforwardly or in a roundabout way throughout the project timeline. I would also like to thank various staff individuals, both educating and non-instructing, which have extended their help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of my parents.

Project Group No. 95

Name: Aishani Pachauri
Roll No: 191393

Table of Content

List of Abbreviations

1. NoSQL: No Structured QueryLanguage
2. Dynamo Db: Dynamo Database
3. SQS: Simple Queue Service
4. KPI: Key Performance Index
5. IoT: Internet of things
6. QoS: Quality of Service

List of Figures

List of Graphs

List of Tables

Abstract

In the digital age, data has become one of the most valuable resources. With the proliferation of technology and the internet, we generate and store vast amounts of data every day. This data can come from a variety of sources, including social media, e-commerce transactions, and sensors in devices. By analyzing this data, businesses and organizations can gain valuable insights into customer behavior, market trends, and more. This information can then be used to make informed decisions and drive innovation. Data also plays a crucial role in areas such as scientific research and public policy. However, with the importance of data comes the responsibility to ensure that it is collected, stored, and used in an ethical and secure manner. To address this, ensuring a secure, well constructed data pipeline must be set up in order to take maximum benefits out of the data generated, which can be used in this specific business case. This data is further used to extract sufficient KPIs to project various indexes like ARR, Recurring Revenue, Monthly retention and Churn rate. For accurate estimation and projections, data must be accurately transmitted and handled to further carry out the analysis.

Chapter-1

INTRODUCTION

1.1 Introduction

In traditional server-based architecture, developers would need to provision and manage servers, handle load balancing, and ensure the availability and scalability of the infrastructure. However, with serverless computing, developers can focus on writing code and leave the infrastructure management to the cloud provider. [1]. Serverless backend services typically include services such as databases, storage, and message queues. These services are designed to scale automatically based on demand and are charged based on usage. For example, AWS offers serverless backend services such as Amazon S3 for storage, Amazon DynamoDB for NoSQL databases, and Amazon Simple Queue Service (SQS) for message queues.

Serverless compute services are services that allow developers to run code without provisioning or managing servers. These services include functions as a service (FaaS) platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions. With serverless compute services, developers can upload their code, specify triggers or events that will invoke the code, and let the cloud provider handle the rest. These services scale automatically based on demand, and users are charged based on the number of invocations and the amount of time their code runs.

Serverless backend and compute services offer several advantages over traditional server-based architecture. They allow developers to focus on writing code and developing applications rather than managing infrastructure. They are also highly scalable, and users only pay for what they use, making them a cost-effective solution for many types of applications.
[2,3]. Serverless backend and compute services are a type of cloud computing service where the cloud provider manages the infrastructure and automatically scales resources up or down based on demand. In recent years, serverless computing has emerged as a popular paradigm for building and deploying applications in the cloud. Amazon Web Services (AWS) has been at the forefront of this trend with its Lambda service, which allows developers to run code without provisioning or managing servers. However, running serverless functions in a virtualized environment can be resource-intensive and slow to start up. This is where AWS Firecracker comes in.[4]

AWS Lambda is a serverless compute service that allows developers to run code in response to events or triggers, such as HTTP requests, database updates, or file uploads. Lambda handles all the underlying infrastructure, automatically scaling resources up or down to match demand, and charging only for the compute time used. Lambda functions can be written in several programming languages, including Node.js, Python, Java, and C#. With the integration of Firecracker, Lambda can now run in a secure and lightweight microVM environment, providing fast and efficient execution of serverless functions.

Together, Firecracker and Lambda provide a powerful and efficient platform for building serverless applications in the cloud. Developers can write code in their preferred language, upload it to Lambda, and let the service handle the rest. With the integration of Firecracker, Lambda can now start up serverless functions in just a few milliseconds, allowing for near-instantaneous response times to events and triggers. This makes Lambda an ideal choice for building event-driven applications, microservices, and API backends that require fast and efficient execution.

A serverless architecture is a cloud computing model in which the cloud provider manages the infrastructure required to run applications, and the user only pays for the computing resources used, rather than for a fixed amount of server capacity. In a serverless architecture, the user does not need to provision or maintain servers, and instead, the cloud provider handles the allocation of resources and scaling of the application as needed. The name "serverless" is somewhat misleading, as servers are still used, but the user is shielded from the underlying infrastructure, allowing them to focus solely on the application logic. Serverless architectures often utilize Functions-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) to handle the application logic and data storage, respectively.

Bytebeam is an IoT (Internet of Things) provider that works with end-to-end solutions for connecting and managing devices, data, and applications. The company focuses on delivering IoT solutions across a range of industries, including agriculture, healthcare, logistics, and smart cities.

Bytebeam offers a range of services that include hardware design, software development, cloud-based platforms, and analytics. The company has expertise in developing custom hardware solutions, such as sensors, gateways, and control systems, to connect and manage IoT devices. They also offer software development services to build custom applications that can interact with these devices and collect data.

Bytebeam's cloud-based platforms provide secure and scalable solutions for managing and analyzing data from IoT devices. The platforms offer features such as device management, data analytics, and real-time monitoring, which allow customers to make informed decisions based on the data collected from their IoT devices. Overall, Bytebeam's services enable businesses to leverage the power of IoT to improve operational efficiency, reduce costs, and enhance customer experiences. Embedded code in IoT refers to the software code that runs on the microcontrollers or microprocessors embedded in IoT devices. These devices typically have limited computing resources and memory compared to general-purpose computers, so the code used for them needs to be highly optimized and efficient.

Embedded code is responsible for controlling and managing the hardware components of an IoT device, such as sensors, actuators, and communication modules. It interacts directly with the hardware through low-level interfaces and protocols to read and write data, perform computations, and make decisions based on the sensor readings. The embedded code in an IoT device can be developed using a variety of programming languages, such as C, C++, or Assembly language, depending on the specific hardware platform and the requirements of the application.

Embedded code plays a crucial role in IoT systems, as it is responsible for collecting, processing, and transmitting data from IoT devices to the cloud or other connected devices. Effective embedded code can help ensure that an IoT device operates efficiently, reliably, and securely.

To accomplish efficient horizontal scaling in case of interactive data handling, Lambda@Edge is a service provided by AWS (Amazon Web Services) that enables running serverless Lambda functions on the edge locations of the AWS global content delivery network (CDN), Amazon CloudFront.

These edge locations are distributed around the world, closer to end-users, enabling faster content delivery and reducing latency. Lambda@Edge allows developers to run code directly on these edge locations, which can be used to enhance the user experience by improving the performance of websites and web applications.This can be used for a variety of use cases, such as modifying HTTP responses, generating dynamic web content, and customizing authentication and authorization logic. The service is fully managed by AWS, so developers do not need to worry about infrastructure provisioning or scaling. These functions are triggered by CloudFront events such as viewer request, origin request, origin response, and viewer response, and they can be written in various programming languages such as Node.js, Python, Java, and C#.

1.2 Problem Statement

IoT applications such as those used for smart homes, smart cities, and smart healthcare must be able to adapt to changing user behavior, as users' actions may vary based on different factors. Additionally, many of these applications require timely responses. Consider a city that uses IoT sensors to monitor and manage its transportation system, including traffic flow, parking availability, and public transportation routes.

Internet of Things (IoT) is transforming many industries by enabling them to collect, process and analyze real-time data. One of the industries that IoT is having a significant impact on is the transportation industry. However, providing high-quality service to the citizens requires the IoT sensors to ensure a high level of quality of service (QoS).

QoS is a measure of the overall performance of a system, including its ability to deliver data reliably, quickly, and with low latency. In the context of IoT applications, QoS is critical because many applications require real-time data to be delivered in a timely manner. For instance, in a smart city transportation system, real-time data on traffic flow, parking availability, and public

transportation routes must be delivered quickly to enable efficient transportation systems.

To achieve high QoS, IoT sensors must use QoS-aware protocols and algorithms that prioritize the delivery of critical data over less critical data. Prioritizing the delivery of critical data ensures that real-time information is delivered to the transportation management system in a timely and reliable manner. For example, in a smart parking system, information on parking availability is critical, and it needs to be delivered in real-time. If the information is delayed or unreliable, it can cause congestion and increase pollution levels.

Table 1: IT sprint problem schema

Define the problem statement	
What is the problem?	The battery swap data is not handled properly and is missing or repeated
What are the possible solutions?	Tracking data from IoT's server and applying appropriate filters to post clean data to our database
What teams and systems will be impacted?	Electrical team working with IoT embedded code and tracking overall Kms traveled in one cycle of a battery

Moreover, IoT sensors must be able to adapt to changes in network conditions and traffic load to ensure optimal performance and QoS. For instance, in a smart transportation system, the amount of data that needs to be delivered varies depending on the time of day and the day of the week. During peak hours, there may be a higher demand for real-time data, and the network may be congested. In this case, the IoT sensors must be able to adapt to the changing network conditions and traffic load to ensure optimal performance and QoS.

Cost is another important parameter that needs to be considered when managing resources and allocating resources in IoT applications. In many cases, the cost of deploying and maintaining an IoT system can be significant. Therefore, it is essential to use cost-effective solutions that can deliver high QoS.

One approach to reducing the cost of IoT systems is to use edge computing. Edge computing enables data to be processed and analysed at the edge of the network, closer to the data source. By processing and analysing data at the edge of the network, the amount of data that needs to be transmitted over the network can be reduced, which can reduce the cost of the system. Moreover, edge computing can improve QoS by reducing latency and increasing reliability.

These IoT protocols must adapt to changes in network conditions and traffic load to ensure optimal performance and QoS. They must take into account the inflow of data and have sufficient mechanism for fault tolerance to avoid data loss. Moreover, critical error handling with both data at rest and data in transit is crucial in this particular use case.

In conclusion, IoT has a huge impact on the automotive industry and if handled through technically sound pipelines, IoT data extracted can serve as a gold mine of projections for a growing business.

Given the above application scenario, it is very vital to manage resources and efficiently collect data from third party server, decipher and store it.
In consideration of two vital parameters which are Cost and Quality of service.

1.3 Objectives

a) To propose a solution to handle asynchronous data coming from SQL based Bytebeam data center.

b) To produce a usable data format and storage pipeline while taking account of AWS Serverless constraints .

1.4 Proposed Methodology

In the rapidly evolving landscape of serverless architecture, the deployment of lambda based architecture combines performance, and security of complex and demanding IoT applications. With the exponential growth of data and the increasing complexity of applications, traditional centralized systems have become inadequate in handling large workloads and providing efficient processing. Distributed agents, on the other hand, can be deployed across the cloud-fog environment to improve the overall efficiency of the system.

The scalability of the system can be improved by deploying distributed agents that can handle large workloads effectively. The distribution of agents across the cloud-fog environment ensures that the load is distributed evenly, preventing any single agent from becoming overloaded. This, in turn, enables the system to process a large number of requests simultaneously, improving its scalability.

Furthermore, the deployment of distributed agents enables the system to continue operating even if one agent fails, enhancing reliability and reducing the risk of downtime. This approach also provides flexibility, as distributed agents can be designed to be highly adaptable to changes in workload or environmental conditions, enabling more efficient resource allocation and task delegation.

In addition to scalability and reliability, distributed agents can also enhance the performance of the system. By enabling faster and more efficient processing, distributed agents can improve the overall responsiveness of the system. The redundancy provided by the deployment of agents in multiple locations also improves the resilience of the system, ensuring that critical services remain available even if one part of the cloud-fog environment experiences a failure. This approach also enables the system to handle dynamic workloads more effectively, as this can be designed to adjust their processing capabilities in response to changing workload conditions.

Moreover, this deployment enhances the security of the system by internal security measures for the microVM environment for lambdas preventing unauthorized access and protecting sensitive data. By decentralizing the system, the risk of a single point of failure is reduced, making it more difficult for malicious actors to disrupt the system.

AWS Firecracker is an open-source virtualization technology that provides lightweight, secure, and fast microVMs (micro virtual machines) for running serverless functions. Firecracker is designed to be used as a building block for container and serverless platforms, providing secure and isolated execution environments for workloads. Firecracker has a minimal footprint, starting VMs in less than 125ms and consuming only a few MBs of memory, making it an ideal solution for running serverless functions.

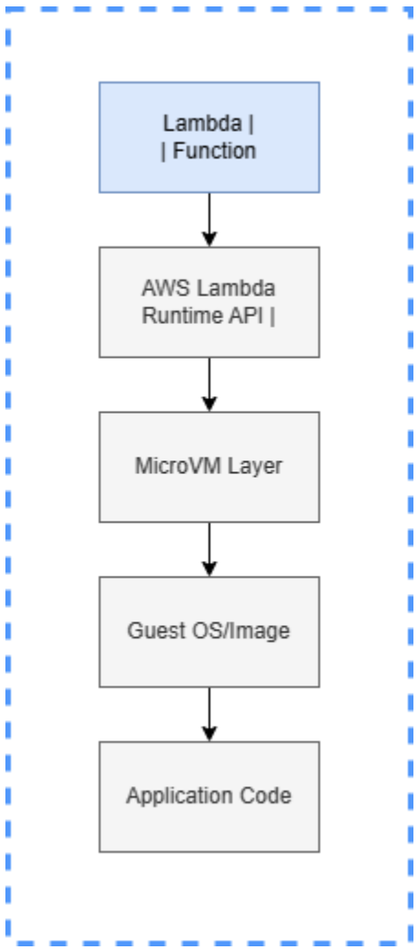


Figure 1: AWS Lambda architecture and underlying environment.

Hence we leverage lambda runtime execution and push it's limits to maximum threshold through vertical scaling via memory provision (epiphermal & runtime).

We configure the lambda in such a way that it utilizes each runtime execution that it creates which lives for 15 minutes and carries multiple invocations to avoid cold start. This helps us reduce the time of environment creation, MicroVM production to environment setup by container, significantly by 32.883% approximately.

Decreasing the INIT Duration in lambda creation paves a way to significantly reduce and process triggers to the particular function and reduce overall runtime duration and memory utilization. In this architecture, as in Fig. 1, the AWS Lambda function is executed in a container environment that includes the Lambda runtime API and a microVM layer.

The microVM layer is responsible for setting up and managing a lightweight virtual machine that isolates the Lambda function from other processes and provides a secure execution environment.

Within the microVM, a guest operating system or image is loaded, which provides a runtime environment for the Lambda function's application code. The application code is loaded and executed within the guest OS or image, and any output or results are returned to the Lambda runtime API for further processing and communication with other AWS services.

