Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Department of Information Engineering

Master Thesis in Computer Engineering

Design and Development of a Cloud-Based Data Lake and Business Intelligence Solution on AWS

Master Candidate

Christian Marchiori

Student ID 2078343

Supervisor

Prof. Gianmaria Silvello

University of Padova

Academic Year
2023/2024
Date : 4 december 2024

To my parents
and friends

Abstract

In a corporate context where rapid and accurate information analysis is crucial to improving operational performance and supporting data-driven strategic decisions, this thesis focuses on the design and implementation of a Business Intelligence (BI) system and a Data Lake for the company UNOX S.p.A., leveraging the technologies provided by the Amazon Web Services (AWS) platform.

The BI system developed allows the collection and analysis of data from various sources, helping to reduce inefficiencies, flag potential issues, identify new revenue streams, and pinpoint areas for future growth. Through the creation of a centralized Data Lake and the automation of data integration and analysis processes, it became possible to optimize the management of corporate information, providing a comprehensive and detailed view of business performance.

The system was designed to be easily accessible to various company teams, including Research and Development, IT, and other technical departments, thanks to intuitive tools such as AWS QuickSight. These tools enable a simple and visual interaction with the data without requiring specific knowledge of DBMS or query languages. This approach has made data more accessible at all company levels, promoting greater autonomy in their analysis.

The implementation made use of AWS components such as Amazon S3, Glue, Athena and QuickSight, ensuring scalable, secure, and fully automated data management. The final result is an integrated Business Intelligence system that significantly reduces the time required for analysis and reporting, enhancing decision-making capabilities and promoting a data-driven corporate culture.

Sommario

In un contesto aziendale in cui l'analisi rapida e precisa delle informazioni è cruciale per migliorare le performance operative e supportare decisioni strategiche basate sui dati, il presente lavoro di tesi riguarda la progettazione e implementazione di un sistema di Business Intelligence (BI) e di un Data Lake per l'azienda UNOX S.p.A, sfruttando le tecnologie messe a disposizione dalla piattaforma Amazon Web Services (AWS).

Il sistema di BI realizzato consente di raccogliere e analizzare dati provenienti da fonti diverse, contribuendo a ridurre le inefficienze, segnalare eventuali criticità, individuare nuovi flussi di ricavi e identificare aree di crescita futura. Attraverso la creazione di un Data Lake centralizzato e lautomazione dei processi di integrazione e analisi dei dati, si è reso possibile ottimizzare la gestione delle informazioni aziendali, garantendo una visione complessiva e dettagliata delle prestazioni.

Il sistema è stato progettato per essere facilmente accessibile anche da team aziendali, tra cui Ricerca e Sviluppo, IT e altri reparti tecnici, grazie all'uso di strumenti intuitivi come AWS QuickSight, che consentono un'interazione semplice e visiva con i dati, senza necessità di conoscenze specifiche in DBMS e linguaggi di query. Questo approccio ha permesso di rendere i dati fruibili a tutti i livelli aziendali, favorendo una maggiore autonomia nella loro analisi.

L'implementazione ha utilizzato componenti AWS quali Amazon S3, Glue, Athena e QuickSight, garantendo una gestione scalabile, sicura e completamente automatizzata dei dati. Il risultato finale è un sistema integrato di Business Intelligence che riduce significativamente i tempi di analisi e reporting, migliorando la capacità decisionale e promuovendo una cultura aziendale data-driven.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

BI Business Intelligence

AWS Amazon Web Services

B2B Business-to-Business

IoT Internet of Things

ETL Extract-Transform-Load

KPI Key Performance Indicator

CIO Chief Information Officer

RDBMS Relational Database Management System

MVCC Multi-Version Concurrency Control

JSON JavaScript Object Notation

DDC Data Driven Cooking

EC2 Elastic Compute Cloud

S3 Simple Storage Service

YAML YAML aint markup language

HDFS Hadoop Distributed File System

VPC Virtual Private Cloud

IAM Identity and Access Management

RDS Relational Database Service

ORC Optimized Row Columnar

SQL Structured Query Language

CSV Comma Separated Values

ASL Amazon States Language

HTTP HyperText Transfer Protocol

ACID Atomicity, Consistency, Isolation, Durability

JDBC Java Database Connectivity

CDC Change Data Capture

DMS Database Migration Service

SDK Software Development Kit

RAM Random Access Memory

CPU Central Power Unit

ORM Object-Relational Mapper

SNS Simple Notification Service

API Application programming interface

I/O Input/Output

DPU Data Processing Units

1 Introduction

1.1 The company

This thesis was conducted in collaboration with Unox S.p.A., a leading company in the professional cooking ovens market. Unox manufactures various oven series and distributes its products to over 130 countries worldwide. In addition to its product offerings, the company provides after-sales services, including cooking training, customer support, and technical assistance.