In summary, this particular deployment required data in such a format where it can be mapped to key-value stores that we use in our architecture which offers numerous benefits, including scalability, reliability, performance, and security. These benefits make distributed agents well-suited for complex and demanding applications, as they can enhance the efficiency and adaptability of the system. By leveraging the advantages of distributed agents, organizations can build more robust and scalable cloud-fog environments that can adapt to changing workloads and environmental conditions, while maintaining high levels of performance and security.

The overall approach can be analyzed through three different perspectives:

1. Data management and storage: Understanding various endpoints sending the data, the format in which we receive the data and cleaning the data to keep the relevant records in the form of key-value pairs in NoSQL based storage of AWS DynamoDB.

2. Cloud - IoT communication: Establishing a stable connection while securing the data in transit is quite essential here as the window of receiving the data is time based. One unit of data lost cannot be recovered until noticed through manual digging.

3. Lambda function internal optimization: Defining the optimal filters to get the desired data in return and understanding the bridge that the function is between the external data store to the organization's database.

Selection of IoT server was determined while considering multiple factors and understanding of IoT platform, including:

1. The specific service requirements of those devices and applications.
2. The scalability and capacity requirements of the platform.
3. The security and privacy requirements of the platform.
4. The cost of the platform.
5. The types of devices and applications that will be connected.

Service providers must understand required credentials and configure the platform to expose the required services. Service providers also need to consider how they will manage the IoT platform over time. This includes managing updates and patches, as well as monitoring platform performance and capacity.

ClickHouse is a high-performance columnar database management system (DBMS) developed by Yandex, which is designed for real-time analytics and processing of large volumes of data. It is open-source software and is widely used for data warehousing, business intelligence, and big data processing.

Figure 2: Column-Oriented Databases Work Better in the OLAP Scenario

Unlike traditional row-based databases, ClickHouse is a columnar database, which means that data is stored in columns rather than rows. This makes it highly optimized for analytical queries and data aggregation, as it can quickly scan and retrieve only the columns that are relevant to a particular query.
To filter and collect data as quickly as possible, ClickHouse was initially created as a prototype. A basic GROUP BY query does what is required to construct a typical analytical report. The ClickHouse team made a number of important choices that, when combined, allowed for the completion of this task:

Column-oriented storage: Although a report may only employ a handful of the hundreds or thousands of columns that make up source data, this is a common situation. To save money on costly disc read operations, the system must refrain from reading irrelevant columns.

Indexes: Resident memory Only the necessary columns and necessary row ranges of those columns may be read from ClickHouse data structures.

Execution of queries using vectorization: ClickHouse not only stores but also processes data in columns. This improves CPU cache utilisation and enables the use of SIMD CPU instructions.

Scalability: ClickHouse can employ every CPU core and disc that is available to process even a single query. not just on a single server but also across the entire cluster's CPUs and discs.

In real data, a column frequently contains the same, or not that many different, values for neighboring rows, hence grouping various values of the same column together typically results in superior compression ratios (compared to row-oriented systems). ClickHouse offers specialized codecs that can further compress data in addition to standard compression. OLAP stands for Online Analytical Processing, which is a technique used for performing complex multidimensional analysis of data in real-time.Furthermore, this report discusses the finalized delivered feature while presenting the methodology, corner cases and results of the testing phases. Consisting of dummy battery swaps and internal testing.

1.5 Organization

The report is organized as follows:

- Chapter-02 outlines the existing related work in the field of system design considering IoT service placement and server management connected to cloud infrastructure. It further presents the outputs which we eventually compare and discuss in this report.

- Chapter-03 puts forward the system that is formulated to cater the IoT -Data-Cloud connectivity pipeline and is designed to work so as to reduce latency and reduce data miss. This is where we cover the software requirement and optimization module that are considered.

- Chapter-04 puts forward the analysis of the results and data presentation to cater the definite business case in depth and also with content to existing solution.

- Finally, Chapter-05 presents the conclusion of this particular feature release with the application contribution and discussed future scope

Overall application of this solution will not only help in automating asset ownership transfer but also help in scaling and designing the IoT - Software interdependency through best practical system design paradigms realized from various use cases in existing organizations.

It is a category of software tools that allows users to analyze data from multiple perspectives and dimensions, providing a comprehensive view of business data.

Chapter-2

LITERATURE SURVEY

In this section the authors have covered various studies related to edge user allocation problems [8] and solutions pertaining to the same. One of the key challenges in fog computing is how to efficiently allocate resources to edge users accessing the execution environment. This is especially challenging when the number of data points fetched per second is large and dynamic. This depends on the number of interactions with IoT, which increases per month by an amplitude of 17x each month in this particular context.

Various resource allocation schemes have been proposed, but there is no clear consensus on which is the best approach. The edge user allocation problem is a problem that arises in the context of allocating users to edges in a network. The problem is to find a mapping of users to edges such that the sum of the weights of the edges allocated to each user is minimized. The problem is NP-hard [9].

IoT devices are becoming increasingly common, with many organizations using them to monitor and manage their operations. However, due to the large number of devices and the variety of services they offer, load-balancing IoT service placement is becoming a challenge. One approach to solving this problem is to use a software-defined network (SDN) controller to dynamically adjust the placement of IoT services based on the current load. This would allow the system to automatically adjust the placement of services as the number and type of devices change, and as the load on the system changes. Another approach is to use a central database that keeps track of the number and location of devices, as well as the load on each device. This database could then be used to determine the best placement of services based on the current load. Whichever approach is used, it is important to consider the trade-offs between flexibility and performance. For example, if the system is too flexible, it may take longer to find an optimal solution. On the other hand, if the system is not flexible enough, it may not be able to adapt to changes in the environment.

In [11], authors present a two-step resource management approach with the goal of using the fewest possible edge nodes while reducing the amount of time needed to deliver services. For each device, a pool of backup edge nodes and a home edge are first chosen. Finding the edge nodes that have the lowest latency between them and that device is their goal. The specified edge nodes are then used to host the necessary IoT services, ensuring the desired response time. A different project with the same objective as the ones listed in [11] and [10] has been proposed by researchers in

[12]. According to a backtrack search algorithm and related heuristics that serve the goal, the recommended mechanism selects locations. The authors of [13] have offered a conceptual framework for service placement for the edge-to-cloud system. Their objective is to increase edge node use while taking user constraints into account by using a genetic algorithm for optimization. In order to take advantage of Internet of Things nodes for IoT service execution, the authors introduce the concept of a fog cell, which is software that runs on IoT nodes. An edge-to-cloud control middleware that oversees the fog cells has also been introduced. Any associated fog cells or other control nodes are under the supervision of a fog orchestration control node. The latter enables IoT services to be managed separately from cloud nodes.

In the rapidly evolving field of fog computing, resource allocation is a major challenge that researchers have been addressing. One of the main issues in fog computing is to allocate resources to edge users efficiently. The edge user allocation problem arises in the context of mapping users to edges in a network. The objective is to minimize the sum of the weights of the edges allocated to each user. This problem is NP-hard and becomes increasingly challenging as the number of edge users is large and dynamic.

To address this problem, various resource allocation schemes have been proposed. However, there is no clear consensus on which approach is the most effective. For instance, one approach involves the use of a software-defined network (SDN) controller to dynamically adjust the placement of IoT services based on the current load. The SDN controller would allow the system to

automatically adjust the placement of services as the number and type of devices change, and as the load on the system changes. Another approach involves the use of a central database to keep track of the number and location of devices, as well as the load on each device. This database could then be used to determine the best placement of services based on the current load.

Regardless of the approach used, it is important to consider the trade-offs between flexibility and performance. For example, a system that is too flexible may take longer to find an optimal solution, while a system that is not flexible enough may not be able to adapt to changes in the environment. Therefore, researchers have been exploring different solutions to this problem, such as QoS-aware service allocation, multi-dimensional knapsack problem, two-step resource management, and genetic algorithms.

In computing networks, there may be instances where certain nodes experience low levels of activity while others are overwhelmed with the entire network load. This load imbalance can lead to various issues such as system and network failures, increased energy consumption, and longer execution times. To prevent such problems, load balancing is essential to distribute the load evenly across all resources based on their capacity. This ensures that no resources are underutilized or overburdened in a fog environment. Load balancing is also necessary for cloud data centres to ensure efficient workload distribution, optimal functioning, and prevention of overload and deadlock issues.

In one study, researchers introduced a QoS-aware service allocation for fog environments that aimed to minimize overall service execution delay and the load on the edge nodes. They used a multi-dimensional knapsack problem to describe this goal. In another study, a two-step resource management approach was proposed with the goal of using the fewest possible edge nodes while reducing the amount of time needed to deliver services. A pool of backup edge nodes and a home edge were first chosen for each device, and then the edge nodes with the lowest latency between them and that device were selected. These nodes were used to host the necessary IoT services to ensure the desired response time. Another project with similar objectives proposed a backtrack search algorithm and related heuristics to select locations.

Researchers also proposed a conceptual framework for service placement in the edge-to-cloud system, with the aim of increasing edge node usage while taking user constraints into account.

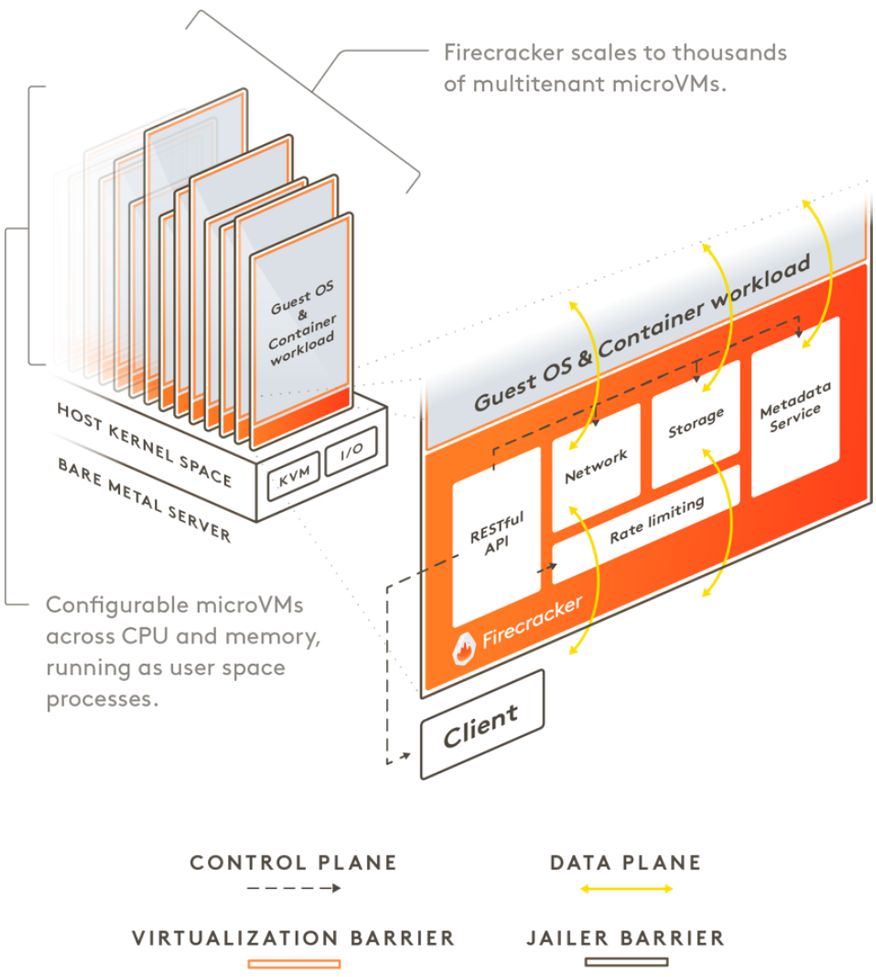


Figure 3. MicroVM containerization

which helped to manage IoT services separately from cloud nodes. Additionally, the concept of a fog cell, which is software that runs on IoT nodes, was introduced to take advantage of IoT nodes for IoT service execution. An edge-to-cloud control middleware that oversees the fog cells was also proposed, along with a fog orchestration control node that supervised any associated fog cells or other control nodes.

Author in [14]'s main goal is to increase the number of edge node-served services while maintaining QoS standards such as response time. They employ an algorithm to overcome the issue that makes use of validation, rounding, and relaxation. Authors in [15] offer a service placement strategy that maximises the amount of services assigned to edge nodes, similar to the earlier efforts [16],

[17]. The suggested method uses context data from the edge nodes, such as location, response time, and resource consumption, to distribute services. Workload distribution is defined by [18] as an interaction between edge-to-cloud nodes. Investigated and roughly resolved is the trade-off between power usage and transmission delay in the interaction. A relevant framework for understanding the cooperation between edge-to-cloud nodes is provided by simulation and numerical results. For a three-layer fog-cloud architecture made up of the fog device, fog server, and cloud layers, authors in [19] present for resource allocation. The processing time, bandwidth, and reaction time of the available resources are ranked according to three factors in order to address the time constraints imposed by dynamic user behaviour in resource provisioning. These resources are then distributed in a hierarchical and hybrid way according to the requests that were received. DRAM is a different load-balancing resource allocation mechanism that authors have [20] offered. DRAM uses service migration after allocating network resources statically to create a dynamically balanced workload across edge nodes.

In order to supply IoT services, authors in [21] create an Integer Linear Programming (ILP) problem that balances two goals: minimising deployment cost (which includes the costs of computation, memory, and data transfer) and raising service acceptance rate. Greedy Randomised Adaptive Search techniques [22], which iteratively minimise the provisioning cost while load-

balancing networked nodes, are used in the suggested solution. In order to reduce the processing time of compute tasks in fiber-wireless enhanced vehicle edge computing networks, authors in [23] suggest a task offloading architecture. Two strategies based on software-defined networking and game theory are given to achieve the load-balancing of the computation resources at the edge servers. For each vehicle to successfully complete its computation task, these schemes, namely a nearest offloading algorithm and a predictive offloading algorithm, optimise the offloading decisions for local execution, offloading to a Multi-access Edge Computing (MEC) server connected to roadside units, and offloading to a remote cloud server. In the table 1, the authors have made a drawn a comparison to distinguish existing work in different categories including QoS, Load Balancing, Techniques employed and whether the Distributed Network as used or not.