1.1.1 Company Profile

Unox S.p.A. has been active since 1990, specializing in the production of professional appliances for the catering and baking industries. As a product-focused company, its primary emphasis is on manufacturing and customer support for its ovens, rather than software development. Unox operates in the Business-to-Business (B2B) sector, serving a diverse range of clients, from small bakeries to large restaurant chains and catering centers.

Initially, the company capitalized on a market gap by producing ovens primarily for suppliers of frozen croissants in Southern Europe. These products were mainly provided to small retailers through lease agreements with large suppliers. Over time, Unox shifted its focus to producing higher-quality products for direct sale to end-users. This strategic change allowed the company to expand into Northern Europe and laid the groundwork for further growth in other continents.

Today, Unox offers a wide range of products, reflecting its evolution over the years. Recently, the company has increasingly focused on the digital advancement of its offerings, aiming to provide customers with a more comprehensive and satisfying experience. The company's flagship models now feature touch-screen panels, voice control, remote operation, data management for large companies, and other advanced features.

To meet the demands of this digital shift, Unox began developing software in-house to support these services, expanding its workforce to include software developers. The company currently has two dedicated software development teams and a Research and Development (R&D) team.

Despite its growth into a multinational corporation, Unox has remained family-owned. Production was always mainly based in Italy until a new production site was opened in the US in 2022. Since the beginning, international expansion has been focused on commercial operations, with the company establishing sales offices worldwide.

Unox employs a high level of vertical integration, producing most of its components in-house or through subsidiaries.

1.1.2 Software Development at Unox

Despite being a company primarily focused on manufacturing high-quality professional ovens, software development plays a critical role in Unox's operations. The digital transformation of its products, combined with the need for integrated connectivity and advanced functionalities, has made software a cornerstone of the company's offerings. Today, Unox ovens are fully digitized and connected, featuring capabilities such as remote control via mobile applications, data export for performance analysis, and integration with external systems through APIs. This shift has led to the formation of specialized software development teams that ensure Unox ovens remain at the cutting edge of technological innovation.

Unox currently operates with four distinct software development teams, each with a specialized focus:

- **IT Team**: This team is responsible for the internal infrastructure that supports Unox's daily operations. Their work includes managing the company's network, ensuring data security, and maintaining the systems that enable smooth communication and operational efficiency across all departments.

- **R&D Team**: Unox's Research and Development team focuses on innovation, developing new technologies for ovens to improve performance, energy efficiency, and user experience. This team collaborates with various departments to drive the technical evolution of Unox's products.

- **Software Developer Team**: This team is at the heart of the technical development process and is responsible for creating, maintaining, testing, and documenting the algorithms that optimize oven performance. Their tasks include defining technical specifications in collaboration with other teams, implementing reliable and efficient solutions, and supporting field technicians to resolve software-related issues. From initial concept to final release, the Software Developer team ensures that each line of code contributes to the operation of Unox products.

- **Digital Experience Team**: As part of the company's push towards a fully connected ecosystem, the Digital Experience team focuses on developing cloud-based applications, both for web and mobile platforms, and managing REST APIs. They are responsible for creating digital interfaces that allow users to remotely control and monitor ovens, manage data streams from connected devices globally, and integrate Unox products with other systems. Additionally, this team designs and maintains the cloud infrastructure, ensuring the reliability and scalability of Unox's digital services. They collaborate with data scientists to extract valuable insights from the vast amounts of telemetry data produced by the ovens. They also coordinate closely with the UI/UX team to deliver intuitive user experiences. This is the team I have been a part of during my internship, where I contributed to the development and enhancement of Unox's digital services, helping to bridge the gap between product performance and user interaction.

Each of these teams plays a critical role in ensuring that Unox continues to lead the market, not only with its physical products but also through its advanced digital offerings.

1.2 Initial problem

In the modern business environment, the ability to access information quickly and accurately is a key factor for competitiveness. Strategic decisions rely on precise analysis, which requires not only access to data but also appropriate tools for managing and interpreting it. Without the implementation of a structured BI system, such as the one developed in this project, Unox faces several challenges that limit the effectiveness and efficiency of its data extraction and analysis processes, impacting the entire company structure.

The main issue concerns the way data is requested, processed, and distributed within the company. Specifically, when an employee from a department like Research and Development, IT, or a technical division needs data for specific analyses, they turn to the Digital Experience team, which is designated to manage access to the company's databases and has greater expertise in running complex queries. While other teams may have some knowledge in this area, the Digital Experience team is responsible for overseeing and managing these processes. However, this approach slows down the flow of information and affects overall company efficiency, creating bottlenecks in decision-making processes.

The data extraction process involves several labor-intensive stages: the Digital Experience team must first understand the nature of the problem, identify the relevant tables and data, envision the final output of the analysis, and then develop a TypeScript script to access the databases, execute queries, and transform the data into a usable format for analysis. This "ad hoc" approach, while effective for specific requests, requires a significant amount of time and resources. Each request for analysis or reporting typically involves hours of work from an expert, limiting the company's ability to respond quickly to new market demands or opportunities.

In some cases, automated scripts are developed to reduce the repetitiveness of this process, executing periodic extractions and sending the data via email to relevant stakeholders. However, while this approach is useful, it remains limited. First, each automated script must be specifically developed for each case, leading to development and maintenance costs, along with cloud resource execution costs (such as AWS Lambda used to automate the periodic execution of these scripts). Additionally, these automations only cover a small portion of the company's overall needs and lack the flexibility to quickly respond to more complex or unexpected analysis requests.

The current working model also strongly limits the autonomy of non-technical teams. Many employees, despite needing data to improve their analyses and make informed decisions, are unable to directly access the information, as using advanced query languages or interacting with complex databases is beyond their expertise. This not only increases the workload for the Digital Experience team but also slows response times and decision-making, negatively impacting overall operational efficiency.

Other motivations behind this project include the need to improve data governance and integration between different information silos. In an environment where data is fragmented across various systems and databases, it becomes difficult to obtain a unified and coherent view of business performance, identify inefficiencies, or explore new growth opportunities. Implementing a BI system, supported by an AWS-based Data Lake, overcomes these barriers, improving the management of company data and simplifying real-time access to information.

In summary, the main reasons for implementing a new Business Intelligence system are:

- **Simplifying data access for non-technical teams**: Creating an interface and tools that enable employees without advanced technical skills to perform analysis and reporting independently.
- **Data integration**: Overcoming data fragmentation and ensuring a coherent integration of information from various systems, facilitating collaboration and data-driven decision-making.
- **Improving operational efficiency**: Optimizing the use of the Digital Experience team's resources, reducing the workload related to ad hoc requests, and allowing them to focus on higher-value tasks.
- **Reducing data extraction and analysis times**: Eliminating bottlenecks and automating repetitive processes to allow different teams to independently access relevant information.
- **Scalability and flexibility**: Adopting a scalable and flexible platform, such as AWS, to efficiently manage large volumes of data and quickly adapt to the company's evolving needs.

With these premises, the project aims to revolutionize the way the company manages, accesses, and analyzes data, significantly improving the effectiveness of decision-making processes.

1.3 Objectives

The primary objective of this project is to develop a unified and flexible system for managing the data generated by industrial ovens, without the need to create separate workflows for different stages of the process. The solution must ensure a consistent approach both during the bulk load phase, which involves large-scale data imports, and in the subsequent operational phase, where smaller but more frequent data updates are managed. This requires the design of a data ingestion infrastructure that can dynamically adapt to different data volumes, ensuring efficiency and ease of maintenance.

Another key goal is to ensure the efficiency of the system in terms of resource usage, with a strong focus on cost optimization. Given that the architecture relies heavily on several managed services from AWS, such as Glue, Lambda, and Step Functions, it is crucial to minimize resource consumption, reducing the execution time and memory usage of various tasks. This helps to keep operational costs in check, as AWS pricing is often directly tied to the resources utilized.

An additional objective of the project is to provide a system that integrates a query engine for data analysis and a dashboard for Key Performance Indicator (KPI) visualization. Queries should be executed using tools like AWS Athena, which allows for SQL queries to be run directly on data stored in Amazon S3, leveraging a flexible and scalable system without the need for complex database setups. The interactive dashboards, created using AWS QuickSight, will allow users to visualize data intuitively, monitor key metrics, and generate customized reports.