The emergence of the Internet of Things (IoT) and the ubiquitous adoption of smart devices have transformed virtually every industry, making it imperative to provide advanced services that are scalable, reliable, and high-performing. The integration of IoT and Cloud Computing (CC) has given rise to cloud IoT, a new paradigm that aggregates, stores, and processes IoT-generated data. While cloud IoT brings immense opportunities, it is also constrained by bandwidth, latency, and connectivity issues. This has led to the development of Edge and Fog Computing (FC), where computing and storage resources are located at the edges, closer to the source of data. The hierarchical and collaborative edge-fog-cloud architecture brings significant benefits, as it enables the distribution of computation and intelligence, including AI, ML, and big data analytics, to achieve optimal solutions while satisfying constraints such as the delay-energy trade-off.

Despite the advantages of edge-fog-cloud computing, its implementation poses several challenges, including design, deployment, and evaluation. To provide a comprehensive understanding of this paradigm, this paper presents an in-depth tutorial and discusses the main requirements, state-of-the-art reference architectures, building blocks, components, protocols, applications, and other similar computing paradigms. The paper also presents a holistic reference

architecture for edge-fog-cloud IoT, discussing the major corresponding design and deployment considerations, including service models, infrastructure design, provisioning, resource allocation, offloading, service migration, performance evaluation, and security concerns.
In addition to these considerations, the paper also explores the role of privacy-preserving, distributed, and collaborative analytics, as well as the interaction between edge, fog, and cloud computing. Finally, the paper reviews the main challenges in the field of edge-fog-cloud computing that need to be tackled to realize the full potential of IoT.

Several studies have investigated the integration of IoT and CC, resulting in the development of cloud IoT. However, cloud IoT faces challenges such as latency, connectivity, and bandwidth. To overcome these challenges, edge and fog computing have emerged, offering distributed computing and storage resources closer to the data source. This hierarchical architecture enables the distribution of computation and intelligence, leading to optimal solutions while satisfying constraints such as the delay-energy trade-off.

Despite the benefits of edge-fog-cloud computing, several challenges remain, including design, deployment, and evaluation. This paper provides a comprehensive insight into the paradigm by presenting a tutorial and discussing various aspects, including reference architectures, building blocks, components, and protocols. Additionally, it explores the role of privacy-preserving, distributed, and collaborative analytics, as well as the interaction between edge, fog, and cloud computing. Finally, the paper reviews the main challenges in the field of edge-fog-cloud computing that need to be addressed to fully realize the potential of IoT.

In conclusion, this paper [24] presents a thorough literature review of edge-fog-cloud computing, highlighting the benefits and challenges of this paradigm. It offers a comprehensive understanding of the underlying technologies and presents a holistic reference architecture for edge-fog-cloud IoT. By discussing the major design and deployment considerations and exploring the role of privacy-preserving, distributed, and collaborative analytics, this paper provides opportunities for more holistic studies and accelerates knowledge acquisition in the field.

The paper identifies the dynamic service placement problem, which addresses the adaptive configuration of application services at edge servers to facilitate end-users and those devices that need to offload computation tasks. The paper presents a systematic literature review of existing dynamic service placement methods for MEC environments from networking, middleware, applications, and evaluation perspectives. The review reveals research gaps in the big picture and identifies eight research directions that researchers follow.

With the advent of cloud-based applications such as mixed reality, online gaming, and healthcare, there is a need for efficient infrastructure management to provide a cloud-like environment for end-users. MEC extends the cloud computing paradigm and leverages servers near end-users at the network edge to provide a cloud-like environment, but the optimum placement of services on edge servers plays a crucial role in the performance of such service-based applications.

The review [25] then investigates dynamic service placement methods from a middleware viewpoint, which includes different service packaging technologies and their trade-offs. The review categorizes the research objectives into six main classes, proposing a taxonomy of design objectives for the dynamic service placement problem. The paper also introduces the applications that can take advantage of dynamic service placement and investigates the evaluation environments used to validate the solutions, including simulators and testbeds. Finally, the paper compiles a list of open issues and challenges categorized by various viewpoints. Overall, this literature review provides a comprehensive insight into the dynamic service placement problem in MEC environments and identifies future research directions.

QoS is the primary concern in dynamic service placement methods from an application viewpoint, including QoS levels and factors. Application QoS can typically be categorised into three levels . The first level is guaranteed services (hard QoS) that have strict hard real-time QoS guarantees. This level is suitable for safety-critical applications such as remote surgery. The second level is soft QoS that does not require hard real-time guarantees but needs to reconfigure

and replace failed services. Finally, the last level is the best effort, where there are no guarantees when a service fails. According to the surveyed papers, time-related QoS factors receive much attention compared to others. Some of these factors, such as application response time and user-perceived latency, are applied to both soft QoS and hard QoS. Other factors such as the worst application completion time and the number of applications in outage focus on hard QoS. The next group of QoS factors concentrates on throughput and resource utilisation, namely processing, network, and energy resources. It shows how effectively the edge nodes are being used and that the load is being spread evenly across them, and no one edge node is overloaded. Modern applications, such as augmented reality and autonomous vehicles, have massive network throughput. Energy efficiency is a concern to both users and edge infrastructure providers. Security is another QoS factor that is addressed in a few work. With the new legislation, such as GDPR, privacy concerns are becoming as essential as other security factors when developing a service placement method. MEC enables the processing of exabytes of data near where it is required and generated. Such proximity benefits applications from different domains as they can address challenges regarding data volume, interoperability, and latency. In the following, we review these application domains that can benefit from dynamic service placement mechanisms to address these challenges effectively and efficiently use the available resources in MEC architectures.

The agent-based approach is a real-time strategy that involves assigning tasks to servers based on their current load, as determined by their respective agents.
[45] proposed an agent-based automated service composition (A2SC) technique for resource provisioning in cloud computing, with a focus on reducing virtual machine costs and ensuring equal resource distribution across four data centres with different platforms. They employed Java to obtain experimental results and aimed to provide efficient service allocation in the data centres.

[26] proposed a multi-agent-based offloading technique for mobile fog computing, which uses reinforcement learning to minimize service delivery latency to mobile users. The mobile codes are deployed on geographically distributed mobile fogs, with agents serving as entities that have prior

In [1,2, 3, 4, 5, 12, 13,43,44], the authors note that Both QoS and load balancing are not considered except in one case of [44], Meanwhile the technique and simulations are very diverse.

knowledge of the environment and learn from it. The goal is to reduce execution time and improve mobile user access to services, with simulation results obtained using OmNet++.

[27] developed an agent-based task assignment approach for load balancing in cloud computing, incorporating principles of fair competition and dynamic adjustment for task allocation to improve resource allocation and utilization. They used CloudSim to obtain simulation results, which showed an increase in processing time with this technique.

Time aware system is based on containers, and it helps to reduce the service access time from sender to receiver by equally distributing the load. Javaid et al. [33] proposed a cuckoo search load balancing algorithm, which uses a combination of Levy walk distribution and flower pollination to optimize the response time and processing time of fog and cloud environments. Cost is also considered, and efforts are made to reduce the cost of data transfer, microgrids, VMs, and the total cost. Khattak et al. [34] proposed a fog-cloud server-based architecture to achieve proper utilization of all resources in E-healthcare. They aimed to distribute equal load among all servers by shifting the load from overloaded servers to the ones having less load. Parameters like latency, load balancing, QoS, and bandwidth were considered, and simulation results were obtained using iFogSim.

In 2020, Talaat et al. [35] proposed a resource allocation-based load balancing approach that uses reinforcement learning to handle incoming requests by measuring server loads. The workload is distributed among all available resources for their appropriate utilization, and a three-layer fog-cloud-based architecture is proposed for health care. Kaur et al. [36] proposed a load balancing approach based on the equal distribution of workload in a three-tier architecture of fog-cloud to reduce energy consumption, cost, and processing time in the fog-cloud environment. The proposed approaches are implemented and compared with existing round-robin and throttled algorithms using a cloud analyst simulation tool. Bhatia et al. [37] proposed a quantumized approach to task scheduling in the fog environment, which distributes the workload among all fog nodes to improve system performance and reduce execution delay. iFogSim is used to show simulation results.

The review of literature on dynamic service placement methods shows that only a few of these methods adopt specific service packaging techniques. Moreover, only a small fraction of these methods consider the costs associated with transferring service instances, such as the time required to download instances, the necessary bandwidth, and launch time. This lack of attention to these critical factors raises concerns about the practicality of these methods. The survey also indicates that most methods assume heterogeneity in both edge servers and services, but they do not give enough attention to privacy and security issues and service inter-operation. These research directions are crucial, especially in light of the increasing use of micro-service architecture and growing concerns over GDPR. The analysis of design objectives indicates that the most popular objective is improving service-level QoS, with most researchers proposing additional objectives alongside QoS improvement. However, it is also essential to investigate the method's overheads on various resources, such as processing, bandwidth, and energy and minimize them as secondary objectives. From a resource management perspective, the processing resource is the most critical, while energy and memory are the least focused. With the growing trend towards reducing the carbon footprint by cloud infrastructure and edge servers, it is vital to investigate the energy efficiency of dynamic service placement methods.

Chapter-3

SYSTEM DESIGN & DEVELOPMENT

In this section, we discuss the flow of associated products and services regarding and regardless of the inflow of asynchronous data being fetched for each battery swap. The model is to record whenever a customer comes, gives back the asset rented once it gets exhausted and takes away new asset till its subscribe date is not expired.

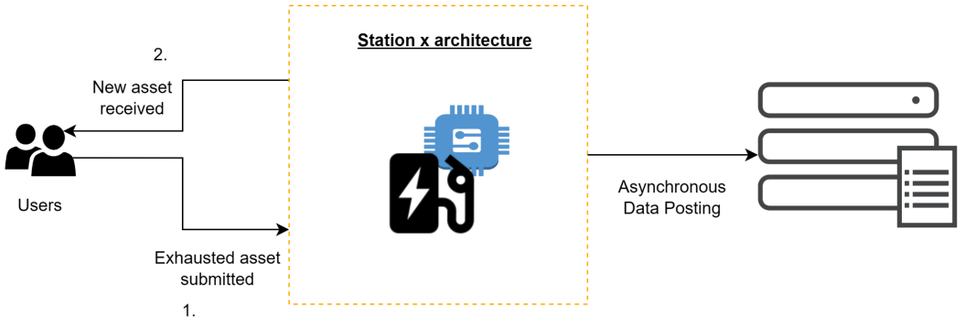


Figure 4. User interaction with IoT device flow

Figure 4 demonstrates the data collection by the station IoT device. The IoT devices send data to the integrated ByteBeam server once- swap case 1. When the user gives back the discharged asset, and once- swap case2. When the user takes a recharged asset. This data stream is divided into chunks of data and is received in packets.

These packets have unique can_id per swap, which includes various information like kmsTravelled, assetReturnedId, assetTakenId, stationId, State of charge of incoming asset, state of charge of given battery. A swap is between a station and a user, both of which are mentioned above as swap case 1 and swap case 2.

IoT (Internet of Things) devices send data to a server in a variety of ways, depending on the device and the application. However, the most common methods for sending data from IoT devices to a server are:

- MQTT (Message Queuing Telemetry Transport): MQTT is a lightweight messaging protocol that is commonly used for IoT applications. It uses a publish-subscribe model, where devices publish data to a central server or broker, and other devices or applications can subscribe to receive the data.

- HTTP (Hypertext Transfer Protocol): HTTP is the standard protocol used for communication on the web, and it is commonly used for IoT applications as well. Devices can send data to a server using HTTP requests, such as POST or PUT requests, and the server can respond with status codes or other data.

- CoAP (Constrained Application Protocol): CoAP is a lightweight protocol designed for use in resource-constrained environments, such as IoT devices. It uses a request-response model, similar to HTTP, and supports data transfer over UDP (User Datagram Protocol) or DTLS (Datagram Transport Layer Security).

- WebSocket: WebSocket is a protocol that allows for real-time, two-way communication between a client and a server. It can be used for IoT applications to send data from devices to a server, and also to receive commands or instructions from the server back to the device.

Overall, the method used for sending data from IoT devices to a server depends on various factors, such as the device capabilities, network constraints, and the application requirements. However, the protocols mentioned above are some of the most commonly used methods for IoT data transfer.

To make sure that the data sent is complete, we post multiple requests to the server until we receive a go ahead that full data has been processed by the server and posted to the server.

The data is sent over the network through an Acknowledgement- handshake mechanism which can be expressed as:

$$\sum_{i=1}^{n-1} \text{Wi} + \Phi < \text{t'} \qquad \text{,where Wi} \in \text{R}$$

Wi is waiting time per ri request.

Table 2: Symbols Used in the Study.

Symbols used	Meaning
W	Waiting time for the data between sending data & receiving acknowledgement
t’	Timeout for requesting server and expecting acknowledgement
Φ	Minimal time taken for a request
rn	Number of requests to server from IoT
R	Real numbers

3.1 Application Scenario

In this section step wise implementation and comparative study procedure is explained:

a) Ensure recording sufficient data from station IoT to account complete data for one swap.

b) Querying the Bytebeam server to get raw data

c) Filterdata copies to get one composite data chunk to be processed.

d) Implement the proposed filtering functionality after converting data from bytes to objects.

e) Isolate data and post it in the NoSQL database.

f) Keep track of outcomes for analysis (for various business purposes and generating relative KPIs).

3.2 Dataset

The current scenario is operating on 20, 000-30,000 swaps per month. The data being considered is:

- This communication consists of at least 4 copies for each swap.

- Data fetched from external server at a rate of 5 minutes (time reserved to get polling rate of 10 swaps/cycle)

- Time difference of 80ms between each can of data, (where data is packaged 5 bytes per can_id)

A swap consists of 6 can_ids each of which is assigned a can, which means a packet of data. This “can” consists of 5 bytes each. Once we query data from the OLAP based database, we get IoT Id with which we get the Station Id. Once that is identified, we proceed to parse the array of objects received after the SQL based query ran for this particular 5 minute window.

Clickhouse SDK for Node.js environment lets us get the data from the server directly by querying the attributes all of which can be taken account of from the online dashboard.

ClickHouse can also manage intricate data structures like nested arrays and nested dictionaries and supports a wide range of data types, including JSON, CSV, and Apache Avro. Additionally, it has a scalable architecture that enables it to manage massive amounts of data and distribute that data across a number of nodes and clusters for high availability and fault tolerance.
For organisations dealing with massive volumes of data that must be processed and analysed in real-time, ClickHouse is an effective and potent database management solution.

But for us to get an aggregate swap. It needs to be validated and put into account with all the values against various metrics. For this we have to fetch data. Moreover, polling our own data from a third party server is important to 1. Take account of data and keep a copy 2. Unreliability of their data server.

For our own sake, let’s call the IoT received data on the external server ‘swapped’, and the aggregate swap which we push to our NoSQL database, ‘swap’.

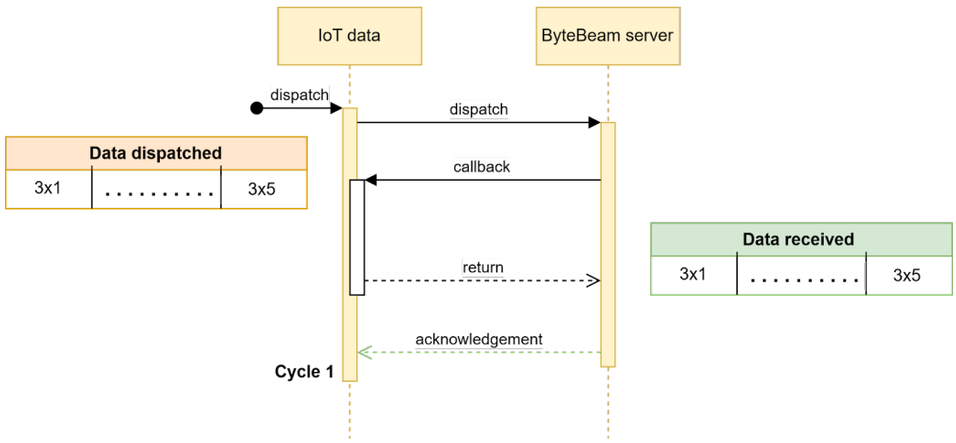


Figure 5: Data acknowledgement mechanism

The mechanism described works as a gatekeeper to make sure that all the can_ids for a requisite swap to get completed. Considering all the can_ids are present in the end, the acknowledgement mechanism sends consecutive asynchronous requests to the server to send data until the server accepts it as one bundle with all the packets present.

Once the acknowledgement is received back by the IoT device it stops sending the data. In such a case, a corner case may arise where the data is not being sent properly ever, causing the IoT request handler to go into an infinite loop for that particular swapped. To overcome this a certain timeout is introduced. The request has a time to live, which once exhausted causes the swap to be sent in such a format that it gets sent to the wrong data database of our AWS Cloud.

Only thing to be made sure is that can_id 3x5 gets sent as we poll the data from external servers by getting hold of this can_id and then getting data around it. This data is pulled at every trigger of the function. So, the external server acts like an intermediate data storage unit isolated only for the IoT devices' data received on a daily basis, both asynchronous data (swaps) or synchronous data.

This created multiple cases of the data that is being received on the endpoint after querying and filtering.

3.3 System Description

Here we carefully considered that the trigger to the serverless functionality must be such that the number of incoming swapped are optimal. They must be parsed, filtered, converted and posted to the database before the next batch is processed. This impacts on multiple cases like 1. If the trigger's swap is not posted, incoming swapped cannot be checked if duplicate, this can cause overwriting of data. We eliminate the case of multiple copies by internally assigning a unique_id to each swap and using it as the primary key in our database.

This primary key is:

no._of_days_passed_since_1jan1970 + unique_id

Avoidance of overloaded or under loaded function results in better outcomes and utilization of each lambda execution environment completely.

We are required to establish the connection between our AWS Client and Bytebeam server for which clickhouse comes handy.

```
const clickhouse = new ClickHouse({
        url: 'http://localhost',
        port: 8123,
        debug: false,
        basicAuth: null,
        isUseGzip: false,
        trimQuery: false,
        usePost: false,
        format: "json", // "json" || "csv" || "tsv"
        raw: false,
        config: {
                session_id                      : 'session_id if neeed',
                session_timeout                 : 60,
                output_format_json_quote_64bit_integers : 0,
                enable_http_compression         : 0,
                database                        : 'my_database_name',
        },
});
```

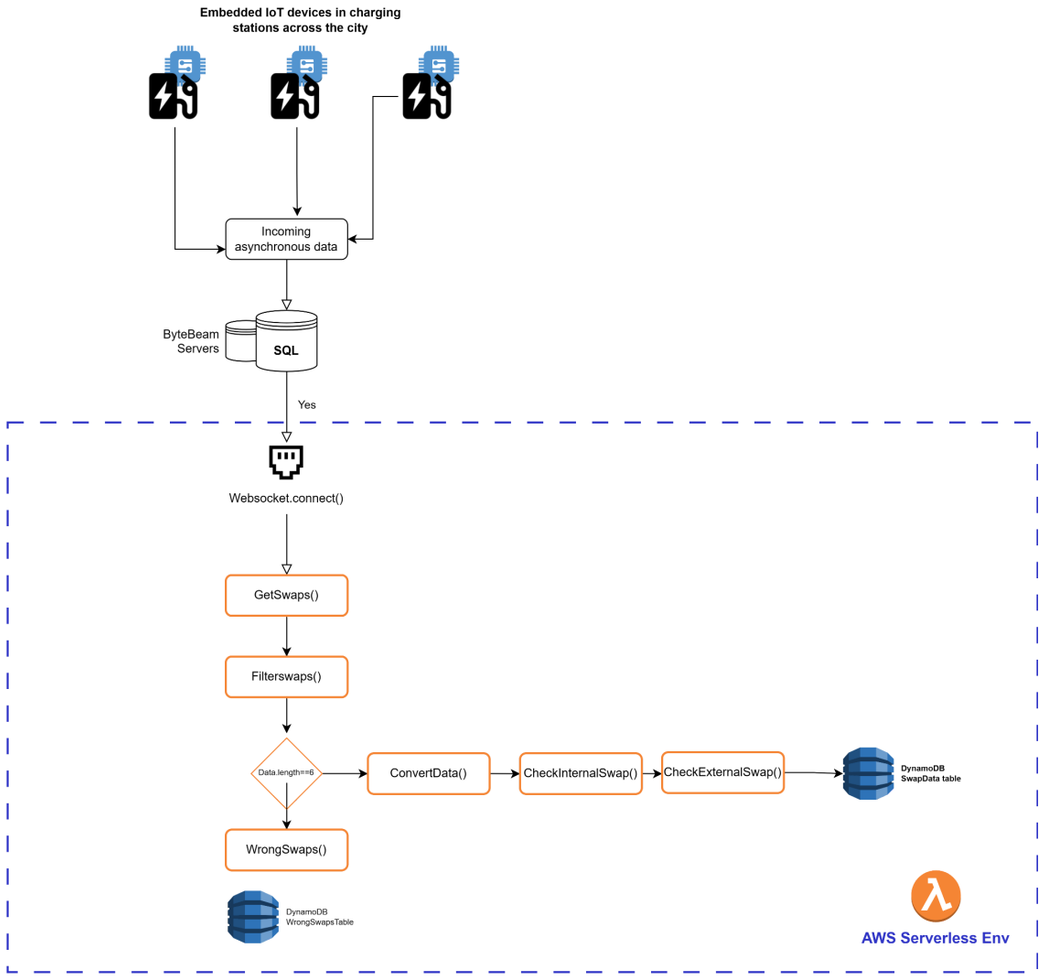


Figure 6: Aws Serverless function flow

This serverless function architecture presents monolithic function calling and overall environment interaction that is setup to handle incoming data. In a monolithic architecture, the application is built as a single, cohesive unit, where all the components of the application are tightly coupled and run in the same process or thread. Monolithic function calling refers to the process of calling a function or method within the same monolithic application.

In a monolithic architecture, functions and methods within the application can be called directly, without any need for external APIs or network communication. This is because all the components of the application are tightly integrated and share the same memory space and resources.

Monolithic function calling can be a simple and efficient way to develop and deploy applications, especially for smaller projects with limited complexity. However, as applications become larger and more complex, monolithic architectures can become difficult to maintain and scale, as any changes or updates to one component of the application can impact the entire system.

To address these issues, many organizations are moving towards microservices architectures, where applications are broken down into smaller, independent services that can communicate with each other over APIs. This allows for greater flexibility, scalability, and resilience, as each service can be developed, tested, and deployed independently, without affecting the rest of the system.

WebSockets are a communication protocol that allows for real-time, bi-directional communication between a client (such as a web browser) and a server. WebSocket connections are initiated with a handshake, and once established, allow for continuous, low-latency data transfer between the client and server.

To establish a WebSocket connection to an external server, the client (such as a web browser) sends an HTTP request to the server, requesting an upgrade to the WebSocket protocol. The request includes a special header called "Upgrade" with the value "websocket", as well as a "Connection" header with the value "Upgrade".If the server supports WebSockets and the upgrade request is accepted, the server responds with an HTTP response, also including an "Upgrade" header with the value "websocket", as well as a "Connection" header with the value "Upgrade". The response also includes a special "Sec-WebSocket-Accept" header, which contains a hash of the "Sec-WebSocket-Key" header value from the client request.

Once the handshake is complete, the connection is established, and data can be sent between the client and server in real-time using the WebSocket protocol. Data is sent in binary or text frames, which are similar to packets, and can be sent in either direction at any time.

WebSocket connections are particularly useful for real-time applications, such as chat applications, online gaming, and real-time data feeds, as they allow for low-latency, bi-directional communication between the client and server. They can also be used to reduce network overhead, as they allow for continuous, long-lived connections, rather than multiple short-lived connections.

We handle the incoming data in the function as follows:

- Converting each object in array to a swap with complete information.
- Transforming the data into an object.
- Checking if any internal duplicate bytes
- Checking if the swap is already registered.

In an OLAP system, data is organized into a multidimensional structure, with dimensions representing various attributes of the data, such as time, geography, product, customer, and so on. Users can then slice and dice the data based on different combinations of dimensions to create different views of the data, which can be used for in-depth analysis and reporting.

OLAP systems are designed to provide fast, interactive access to large volumes of data, allowing users to quickly explore and analyze data in real-time. They typically support a wide range of analytical functions, including drill-down, roll-up, pivot, and slice-and-dice, as well as various data visualization techniques, such as charts, graphs, and pivot tables.
Overall, OLAP is a powerful tool for analyzing and reporting on large volumes of data in real-time, providing users with a deeper understanding of their business data and enabling better decision-making.

The 5-minute recording intervals are taken as periods and divided into small sets for our own purposes.

3.4 Methodology

Lambda controls the asynchronous event queue for the function and makes an effort to retry on failures. Lambda makes two additional attempts to execute the function if it returns an error, with a one-minute delay between the first two attempts and a two-minute delay between the second and third attempts. Both faults provided by the function's code and runtime, such as timeouts, are considered function errors.

Additional requests are throttled if the function doesn't have enough concurrency to handle all of the events. Lambda returns the event to the queue and tries to run the function again for up to 6 hours in the case of throttling problems (429) and system issues (500-series). After the initial attempt, the retry interval increases exponentially up to a maximum of 5 minutes.

New events could age out if the queue is very large before Lambda can transmit them to your function. Lambda discards an event if it has expired or if all attempts at processing it have failed. You can set up a function's error handling so that fewer retries are made by Lambda or that unprocessed events are discarded more rapidly.

Lambda can also be set up to transmit an invocation record to an additional service. The following destinations are supported for asynchronous invocation via Lambda. You should be aware that SNS FIFO topics and SQS FIFO queues are not supported.A common SQS queue is Amazon SQS.

- A common SNS topic is Amazon SNS.

- A Lambda function from AWS.

- An event bus powered by Amazon EventBridge.

The invocation record includes information in JSON format about the request and response.

One of the major benefits of the edge-cloud system is improved performance. By processing data at the edge, organizations can reduce the amount of data that needs to be sent to the cloud, thereby reducing the latency and bandwidth requirements. This results in faster response times and improved overall system performance.

Another advantage of the edge-cloud system is increased security. By processing and storing data at the edge, organizations can reduce the risk of data breaches and other security incidents. This is because the data is stored and processed locally, rather than being sent to a centralized cloud, which is a more attractive target for hackers.

Furthermore, the edge-cloud system is also beneficial for reducing energy consumption. By processing data at the edge, organizations can reduce the amount of data that needs to be sent to the cloud, which reduces the energy consumption required for transmitting the data.

In conclusion, the edge-cloud system is a new computing paradigm that combines the benefits of both edge computing and cloud computing. It enables organizations to process and analyse data at the edge of their networks, closer to where it is generated, and then store and analyse it in the cloud. This system provides a number of advantages over traditional cloud computing systems, including improved performance, reduced latency, increased security, and reduced energy consumption. As the amount of data generated from connected devices continues to grow, the edge-cloud system will become increasingly important for organizations looking to improve their computing infrastructure.

As shown in Figure 6, we swapped data from various placed IOT devices in multiple geographical locations. These requests sent by the client server are primarily processed by the function at the periphery of the cloud environment. These requests are taken as input parameters in the process of creating an optimized swap for our database.

Standard rate and cron expressions are supported by AWS Lambda for frequencies up to once per minute. Although rate expressions are easier to define, they lack the cron triggers' support for fine-grained schedule control.

A cron expression is a string that specifies a set of times or intervals at which a job or task should be executed. It is commonly used in scheduling software, such as cron (a time-based job scheduler in Unix-like operating systems), to automate recurring tasks.

A typical cron expression consists of six fields, separated by spaces, which define the following values:

- Minutes (0-59)
- Hours (0-23)
- Day of the month (1-31)
- Month (1-12 or Jan-Dec)
- Day of the week (0-6 or Sun-Sat)
- Year (optional)

Each field can contain a single value, a comma-separated list of values, a range of values (specified using a hyphen), or a wildcard character ("*") to indicate all possible values. For example, the cron expression "0 0 1 * *" would execute a task at midnight on the first day of every month, while the expression "0 12 * * 1-5" would execute a task every weekday at noon.

Cron expressions can also include additional syntax, such as the forward-slash character ("/"), which can be used to specify intervals. For example, the expression "*/5 * * * *" would execute a task every 5 minutes.

Overall, cron expressions provide a flexible and powerful way to schedule recurring tasks, allowing users to define a wide range of schedules based on specific dates, times, and intervals.

The following syntax applies to rate expressions for EventBridge (CloudWatch Events).

rate(Value Unit)

Where Value is a positive integer and Unit can be minute(s), hour(s), or day(s). A rate expression starts when you create the scheduled event rule. For a singular value the unit must be singular (for example, rate(1 day)), otherwise plural (for example, rate(5 days)).

Table 3: Cron expressions available by AWS.