Furthermore, the system must ensure that the data is kept synchronized with the production databases, offering up-to-date access to information for reporting and monitoring purposes. A secondary, yet innovative, optional objective is the exploration of generative AI techniques to automate the creation of dashboards in AWS QuickSight, further simplifying the user experience and enhancing the overall efficiency of the data visualization process.

In summary, the project aims to build a solution that is flexible, efficient, and capable of supporting data analysis and visualization effectively, with a focus on automation, resource optimization, and ease of use.

1.4 Proposed Solution

The proposed system is built on a scalable, automated architecture using AWS cloud technologies, enabling near real-time access to up-to-date information for reporting and analysis. The overarching goal is to create a data pipeline that reliably extracts, transforms, stores, and makes data available for analysis with minimal human intervention.

The system is designed to automate the entire data lifecycle, from the extraction of raw data to its transformation into structured formats, storage in a data lake, and final usage for analytics. Automation was a critical requirement, as the high frequency and volume of data generated by the ovens demanded a process that could run continuously without manual oversight. Additionally, ensuring that the data is always current and accessible for users was a priority, which required careful orchestration and scheduling of data extraction and processing tasks.

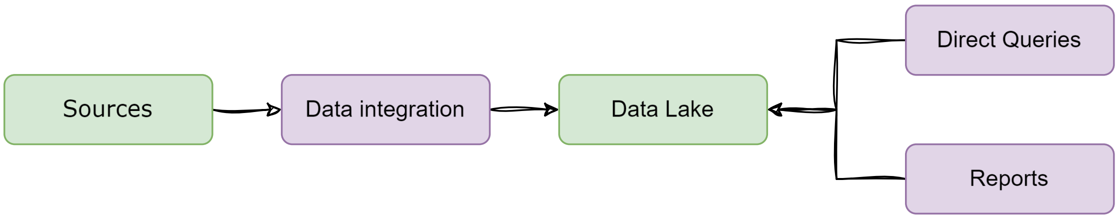

Figure 1.1: System general flow.

The primary sources of data include two distinct databases: **PostgreSQL**, which stores general operational data about the ovens or users' management data, and **MongoDB**, which holds Internet of Things (IoT) data from sensors, alarms, and other event-driven information. Since these databases have different structures and serve different purposes, the solution had to accommodate specific workflows for each.

For the PostgreSQL database, AWS Glue was selected as the main Extract-Transform-Load (ETL) service. Glue is a fully managed, serverless ETL tool that simplifies data preparation by running Python or Scala scripts without the need for managing servers. In this context, Glue is responsible for connecting to the PostgreSQL database, extracting the necessary tables, transforming the data into a columnar file format, and loading it into Amazon S3 for long-term storage. One of the key challenges was ensuring that the system did not reprocess previously extracted data during subsequent extractions. This issue was addressed using Glue's Bookmarks feature, which tracks the progress of each job by recording the last data row processed. When the ETL job runs again, it starts from where the last job left off, ensuring only new data is ingested.

For the MongoDB database, which stores unstructured IoT data, the extraction process required a custom approach, as Glue's Bookmarks feature does not support MongoDB. To address this, the system leverages AWS Lambda, a serverless computing service that runs code in response to specific events. A master Lambda function orchestrates multiple worker Lambdas, each of which processes data from a specific set of devices (ovens). This system distributes the workload efficiently, allowing for a scalable and flexible data extraction pipeline. Each worker Lambda extracts, filters, and transforms the data, then formats it in Parquet and stores it in Amazon S3. By employing this distributed architecture, the system ensures that even large volumes of IoT data are processed efficiently and in parallel.

The extracted data is stored in a data lake on Amazon S3, organized into three distinct layers:

- raw,
- curated,
- and analytics.

These layers reflect the level of transformation and aggregation applied to the data [9, 16]. In the raw layer, data is stored in its original form, directly after extraction, without any significant transformations. The curated layer includes data that has undergone partitioning, formatting, and compression to optimize performance for querying. Finally, in the analytics layer, data is pre-aggregated to facilitate specific use cases, such as recurring reports or complex queries, improving the efficiency of downstream analytics.

To streamline the management of this multi-stage process, AWS Step Functions are used to orchestrate the entire workflow. Step Functions allow the system to define and automate the execution of each task in the pipeline, ensuring that each job runs in the correct sequence and avoiding conflicts between components. Additionally, the frequency of execution can be easily configured, allowing for flexible scheduling of data extraction and processing based on business requirements.