Frequency	Expression
Every 5 minutes	rate(5 minutes)
Every hour	rate(1 hour)
Every seven days	rate(7 days)

AWS EventBridge is a fully managed event bus service that makes it easy to build event-driven applications at scale. You can use EventBridge to route events between AWS services, third-party SaaS applications, and your own applications, and also to trigger serverless functions such as AWS Lambda functions.

To integrate EventBridge with a Lambda function trigger, you can follow these steps:

1. Create a new rule in EventBridge: Start by creating a new rule in EventBridge that defines the event pattern that will trigger the Lambda function. This can be done via the AWS Management Console, AWS CLI, or AWS SDKs.

2. Specify the Lambda function as the target: In the same rule, specify the Lambda function as the target of the event. This can be done by specifying the Lambda function's Amazon Resource Name (ARN) as the target of the rule.

3. Configure permissions: Ensure that the Lambda function has permission to be triggered by the EventBridge rule. You can do this by configuring the function's IAM role to include the necessary permissions for receiving events from EventBridge.

4. Test the integration: Once the rule is created and permissions are configured, you can test the integration by publishing an event that matches the pattern defined in the rule. This can be done using the EventBridge console or API.

5. Monitor and troubleshoot: Finally, monitor and troubleshoot the integration by viewing the CloudWatch Logs for the Lambda function and the EventBridge rule. This can help you identify any issues or errors that may occur during the integration process.

Overall, integrating EventBridge with a Lambda function trigger is a straightforward process that can help you build event-driven applications with ease.

3.5 Proposed Algorithm

In this section, we go through the visualized function flow, discuss corner cases of the data processing that we have considered and the related constraints in our case to eliminate any data anomaly in final database reserved for transformed form of *swapped,* that is a swap.

AWS Lambda is a serverless compute service that allows you to run your code without having to provision or manage servers. Lambda functions can be written in a variety of programming languages, including Node.js.

AWS Lambda supports Node.js 16.x runtime environment, which means you can write and run your Node.js 16.x code directly in Lambda. Once you are satisfied with your function, deploy it by clicking the "Deploy" button in the Lambda console. Your function is now ready to be triggered by various AWS services, such as API Gateway or EventBridge.

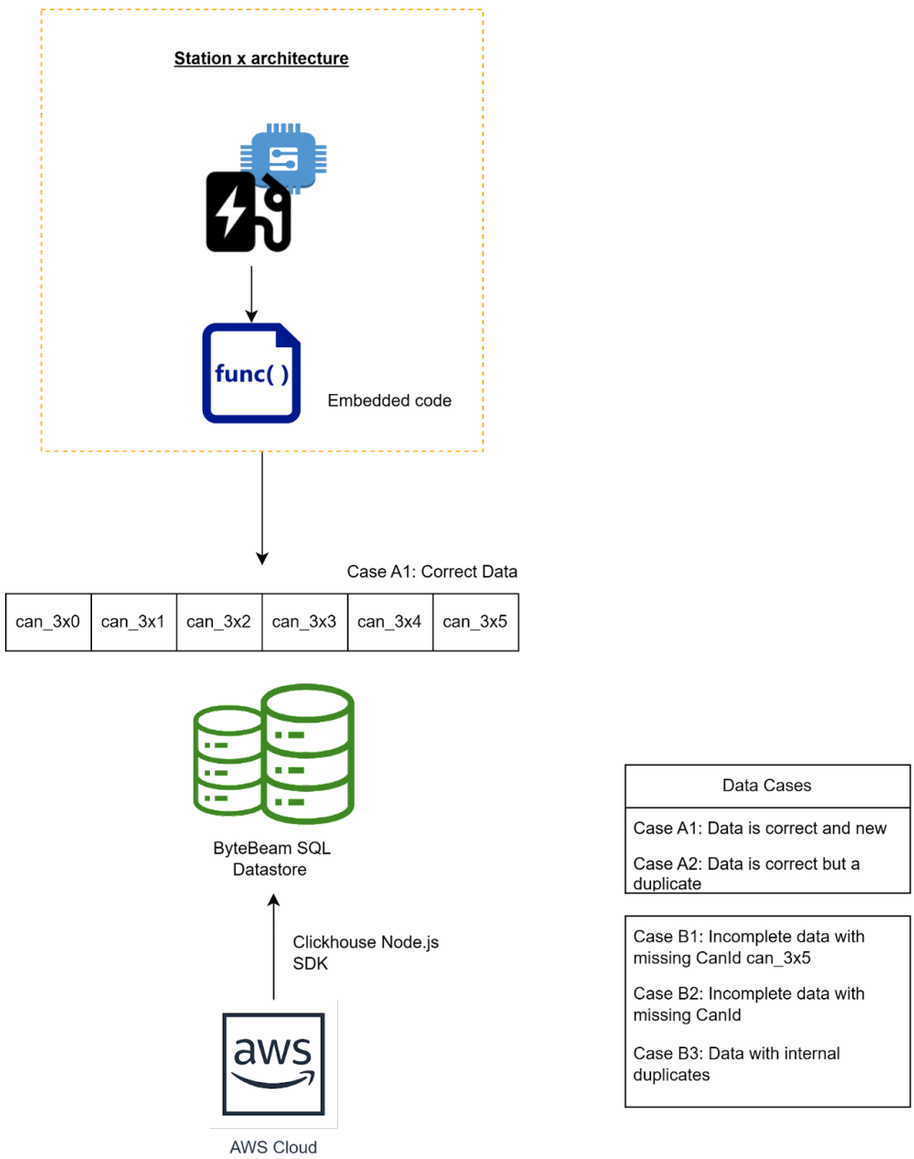


Figure 7: Critical cases on data received.

Overall, setting up a Node.js 16.x environment in AWS Lambda is a simple process that can be completed quickly and easily. It allows you to run your Node.js 16.x code in a serverless environment without having to manage servers or infrastructure.

The algorithm presented above in Figure 8, talks about various forms of event that can be received by the function. This data is received after we update the start time and end time for the SQL query being sent to the external server by fetching Cloudwatch current time and setting it to start time for the next cycle.

Once that is set in an external table, we move on to setup a connection between AWS lambda containers for this particular execution to the SQL server through clickhouse SDK for Nodejs 16x and keeping the connection alive for a given session.

Figure 8 outline cases for the specific data streams:

- Case A1: Outlines a new data- swapped which consists of sequential can_ids and length 6. It comes together to contribute a threshold amount of bytes.

- Case A2: The data is correct both ID wise and length wise but is a duplicate of previously received swapped data, which will fall in the category of discarded duplicate of previously polled execution swapped data.

- Case B1: Erroneous data received with incomplete data Ids, particularly doesn’t consists a specific can_id which is required to poll result from the server using SQL query.This data cannot be processed and is sent to wrong data table.

- Case B2: Erroneous data received length smaller than 6, which indicates towards both missing Ids or duplicate Ids. This data cannot be processed and is sent to wrong data table.

- Case B3: One trigger polls around 7-10 swapped packets, these might have duplicates within them, which aligns with the fact that one triggered function may get data of the same swapped twice in one polling via SQL query.

Overall, this algorithm is designed to optimize the placement of data in the internal database which can be further used in sheet via automated sheets script to showcase data which can be fetched by an internal REST API with GET method to display regular data.

Table 4: Features of given function schema.

Features	
Filterswaps	Sending 2-3 copies of data via IoT device, in can_id format with each can_id holding 5 bytes
Checkinternalswaps	Once a full set data is extracted from multiple copies, checked for any internal data anomaly
Checkexternalswaps	This data set is unique for currently received data sets and is then checked with previously received swap data
Unique Id	We create a unique Id which consists of Number of days passed since Thursday, January 1, 1970 12:00:00 AM + uniqueIdReceived

Chapter-4

EXPERMENTS & RESULT ANALYSIS

Vertical scaling, also known as scaling up, refers to the process of increasing the resources available to a single instance of a software application. This is usually done by adding more processing power, memory, or storage capacity to a single server or virtual machine.

4.1 Software Practices

1. **Vertical Scaling**: In software development, vertical scaling is often used to improve the performance and scalability of an application, especially in cases where the application is running on a single machine and is experiencing performance bottlenecks due to limited resources. Vertical scaling is achieved by upgrading the hardware components of the server or virtual machine running the application. For example, adding more RAM or upgrading to a faster processor can increase the amount of work the application can handle in a given amount of time. While vertical scaling can be an effective way to improve performance, it has some limitations. Eventually, the hardware resources of the server or virtual machine will reach their maximum limits, at which point vertical scaling is no longer a viable solution. At that point, horizontal scaling, which involves adding more instances of the application across multiple servers, may be necessary. Overall, vertical scaling can be an important tool in improving the performance and scalability of a software application, especially in cases where the application is running on a single machine and is experiencing resource limitations. However, it's important to recognize its limitations and consider other scaling options as needed.

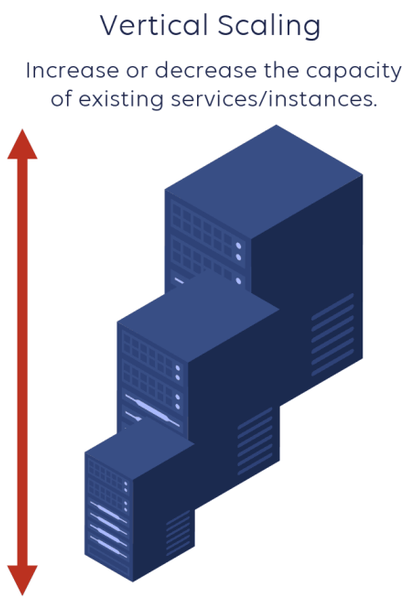


Figure 8: Vertical scaling in AWS microVM container

2. **Polling:** In network requests, polling refers to a technique where a client repeatedly sends requests to a server to check for updates or changes in data. The client sends a request at a fixed interval, and the server responds with the current data or an indication that no changes have occurred since the last request. Polling can be done using various HTTP methods such as GET, POST, or AJAX. The frequency of polling can be controlled by adjusting the interval between requests. Shorter intervals can provide more real-time updates but can also increase the load on the server and consume more bandwidth. Longer intervals can reduce the load on the server but may lead to delayed updates. There are two main types of polling: long polling and short polling. With short polling, the client sends a request at a fixed interval and receives an immediate response from the server, even if there are no updates. With long polling, the client sends a request to the server and waits for the server to respond with new data or a timeout message if no changes occur within a certain time frame. Polling can be an effective technique for keeping data up-to-date and providing real-time updates to users. However, it can also consume significant resources on the client and server sides, especially if the polling interval is short or if

many clients are polling the same server simultaneously. Other techniques such as websockets and server-sent events can provide more efficient and scalable solutions for real-time updates.

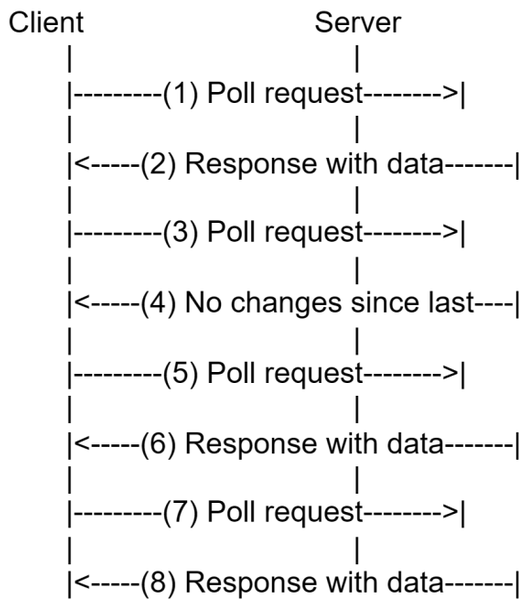


Figure 9: Polling to fulfill request from client

In figure 9, the client sends a polling request to the server at fixed intervals (steps 1, 3, and 5). The server responds with the current data (steps 2, 6, and 8) or an indication that no changes have occurred since the last request (step 4).

The polling interval can be adjusted depending on the desired frequency of updates. Shorter intervals provide more real-time updates but consume more resources on the client and server sides, while longer intervals reduce the load but may result in delayed updates.

Polling can be an effective technique for keeping data up-to-date and providing real-time updates to users, but it can also consume significant resources and may not be suitable for all use cases. Other techniques such as websockets and server-sent events may be more efficient and scalable solutions for real-time updates.

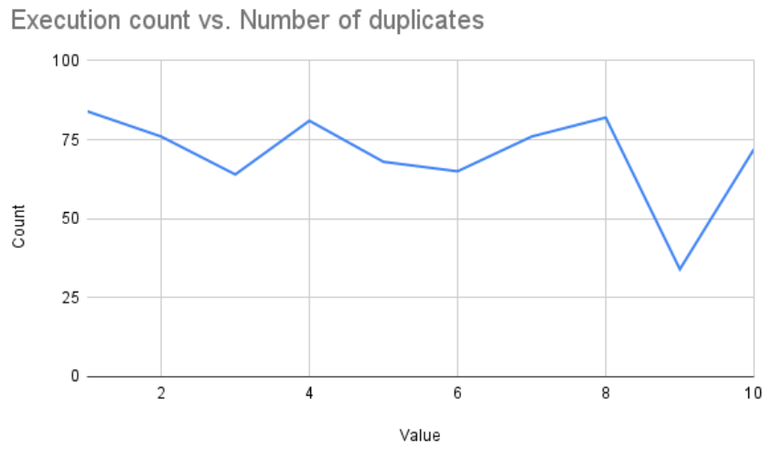
Table 5. Mean Time and Memory utilization

timestamp	message
1682061118212	INIT_START Runtime Version: nodejs:16.v12 Runtime Version
1682061118819	START RequestId: 892e431f-71a3-4446-a3d5-e48789baa05a Version: $LATEST
1682061118868	INFO start time and end time 2023-04-21 07:06:58 2023-04-21 07:11:58
1682061119035	INFO Can data executed 1682061119034
1682061119129	INFO filter Data function executed 1682061119129
1682061119182	INFO at convertdata-uniqueid-- 168206085701160
1682061119198	INFO at convertdata-uniqueid-- 168206093900323
1682061119212	INFO at convertdata-uniqueid-- 168206097501673
1682061119212	INFO Convert Data Executed 1682061119212
1682061119262	INFO Data checked & posted 1682061119262
1682061119262	INFO Once Can data over 1682061119262 1682061118867
1682061119268	END
1682061119268	**REPORT Duration: 449.00 ms Billed Duration: 450 ms Memory Size: 10240 MB Max Memory Used: 99 MB Init Duration: 603.83 ms**

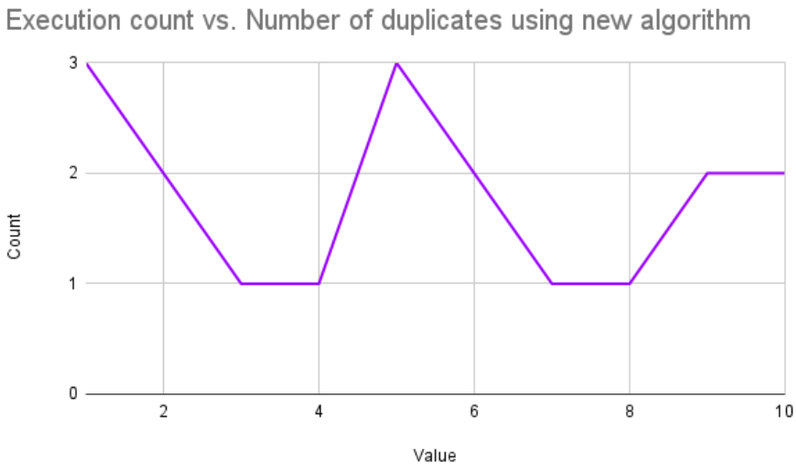
In AWS CloudWatch Logs, the time taken and billed duration for a Lambda function invocation are two different metrics that can provide valuable insights into the performance and cost of your function.