Once the data is processed and stored in S3, it is cataloged using the AWS Data Catalog. The Data Catalog consolidates metadata for all the tables and files stored in S3, making it easier for other AWS services to reference and query the data. This unified metadata management system allows users to interact with the data seamlessly, without needing to define the underlying storage paths or configurations manually.

For querying and analyzing the data, the solution integrates two key tools: AWS Athena and AWS Quicksight. AWS Athena is an interactive query service that enables users to run SQL queries directly on the data stored in Amazon S3, leveraging the metadata defined in the Data Catalog. This provides a powerful tool for on-demand analysis without requiring the setup of additional databases or data warehouses. AWS Quicksight, on the other hand, is a BI tool that allows users to create interactive dashboards, visualizations, and reports. By integrating Quicksight, the system enables non-technical users to explore the data, generate insights, and produce business reports with an intuitive interface.

In conclusion, the proposed solution offers a fully automated, scalable, and flexible architecture for managing and analyzing the data generated by industrial ovens. It leverages advanced cloud services to ensure that data is continually updated, efficiently processed, and readily available for users, while minimizing the need for manual intervention. This approach not only improves the overall efficiency of data management but also enhances the ability to derive meaningful insights from large volumes of industrial data.

1.5 Outcomes

The results of the project were highly positive, with the implemented system proving to function effectively and reliably. During the testing phase, a snapshot of the production database was used, allowing the entire data ingestion pipeline to be validated and the system's performance to be tested in a realistic environment. Specifically, the initial ingestion of IoT data was completed using a snapshot of the production database up to September 2, 2024, ensuring that all historical data was successfully loaded into the data lake. Once ready for use with the live production databases, the transition can be easily achieved by running a pre-configured script that adapts the custom bookmarks, updates the database credentials and connection settings, and activates the scheduler that automates the regular execution of the system.

One of the key advantages of the implemented solution is the significant reduction in on-demand query times, with an estimated improvement of X% compared to the previous approach. Thanks to the architecture based on Amazon S3, Athena, and AWS Quicksight, queries are now much faster and more efficient, making the data readily available for analysis without substantial delays.

Another significant benefit is the simplification of access to the query platforms. Before the implementation, anyone outside the Digital Experience team had to either obtain access to a specific database or install a GUI client and have the necessary credentials for each database. With the new system, data access is managed directly through the AWS console, provided the user has an account with the necessary permissions to use query or reporting tools. If an employee does not have an AWS account, it can be quickly created by one of the company's software teams.

From a cost perspective, the monthly operational cost of the system has been estimated at around 400$, with fluctuations based on actual usage. The initial ingestion of all historical data, going back to 2015, incurred a one-time cost of approximately 3000$, primarily due to AWS Lambda and AWS Glue services, which were essential for populating the data lake.

While the overall results are highly satisfactory, there is still room for improvement, particularly in optimizing the efficiency of AWS Lambda functions. Reducing the execution times and memory usage of the Lambda functions could significantly lower both the computation costs and the query response times. This would enhance the overall system efficiency and further reduce the data ingestion costs.

1.6 Outline

The subsequent chapters are structured to explore the necessary backgrounds, describe the entire system from a technical point of view, prove the application functionalities through a main use case, perform other experiments and evaluate the system's performance.

In particular, the second chapter provides an overview of the fundamental technologies and concepts needed to comprehend the project. It delves into the tools used for data management, file formats, and the cloud services employed during development.

The third chapter focuses on the detailed design and implementation of the system architecture, explaining the role and interactions of each component. Special attention is given to the methodologies used for data ingestion, transformation, and storage, with an emphasis on the scalability and automation features facilitated by the selected AWS technologies.

The fourth chapter validates the functionality and performance of the system through real-world use cases, experiments, and benchmarking activities. Key aspects such as cost efficiency, data partitioning strategies, and the optimization of query execution times are analyzed in depth.

Finally, the fifth chapter concludes the thesis by summarizing the results achieved, discussing the limitations of the implemented solution, and suggesting potential directions for future research and system enhancement. Together, these chapters provide a structured exploration of the design, implementation, and evaluation of a cloud-based data management and business intelligence solution.