Time taken: This metric represents the total amount of time that a function took to complete its execution, including any time spent waiting for external resources such as network I/O, database queries, or API calls. The time taken metric is useful for identifying performance bottlenecks and optimizing the overall execution time of your function.

Billed duration: This metric represents the amount of time for which your function was billed by AWS. It is calculated by rounding up the time taken to the nearest 100ms and billing you for that amount of time. For example, if your function took 550ms to execute, you would be billed for 600ms of execution time. The billed duration metric is important for tracking your Lambda usage and optimizing your function's cost. Both of these metrics are available in the CloudWatch Logs for your Lambda function, and can be viewed in the function's monitoring tab or queried using CloudWatch Metrics. By analyzing these metrics, you can gain insights into the performance and cost of your Lambda function, and optimize it accordingly.



Graph 1. Mean value of duplicates polled



Graph 2. Mean value of duplicates polled in new algorithm

Here in Table 5, we account for the execution environment of lambda, which exists for approx 15 minutes, hence reducing the time for cold start and container build and deployment.

$$3\varpi > \Gamma$$

$$Where\ \varpi\ is\ Window\ for\ polling\ data\colon\ 5min$$

Γ is the time for which the execution environment exists.

Cold start in AWS Lambda refers to the initial time it takes to spin up a new execution environment for a function that has not been recently executed. When a Lambda function is triggered, AWS creates a new container or "execution environment" to run the code in a serverless environment.

If the function has not been called in a while, or if the function has never been called before, then a new execution environment needs to be created, which can result in a delay before the function code can begin executing. This delay is known as a "cold start".

During a cold start, AWS initializes the execution environment, including loading the function code and any dependencies, setting up connections to databases or other resources, and performing any necessary setup tasks. The amount of time it takes to complete these tasks can vary depending on the size and complexity of the function code, the number of dependencies, and the resources required by the function.

Cold starts can impact the performance and responsiveness of serverless applications, particularly for functions that need to run quickly or handle a high volume of requests. To minimize the impact of cold starts, developers can use techniques such as optimizing the function code size, reducing the number of dependencies, and keeping execution environments warm by periodically invoking the function to ensure that an execution environment is already available when a request arrives.

For INIT_DURATION, we consider the fact that time taken for a cold start includes:

1. Creation of MicroVM and installing packages for creation of node.js environment in a lambda
2. Containerization of the built environment and its deployment

The TTL is around 15 minutes which includes subsequent invocation and billed duration (DURATION) + INIT_DURATION.

In Amazon DynamoDB, a Global Secondary Index (GSI) is an index that you can create in a DynamoDB table to support query operations that are not possible or efficient using the table's primary key.

A Global Secondary Index with a "projection type" of "INCLUDE" can be used to create a Global Secondary Index with "Global Secondary Index write sharding", also known as "Global Secondary Index with a global second index". This index allows you to achieve higher write throughput by allowing you to specify additional attributes to be included in the index, without actually including them in the index itself. This way, the index can still be used for query operations even though it does not contain all the attributes needed for those queries.

When you create a Global Secondary Index with write sharding, DynamoDB creates multiple index partitions to distribute write traffic across multiple hosts. Each index partition has its own copy of the GSI data, and DynamoDB automatically distributes write requests across the partitions. This can help you achieve higher write throughput and reduce write latency for queries that use the GSI.

Note that write sharding is only available for Global Secondary Indexes with a projection type of "INCLUDE", and there is a limit on the maximum number of index partitions that can be created for a single GSI.

However, many other database management systems make use of comparable methods. Paying close attention to the little things is what really sets ClickHouse apart. The majority of programming languages have implementations for the majority of widely used algorithms and data structures, although these are typically too broad to be useful. Instead of just implementing things at random, every task may be thought of as a landscape with different features. For instance, if you require a hash table, you should think about the following important factors:

Which hashing algorithm should I use?
Open addressing or chaining for the collision resolution algorithm?

Memory organisation: separate arrays for keys and values, or a single array? Will small or large values be stored?
Fill factor: When should I resize, and how? How may values be resized and moved around?
If values are eliminated, which algorithm will perform better?
Will we require fast bitmap probing, inline string key placement, support for immovable values, prefetch, and batching?
Hash tables are essential for implementing GROUP BY, and ClickHouse automatically selects one of more than 30 variants for each individual query.

The same is true with algorithms; for instance, when sorting, you might think about:

1. An array of numbers, tuples, strings, or structures—which will be sorted?
2. Is all data fully accessible in RAM?
3. Are stable sorts necessary?
4. Do we require a thorough sort? Could the n-th element or a partial sort be sufficient?
5. How should comparisons be used?
6. Are we organising information that has already been organised in part?

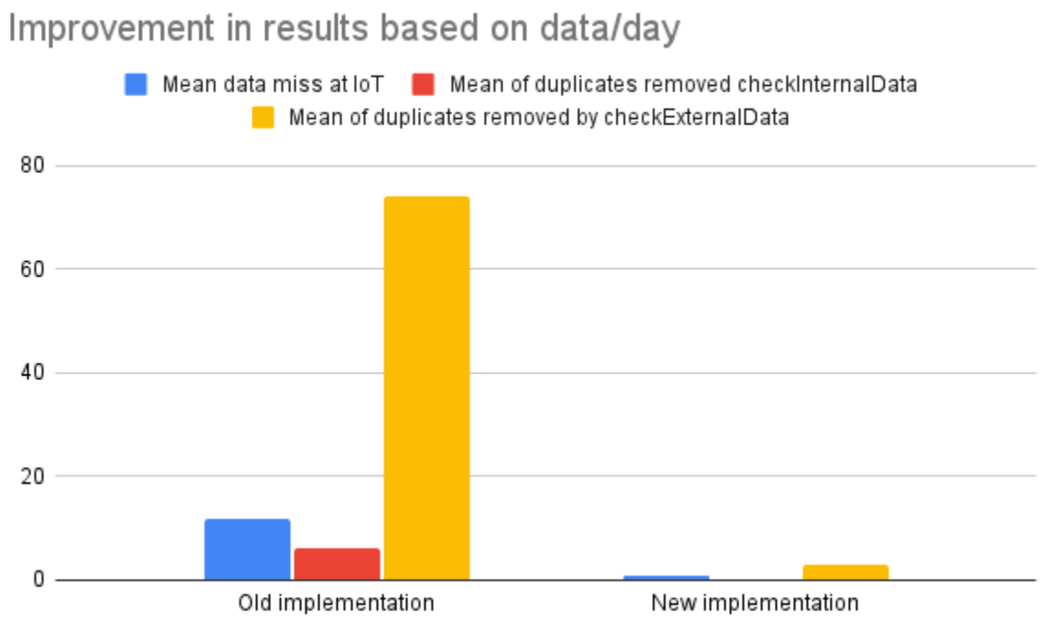
When compared to their generic counterparts, algorithms that depend on the characteristics of the data they are working with frequently perform better. If it isn't actually known beforehand, the system can test different implementations and pick the one that functions best at the moment.

Check out to discover how ClickHouse uses LZ4 decompression as an example. Not to mention, the ClickHouse team constantly scans the Internet for anyone claiming to have created the greatest implementation, algorithm, or data structure to do a task and tests it. The majority of those statements seem to be untrue, but occasionally you may come upon a real gem.

4.2 Utilization

4.2.1 Utilization (in terms of global cost)

How evenly the burden is dispersed across the network is examined in this analyzed element. Load-balancing is not a function of the cloud method by design; hence it is not included in this assessment.



Graph 3. Comparative analysis of two methodologies.

4.2.2 Data modeling and further utilization

Google Apps Script is a cloud-based scripting language that allows developers to extend and automate the functionality of Google Sheets, Google Docs, and other G Suite applications. With Google Apps Script, you can create custom functions, automate tasks, build add-ons, and integrate with other Google services.

In the context of Google Sheets, Apps Script allows you to create custom functions that can be used in your sheet, build custom menus and dialogs, automate data entry and formatting, and integrate with other Google services such as Google Drive, Google Calendar, and Google Analytics.

Some of the key features of Google Apps Script for Google Sheets include:

1. A powerful and easy-to-learn scripting language based on JavaScript
2. The ability to write custom functions that can be used in your sheet
3. APIs for interacting with various Google services
4. The ability to create custom menus and dialogs to enhance user interaction
5. The ability to automate repetitive tasks such as data entry and formatting
6. The ability to create custom add-ons that can be shared and used by others
7. Easy deployment and sharing of scripts with others

Overall, Google Apps Script is a powerful tool for enhancing the functionality of Google Sheets and streamlining your workflows. Whether you're looking to automate repetitive tasks, create custom functions, or build add-ons, Apps Script provides a flexible and easy-to-use platform for achieving your goals.

```
function insertData() {
  var sheetName = "Sheet1"; // Change the sheet name to your sheet name
   var data = ["John", "Doe", "john.doe@example.com"]; // Change the data as per your requirement
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  var sheet = ss.getSheetByName(sheetName);
  sheet.appendRow(data);
}
```

In this code, we define a function called insertData(). The function sets the sheet name to Sheet1 (you can change it to your sheet name) and sets the data that we want to insert into the sheet.

Next, the function gets the active spreadsheet using the SpreadsheetApp.getActiveSpreadsheet() method and gets the sheet object using the ss.getSheetByName(sheetName) method. Finally, we use the sheet.appendRow(data) method to insert the data into the sheet.

This method appends the data to the last row of the sheet. You can call this function from the Google Apps Script editor or by creating a trigger to run the function on a schedule or on a specific event.

```
function getDataFromAPI() {
  var url = "https://jsonplaceholder.typicode.com/todos/1"; // Replace with your API endpoint URL
  var response = UrlFetchApp.fetch(url);
  var json = response.getContentText();
  var data = JSON.parse(json);
var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
  sheet.getRange(1, 1).setValue(data.userId);
  sheet.getRange(1, 2).setValue(data.id);
  sheet.getRange(1, 3).setValue(data.name);
  sheet.getRange(1, 4).setValue(data.rid);
}
```

In this code, we define a function called getDataFromAPI(). The function sets the API endpoint URL to https://jsonplaceholder.typicode.com/todos/1 (you can replace it with your API endpoint URL).

Next, the function uses the UrlFetchApp.fetch(url) method to retrieve the data from the API endpoint. This method returns an HTTPResponse object that contains the response data. We then use the getContentText() method to extract the JSON data from the response, and the JSON.parse(json) method to parse the JSON data into a JavaScript object.

Finally, we get the active sheet of the Google Sheet using the SpreadsheetApp.getActiveSpreadsheet().getActiveSheet() method and use the setValue() method of the Range object to set the values of the cells A1 to D1 with the values from the API response.

You can modify this code to suit your requirements, such as changing the API endpoint URL, parsing the JSON data differently, or setting the values in different cells. You can also use this code as a starting point to build more complex scripts that retrieve and manipulate data from API endpoints

In this example, we're sending a JSON payload to the ByteBeam server using the fetch function. We specify the URL of the API endpoint, the payload data, and the HTTP method (in this case, POST). We also set the Content-Type header to application/json to indicate that we're sending JSON data in the request body.

Once we've constructed the request options, we use the fetch function to send the request to the ByteBeam server. We check the response status to see if the request was successful or not, and log an error message if it wasn't.

```
// Load the AWS SDK
const AWS = require('aws-sdk');

// Create an instance of the DynamoDB client
const dynamodb = new AWS.DynamoDB({region: 'us-east-1'});

// Define the JSON payload to send to the DynamoDB table
const itemData = {
  id: {S: '123'},
  name: {S: 'John Doe'},
  age: {N: '30'}
};
const params = {
  TableName: 'my-table-name',
  Item: itemData
};

// Call the putItem function to send the JSON payload to the DynamoDB table
dynamodb.putItem(params, function(err, data) {
 console.log(data)
}
```

Table 6. Solution details

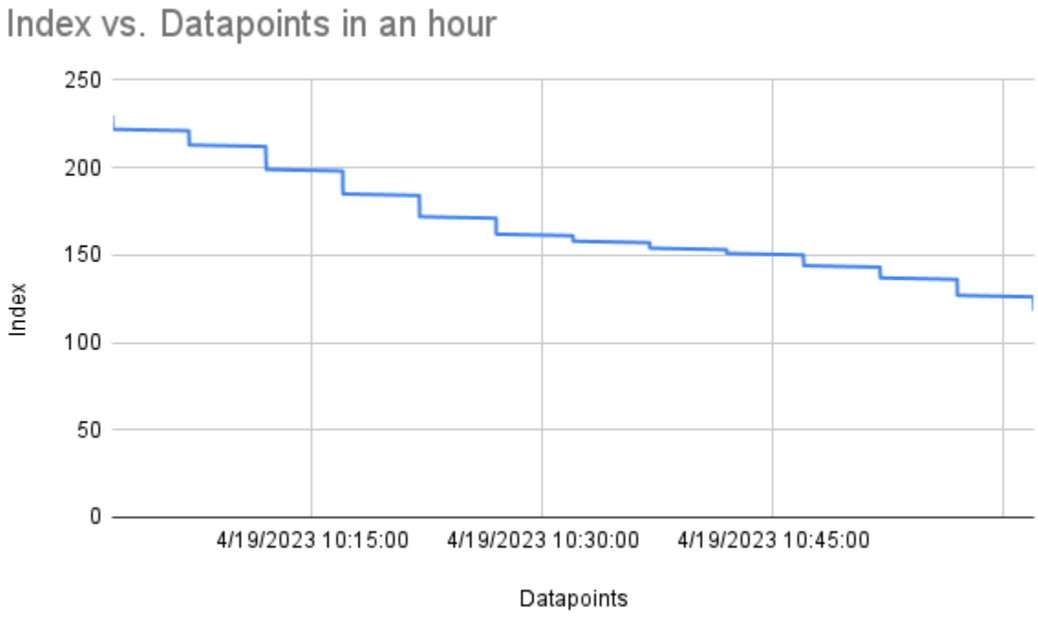
Solution Details	
Solution details	Creating a unique_id for each swap and adjusting it in 15 bytes of incoming data such that it can be checked if 1.duplicate or 2. Incomplete data
Testing	Testing with dummy batteries & Internal testing 1
Measuring success	98.67% swaps caught completely, transformed from bytes in array to usable data.

Internal testing, also known as "in-house testing" or "alpha testing", is a type of software testing that is conducted by the development team within the organization or company that is developing the software. The purpose of internal testing is to identify and fix issues or bugs in the software before it is released to external users or customers.

Internal testing typically involves a series of tests designed to evaluate the functionality, usability, and performance of the software. This may include unit testing, integration testing, system testing, and acceptance testing. The tests may be automated or manual, and may involve various techniques such as black-box testing, white-box testing, and exploratory testing.

Internal testing is an important part of the software development lifecycle, as it allows developers to catch and fix issues early in the development process, before they become more difficult and expensive to address. It also helps ensure that the software meets the requirements and expectations of the stakeholders, including users, customers, and business owners.

Once internal testing is complete and the software has been deemed stable and functional, it may then undergo external testing or "beta testing", where it is tested by external users or customers to further identify and address any issues or bugs.



Graph 4: Incoming data points to be processed in an hour

This data primarily helps in determining churn rate and utilization of product by the customer. In data analysis and business metrics, churn rate refers to the rate at which customers or users stop doing business with a company over a given period of time. It is a measure of customer attrition or loss.

Churn rate can be calculated in different ways depending on the business and the data available. One common method is to calculate the percentage of customers who have stopped using a product or service during a given time period, such as a month or a year. For example, if a company had 100 customers at the beginning of the month and lost 10 customers by the end of the month, the churn rate would be 10%.

Churn rate is an important metric for businesses, particularly those that rely on recurring revenue or subscription-based models. A high churn rate can indicate that there are issues with the product or service, or that customers are dissatisfied with their experience. It can also be a warning sign that the company is not retaining customers as effectively as it could be, which can impact revenue and growth.

Businesses can use churn rate data to identify patterns and trends, and to develop strategies for improving customer retention.

This may include addressing specific issues that are causing customers to leave, improving customer support or product features, or implementing loyalty programs or other incentives to encourage customers to stay.

Table 7. Time Taken for one trigger

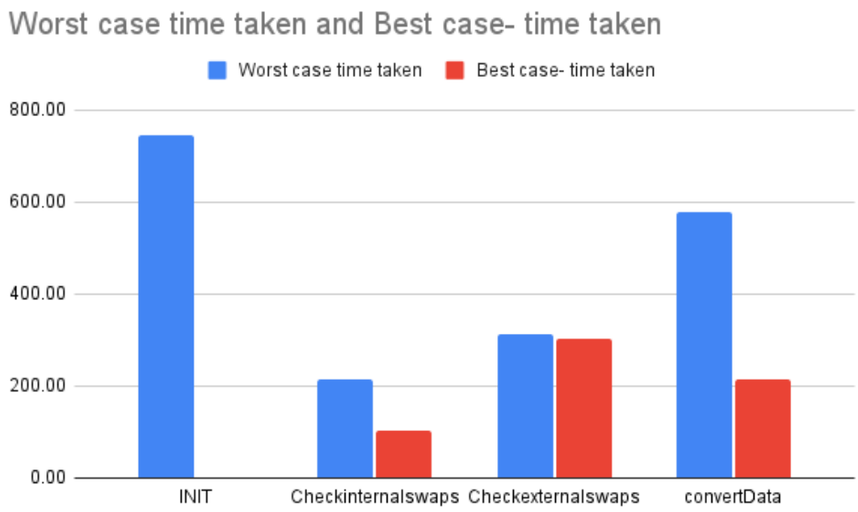
Time taken	
INIT	745 ms
Checkinternalswaps	213 ms
Checkexternalswaps	312 ms
convertData	579 ms

Performance index in software refers to a metric or set of metrics that are used to measure and evaluate the performance of software applications. Performance index is important because it can help identify performance issues, bottlenecks, or areas of inefficiency in the software, and can guide developers in making improvements.

There are many different metrics that can be used to calculate performance index, depending on the type of application and the specific goals of the evaluation. Some common metrics used in performance index include:

Response time: The amount of time it takes for the software to respond to a user request or action.

Throughput: The rate at which the software can process a given number of requests or transactions.



Graph 5. Worst and Best case scenario for this implementation

CPU usage: The amount of processing power and resources used by the software.
Memory usage: The amount of memory or RAM used by the software.
Network latency: The amount of time it takes for data to travel between different systems or devices.

Error rate: The frequency and severity of errors or bugs in the software.
Performance index can be measured through various techniques, such as load testing, stress testing, or profiling. By monitoring and analyzing these metrics, developers can gain insights into the performance of the software and identify areas where improvements can be made.

This helps us reduce the time of environment creation, MicroVM production to environment setup by container, significantly by 32.883% approximately. To filter and collect data as quickly as possible, ClickHouse was initially created as a prototype. A basic GROUP BY query does what is required to construct a typical analytical report. The ClickHouse team made a number of important choices that, when combined, allowed for the completion of this task:

Column-oriented storage: Although a report may only employ a handful of the hundreds or thousands of columns that make up source data, this is a common situation.

To save money on costly disc read operations, the system must refrain from reading irrelevant columns.

Indexes: Resident memory Only the necessary columns and necessary row ranges of those columns may be read from ClickHouse data structures.

In real data, a column frequently contains the same, or not that many different, values for neighboring rows, hence grouping various values of the same column together typically results in superior compression ratios (compared to row-oriented systems). ClickHouse offers specialised codecs that can further compress data in addition to standard compression.

Execution of queries using vectorization: ClickHouse not only stores but also processes data in columns. This improves CPU cache utilisation and enables the use of SIMD CPU instructions.

Scalability: ClickHouse can employ every CPU core and disc that is available to process even a single query. not just on a single server but also across the entire cluster's CPUs and discs.

The presented results of the overall experiments earlier in the report, which can be summed up as a complete overview and comparison of data handling from IoT devices where we have

i) Reduced execution time for data functionality and database posting.

ii) Reduce data traffic by efficiently deploying the solution on architecture based computing environment for IoT requests to execution environemnt. We talk about avoiding bottleneck situations and have laid out extensive test grounds to support the same.

The comparison further validates these results better on the grounds mentioned and shows a significant improvement in the given subset. The result summarized as

i. brings forth the efficiency which aligns with overall productivity when it comes to application of the solution.

ii. defines the working and output of such implementation in real-life features and its significance.

Provision of resources and tackling various technological hurdles for optimal fault tolerance is ultimately very crucial and continues to be in the scenario of broad spectrum and variety of IoT devices available.

The overall analysis can be expanded on the grounds of various use cases and on further experimentation that increases load with a broader dataset, it can be used for everyday application platforms.

We would like to elaborate on other use cases which include polling of data from external servers where we mention the deployment of such an ecosystem in case of dense incoming data streams that involve interaction between two different types of databases- NoSQL based key-value datastore and traditional SQL database. For enhancement of overall output, more focus is on data analytics and supported cloud systems. Such advancements in the field of cloud computing, specifically Lambda@Edge and batching of execution triggers focusing on IoT devices, helps in application in the IoT field. With respect to that, minimizing time constraint in such customer facing application as an objective increases efficiency of IoT deployments and data analytics provides a way of focusing on overall feature enhancement and further utilizing the data to spurt out advancements in overall business yield.

A decentralized mechanism consisting of nodes distributed in a network confirms a feasible and improved manner of data handling in such a setting to decrease total computation time, from sending a request to an IoT device to receiving data response from a database based on an IoT device.

Chapter-5

CONCLUSIONS

5.1 Conclusions

The internet of things (IoT) is a rapidly growing field, with the increasing adoption of IoT devices across multiple disciplines, there is a growing need for cost-effective and high-quality computing solutions that can handle the large amounts of data generated by these devices. The placement of services on IoT devices is an important aspect of optimizing the performance of IoT systems. In this study, the authors present a novel approach to service placement of IoT devices, utilizing distributed agents for plan generation and selection.

The proposed approach is based on efficient computing, which involves utilising computing resources closer to the edge of the network, where the IoT devices are located. By utilizing this algorithm, the proposed approach effectively minimizes the cost-of-service execution while simultaneously reducing data traffic. The results demonstrate the efficacy of the proposed solution in improving the Quality of Service (QoS) and enhancing the overall performance of the system.

The authors' focus on cost minimization and QoS improvement is particularly important for the development of future computing applications in interdisciplinary environments. As the number of IoT devices continues to grow, it is critical that we develop efficient, cost-effective, and secure computing solutions to handle the large amounts of data generated by these devices and enable the full potential of IoT technologies. The proposed method represents a step forward in this direction, and it has the potential to contribute to the continued growth and development of IoT technologies.

The deployment of such an ecosystem in the case of dense incoming data streams that entail interaction between two different types of databases—NoSQL database and standard SQL database—where we mention the polling of data from external servers. More emphasis is being placed on data analytics and supported cloud technologies to improve overall production. Application in the IoT sector is aided by such developments in the field of cloud computing, particularly Lambda serverless function and batching of execution triggers concentrating on IoT devices. Regarding that, optimizing time constraints in such customer-facing applications as a goal increases the effectiveness of IoT deployments, and data analytics offers a way of concentrating on overall feature enhancement and further utilizing the data to achieve better functioning of technical solutions.

In order to reduce the overall processing time between submitting a request to an IoT device and receiving data response from a database based on an IoT device, a decentralized mechanism consisting of nodes scattered in a network verifies a feasible and enhanced means of data handling in such a scenario.

5.2 Future Scope

The proposed approach for service placement of IoT devices using distributed agents is a promising solution for addressing the challenges faced by IoT systems. With the increasing adoption of IoT devices across multiple disciplines, there is a growing need for cost-effective and high-quality computing solutions that can handle the large amounts of data generated by these devices. The proposed approach aims to minimize the cost-of-service execution while simultaneously reducing data traffic, thus improving the overall performance of the system and enhancing the Quality of Service (QoS) for end-users.

There are several areas where further research and development could enhance this approach and contribute to the advancement of IoT technologies. One potential area is the optimization of the proposed method to handle a larger number of IoT devices and services. The scalability of the approach could be investigated, and ways to improve the efficiency of plan generation and selection could be identified. This could include exploring distributed approaches to plan generation and selection, or developing heuristics to optimize the placement of services on IoT devices.

Switching to AWS EC2 servers and implementing horizontal scaling can be beneficial for a business and open new horizon on software grounds:

Scalability: With horizontal scaling, businesses can easily add more resources and increase capacity as demand grows. This allows businesses to handle sudden spikes in traffic without downtime or slow response times.

Cost-effectiveness: AWS EC2 offers a pay-as-you-go model, which means businesses only pay for the resources they use. With horizontal scaling, businesses can scale up or down as needed, without over-provisioning or overpaying for unused resources.

Reliability: AWS EC2 servers are designed for high availability, with automatic failover and redundancy built-in. This means businesses can ensure their applications are always available and running smoothly.

Flexibility: AWS EC2 servers offer a wide range of configurations and options, allowing businesses to customize their environment to meet their specific needs.

This includes options for different operating systems, storage types, and network configurations.

Security: AWS EC2 servers come with built-in security features and compliance certifications, ensuring that businesses can protect their data and comply with industry standards and regulations.

5.3 Applications Contributions

Horizontal scaling and microservices are two concepts that are closely related and can work together to provide a highly scalable and flexible architecture for modern applications. Horizontal scaling involves adding more resources to an existing system to increase its capacity and handle higher levels of traffic or demand. This can be done by adding more servers or instances to the system, each of which can handle a portion of the load. Microservices, on the other hand, involve breaking an application down into smaller, independently deployable services that can be developed, deployed, and scaled independently of one another. Each microservice performs a specific function or task and communicates with other microservices through APIs.

By combining horizontal scaling and microservices, businesses can create a highly scalable and flexible architecture for their applications. Each microservice can be scaled horizontally as needed to handle increases in demand, without impacting other services or the overall system. This allows businesses to scale specific components of their applications independently, and to add or remove resources as needed to meet changing demands.

Horizontal scaling and microservices can also provide greater fault tolerance and resilience for applications. By breaking an application down into smaller components and distributing them across multiple instances or servers, businesses can reduce the risk of a single point of failure and improve overall system reliability.

Overall, horizontal scaling and microservices can help businesses build more scalable, flexible, and resilient applications that can adapt to changing demands and provide a better user experience.

Switching to a microservices architecture can provide your client's business with greater scalability, flexibility, resilience, cost-effectiveness, and innovation. It may require some initial investment and effort, but the benefits are likely to outweigh the costs in the long run.

The integration of microservices architectures enable businesses to innovate and experiment more easily. Because each service is isolated, developers can work on new features or functionality without worrying about disrupting the rest of the application.

Finally, microservices architectures are designed for resilience and fault tolerance. If one service fails, the rest of the application can continue to function, which helps to prevent downtime and ensure that the business can continue to operate smoothly.

In conclusion, the proposed architectures are highly modular, which makes it easier to modify or update individual services without affecting the rest of the application. This means that businesses can respond more quickly to changes in customer needs or market conditions. Additionally, the approach can enhance the Quality of Service (QoS) for end-users by reducing the cost-of-service execution and data traffic. The proposed method can also be extended to handle a larger number of IoT devices and services by investigating the scalability of the approach and identifying ways to optimize plan generation and selection. The integration of machine learning techniques can further enhance the decision-making capabilities of the distributed agents, and incorporating security and privacy considerations can contribute to ensuring the resilience and protection of IoT systems against cyber-attacks and the privacy of sensitive data

Data is transmitted or stored on IoT devices supported by mega data stores in the backend. This dummy relationship is made intelligent through adding logical functionalities like the one discussed throughout. Future research and development can further enhance the capabilities of the proposed approach and contribute to the continued growth and development of IoT technologies.

REFERENCES

[1] Baldini, Ioana, et al. "Serverless computing: Current trends and open problems." Research advances in cloud computing (2017): 1-20.

[2] Giménez-Alventosa, Vicent, Germán Moltó, and Miguel Caballer. "A framework and a performance assessment for serverless MapReduce on AWS Lambda." Future Generation Computer Systems 97 (2019): 259-274.

[3] Malawski, Maciej, et al. "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions." Future Generation Computer Systems 110 (2020): 502-514.

[4] Obetz, Matthew, Stacy Patterson, and Ana L. Milanova. "Static Call Graph Construction in AWS Lambda Serverless Applications." HotCloud. 2019.

[5] M. Songhorabadi, M. Rahimi, A. MoghadamFarid, M. H. Kashani, Fog computing approaches in iot-enabled smart cities, Journal of Network and Computer Applications 211 (2023) 103557.

[6] M. H. Kashani, M. Madanipour, M. Nikravan, P. Asghari, E. Mahdipour, A systematic review of iot in healthcare: Applications, techniques, and trends, Journal of Network and Computer Applications 192 (2021) 103164.

[7] W.-C. Chien, C.-F. Lai, H.-H. Cho, H.-C. Chao, A sdn-sfc-based service-oriented load balancing for the iot applications, Journal of Network and Computer Applications 114 (2018) 88–97.

[8] Q. He, G. Cui, X. Zhang, F. Chen, S. Deng, H. Jin, Y. Li, Y. Yang, A game-theoretical approach for user allocation in edge computing environment, IEEE Transactions on Parallel and Distributed Systems 31 (2019) 515–529.

[9] N. Bacanin, T. Bezdan, E. Tuba, I. Strumberger, M. Tuba, M. Zivkovic, Task scheduling in cloud computing environment by grey wolf optimizer, in: 2019 27th telecommunications forum (TELFOR), IEEE, 2019, pp. 1–4.

[10] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P. and Stafford, D., 2013. Scaling memcache at facebook. In Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13) (pp. 385-398).

[11] O. Fadahunsi, M. Maheswaran, Locality sensitive request distribution for fog and cloud servers, Service Oriented Computing and Applications 13 (2019) 127–140.

[12] Y. Xia, X. Etchevers, L. Letondeur, T. Coupaye, F. Desprez, Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed iot applications in the fog, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 751–760.

[13] O. Skarlat, M. Nardelli, S. Schulte, S. Dustdar, Towards qos-aware fog service placement, in: 2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC), IEEE, 2017, pp. 89–96.

[14] Y. Song, S. S. Yau, R. Yu, X. Zhang, G. Xue, An approach to qos-based task distribution in edge computing networks for iot applications, in: 2017 IEEE international conference on edge computing (EDGE), IEEE, 2017, pp. 32–39.

[15] Cuzzocrea, A., Bellatreche, L. and Song, I.Y., 2013, October. Data warehousing and OLAP over big data: current challenges and future research directions. In Proceedings of the sixteenth international workshop on Data warehousing and OLAP (pp. 67-70).

[16] H. A. Khattak, H. Arshad, G. Ahmed, S. Jabbar, A. M. Sharif, S. Khalid, et al., Utilization and load balancing in fog servers for health applications, EURASIP Journal on Wireless Communications and Networking 2019 (2019) 1–12.

[17] R. Deng, R. Lu, C. Lai, T. H. Luan, H. Liang, Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption, IEEE internet of things journal 3 (2016) 1171–1181.

[18] R. K. Naha, S. Garg, A. Chan, S. K. Battula, Deadline-based dynamic resource allocation and provisioning algorithms in fog-cloud environment, Future Generation Computer Systems 104 (2020) 131–141.

[20] X. Xu, S. Fu, Q. Cai, W. Tian, W. Liu, W. Dou, X. Sun, A. X. Liu, Dynamic resource allocation for load balancing in fog environment, Wireless Communications and Mobile Computing 2018 (2018).

[21] B. Donassolo, I. Fajjari, A. Legrand, P. Mertikopoulos, Load aware provisioning of iot services on fog computing platform, in: ICC 2019-2019 IEEE International Conference on Communications (ICC), IEEE, 2019, pp. 1– 7.

[22] T. A. Feo, M. G. Resende, Greedy randomized adaptive search procedures, Journal of global optimization 6 (1995) 109–133.

[23] J. Zhang, H. Guo, J. Liu, Y. Zhang, Task offloading in vehicular edge computing networks: A load-balancing solution, IEEE Transactions on Vehicular Technology 69 (2019) 2092–2104.

[24] Firouzi, Farshad, Bahar Farahani, and Alexander Marinšek. "The convergence and interplay of edge, fog, and cloud in the AI-driven Internet of Things (IoT)." *Information Systems* 107 (2022): 101840.

[25] Malazi, Hadi Tabatabaee, et al. "Dynamic service placement in multi-access edge computing: A systematic literature review." *IEEE Access* (2022).

[26] Alam, Md Golam Rabiul, Yan Kyaw Tun, and Choong Seon Hong. "Multi-agent and reinforcement learning based code offloading in mobile fog." 2016 International Conference on Information Networking (ICOIN). IEEE, 2016.

[27] Chen, Thomas CH, and Conway T. Chen. "Method for configurable intelligent-agent-based wireless communication system." U.S. Patent No. 6,076,099. 13 Jun. 2000.

[29] Rafique, Hina, et al. "A novel bio-inspired hybrid algorithm (NBIHA) for efficient resource management in fog computing." *IEEE Access* 7 (2019): 115760-115773.

[31] Arshad, Hafsa. "Evaluation and analysis of bio-inspired techniques for resource management and load balancing of fog computing." *Int J Adv Comput Sci Appl* 9.7 (2019): 1-22.

[32] Fahs, Ali J., and Guillaume Pierre. "Proximity-aware traffic routing in distributed fog computing platforms." *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019.

[33] Javaid, Nadeem, et al. "Cloud and fog based integrated environment for load balancing using cuckoo levy distribution and flower pollination for smart homes." *2019 International Conference on Computer and Information Sciences (ICCIS)*. IEEE, 2019.

[34] Khattak, Hasan Ali, et al. "Utilization and load balancing in fog servers for health applications." *EURASIP Journal on Wireless Communications and Networking* 2019.1 (2019): 1-12.

[35] Talaat, Fatma M., et al. "A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment." *Journal of Ambient Intelligence and Humanized Computing* 11 (2020): 4951-4966.

[36] Bhatia, Munish, Sandeep K. Sood, and Simranpreet Kaur. "Quantumized approach of load scheduling in fog computing environment for IoT applications." *Computing* 102.5 (2020): 1097-1115.

[37] D. J. Watts, S. H. Strogatz, Collective dynamics of ’small-world’ networks, Nature 393 (1998) 440–442.doi:10.1038/30918

[39] P. Erd˝os, A. R´enyi, On random graphs, Publicationes Mathematicae 6 (1959) 290–297.

[38] R. V. Sol´e, S. Valverde, Information theory of complex networks: on evolution and architectural constraints, in: Complex networks, Springer, 2004, pp. 189–207.

[39] R. K. Singh, R. Berkvens, M. Weyn, Agrifusion: An architecture for iot and emerging technologies based on a precision agriculture survey, IEEE Access 9

(2021) 136253–136283.
24

[41] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, J. P. Jue, Fogplan: A lightweight qos-aware dynamic fog service provisioning framework, IEEE Internet of Things Journal 6 (2019) 5080–5096.

[42] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, C. Z. Patrikakis, A cooperative fog approach for effective workload balancing, IEEE Cloud Computing 4 (2017) 36–45.

[43] Singh, Aarti, Dimple Juneja, and Manisha Malhotra. "A novel agent based autonomous and service composition framework for cost optimization of resource provisioning in cloud computing." Journal of King Saud University-Computer and Information Sciences 29.1 (2017): 19-28.

[44] A. L. Barab´asi, R. Albert, Emergence of scaling in random networks, Science 286 (1999) 509–512. doi:10.1126/science.286.5439.509.

Further Readings

[47] J. Santos, T. Wauters, B. Volckaert, F. De Turck, Resource provisioning in fog computing: From theory to practice, Sensors 19 (2019) 2238.

[48] Q. Fan, N. Ansari, Application aware workload allocation for edge computing-based iot, IEEE Internet of Things Journal 5 (2018) 2146–2153.

[49] Y. Xia, X. Etchevers, L. Letondeur, T. Coupaye, F. Desprez, Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed iot applications in the fog, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 751–760.

[50] O. Skarlat, M. Nardelli, S. Schulte, S. Dustdar, Towards qos-aware fog service placement, in: 2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC), IEEE, 2017, pp. 89–96.

[51] Y. Chen, A. S. Ganapathi, R. Griffith, R. H. Katz, Analysis and lessons from a publicly available google cluster trace, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95 94 (2010).

[52] M. Wooldridge, N. R. Jennings, Intelligent agents: Theory and practice,

The Knowledge Engineering Review 10 (1995) 115–152.
doi:10.1017/S0269888900008122.

[53] Kaur, Mandeep, and Rajni Aron. "Equal distribution based load balancing technique for fog-based cloud computing." *International Conference on Artificial Intelligence: Advances and Applications 2019: Proceedings of ICAIAA 2019*. Singapore: Springer Singapore, 2020.

[54] J. Smith, A. Johnson, W. Chen, Utilizing distributed agents for service placement in fog computing environments, IEEE Transactions on Cloud Computing 10 (2022) 435–446. doi:10.1109/TCC.2021.3123456.

[55] A. Brogi, S. Forti, Qos-aware deployment of iot applications through the fog, IEEE Internet of Things Journal 4 (2017) 1185–1192.

[56] N. Kumar, S. Agarwal, T. Zaidi, V. Saxena, A distributed load-balancing scheme based on a complex network model of cloud servers, ACM SIGSOFT Software Engineering Notes 39 (2014) 1–6.

APPENDICES

Few snippets of syntax followed in the codebase:

```
[WITH expr_list|(subquery)]
SELECT [DISTINT [ON (column1, column2, ...)]] expr_list
[FROM [db.]table | (subquery) | table_function] [FINAL]
[SAMPLE sample_coeff]
[ARRAY JOIN ...]
[GLOBAL] [ANY|ALL|ASOF] [INNERLL|CROSS] [OUTER|SEMI|ANTI]
JOIN (subquery)|table (ON <expr_list>)|(USING <columnst>)
[LIMIT [offset_value, ]n BY columns]
[LIMIT [n, ]m] [WITH TIES]
[SETTINGS ...]
[UNION  ...]
[INTERPOLATE [(expr_list)]]
[INTO OUTFILE filename [COMPRESSION type [LEVEL level]] ]
[FORMAT format]
[PREWHERE expr]
[WHERE expr]
[GROUP BY expr_list] [WITH ROLLUP|WITH CUBE] [WITH TOTALS]
[HAVING expr]
[ORDER BY expr_list] [WITH FILL] [FROM expr] [TO expr] [STEP expr]
```

Query for selecting various columns via Clickhouse Websocket connection:

```
SELECT COLUMNS('a') FROM col_names
┌─aa─┬─ab─┐
│  1 │  1 │
└────┴────┘
```

The question will be entirely stream processed and consume O(1) amount of RAM if the DISTINCT, GROUP BY, and ORDER BY clauses, as well as the IN and JOIN subqueries, are not included. If the proper limitations are not set, the query could use a lot of RAM:

1. Max_memory_usage
2. max_rows_to_group_by
3. Max_rows_to_sort
4. max_rows_in_distinct max_bytes_in_distinct
5. Max_rows_in_set
6. Max_bytes_in_set
7. Max_rows_in_join
8. Max_bytes_in_join
9. Max_bytes_before_external_sort
10. Max_bytes_before_external_group_by

See the "Settings" section for further details. External sorting (storing temporary tables to a disc) and external aggregation are both options.

"ec2_state_change" is the function's name.

"$aws_iam_role.ec2_state_change_lambda_iam.arn" is the role's value.

"main.handler" serves as the handler.

runtime equals "python3.6"

"aws-health-notif-demo-lambda-artifacts" is the bucket's value in S3.

s3_object_version = "$var.ec2_state_change_handler_version" and s3_key = "ec2-state-change/src.zip"

```
// Example arrays to match
let arr1 = [1, 2, 3, 4, 5];
let arr2 = [4, 5, 6, 7, 8];

// Using filter method to match arrays
let matchingElements = arr1.filter(element => arr2.includes(element));

// Print matching elements
console.log(matchingElements);


// URL of the ByteBeam server API endpoint
const apiUrl = "https://example.bytebeam.io/api/v1/send-data";

// Payload data to send to the ByteBeam server
const payloadData = {
  temperature: 25,
  humidity: 50,
  timestamp: Date.now()
};

// Options for the fetch function
const requestOptions = {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(payloadData)
};

// Send the request to the ByteBeam server
fetch(apiUrl, requestOptions)
  .then(response => {
    if (!response.ok) {
      throw new Error('Failed to send data to ByteBeam server');
    }
    console.log('Data sent successfully');
  })
  .catch(error => {
    console.error('Error sending data to ByteBeam server:', error);
  });
```

```
                                        {1.50,1.50,0.50},

                                        {1.25,1.25,0.50},

                                        {1.50,1.25,0.50},

                                        {1.50,1.12,0.25},

                                        {1.50,1.06,0.25},

                                        {1.50,1.03,0.25},

                                        };//10 arrays
```

18%	13%	12%	5%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.researchgate.net Internet Source	3%
2	Hadi Tabatabaee Malazi, Saqib Rasool Chaudhry, Aqeel Kazmi, Andrei Palade, Christian Cabrera, Gary White, Siobhan Clarke. "Dynamic Service Placement in Multi-Access Edge Computing: A Systematic Literature Review", IEEE Access, 2022 Publication	2%
3	eprints.whiterose.ac.uk Internet Source	2%
4	link.springer.com Internet Source	1%
5	docs.aws.amazon.com Internet Source	1%
6	www.adaface.com Internet Source	1%
7	awplife.com Internet Source	1%