2 Background

2.1 Data Management

2.1.1 PostgreSQL

PostgreSQL, often called Postgres, is a robust open-source Relational Database Management System (RDBMS) known for its flexibility, stability, and full compliance with SQL standards. It supports a wide range of advanced features, such as complex queries, foreign keys, views, triggers, and stored procedures. One of its distinguishing characteristics is its support for both structured and semi-structured data, including JavaScript Object Notation (JSON), making it suitable for modern applications that need to manage different data formats. PostgreSQL also offers powerful indexing techniques (e.g., B-tree, GIN, GiST) to optimize query performance, as well as Multi-Version Concurrency Control (MVCC), which enables high transaction throughput without locking issues, allowing multiple users to interact with the database simultaneously.

In the project developed at UNOX S.p.A., PostgreSQL is used to manage key operational data related to industrial ovens. This includes information about companies, devices (ovens), recipes, and device groupings. These datasets are critical for the Data Driven Cooking (DDC) platform, which leverages the stored information to enhance oven performance, improve operational processes, and provide customers with detailed insights.

PostgreSQL is considered a popular choice for many organizations since is open-source, has extensive community support, and is involved in continuous development. It offers a highly customizable solution for both small-scale applications and large enterprise systems.

2.1.2 MongoDB

MongoDB is an open-source NoSQL database designed to handle large volumes of unstructured and semi-structured data. Unlike traditional relational databases like PostgreSQL, described in the previous section 2.1.1, which store data in structured tables with fixed schemas, MongoDB uses a flexible, document-based model. Data is stored in JSON-like BSON (Binary JSON) format, allowing for dynamic, schema-less storage where each document can have a different structure. This makes MongoDB ideal for use cases where data is heterogeneous or rapidly changing, as it does not require predefined schemas or rigid structures like a relational database.

In contrast to PostgreSQL, which excels at managing structured, relational data with well-defined relationships, MongoDB is optimized for handling data that doesn't fit into regular table structures, such as hierarchical or nested data. Additionally, MongoDB offers easy horizontal scaling, distributing data across multiple servers to handle high write loads, making it well-suited for applications that generate large amounts of real-time data.

At UNOX, MongoDB is used to store IoT data generated by the industrial ovens. This includes sensor readings like temperature, humidity, alarms, and detailed records of cooking processes. The flexible schema in MongoDB allows the system to efficiently capture and store a wide variety of data points, which may differ from oven to oven, or even from one cooking session to another. This dynamic approach enables real-time monitoring and analysis of oven performance, helping to ensure that the ovens operate efficiently and providing actionable insights based on the data collected.

2.1.3 TablePlus

TablePlus is a versatile database management tool designed to simplify working with relational databases. It supports a wide range of databases like PostgreSQL, MySQL, and SQLite, making it a go-to solution for developers who need to manage different database systems through a single interface. The most appreciated feature of TablePlus is its intuitive, streamlined interface, which enables users to easily run queries, edit data, and manage tables without needing to rely heavily on complex command-line tools.

The tool also emphasizes security, providing secure connections via SSH and SSL, which is crucial when dealing with sensitive data. It is lightweight and fast, ideal for those who require quick access to data and the ability to make changes efficiently. Other features that distinguish Table Plus among database professionals are the ability to preview and revert changes, multi-step undo, and export options enhance productivity, and reduce the risk of errors, making TablePlus a popular choice among database professionals.

2.1.4 Studio 3T

Studio 3T is a dedicated tool for managing MongoDB databases, offering a range of features designed to make interacting with NoSQL data easier. Unlike general database tools, Studio 3T is optimized for MongoDB's document-based structure. It provides a visual interface for building and executing queries, which is especially useful for those who prefer not to write complex MongoDB queries by hand.

Key features include the ability to migrate data, visualize aggregation pipelines, and easily manage indexes and collections. Studio 3T also allows users to translate SQL queries into MongoDB's query language, making the transition for those familiar with SQL-based databases smoother. It is a powerful tool for handling MongoDB-specific tasks, enabling developers and administrators to efficiently manage large datasets in a more user-friendly environment.

2.1.5 Apache Spark and Hadoop

Apache Spark and Hadoop are prominent frameworks for distributed data processing, widely adopted in big data environments to handle large datasets efficiently. **Apache Hadoop** provides the foundational infrastructure for storing and processing big data across clusters of computers. At its core, Hadoop consists of the Hadoop Distributed File System (HDFS) and the MapReduce programming model. HDFS breaks files into blocks and distributes them across multiple servers, providing fault tolerance and redundancy.

Apache Spark builds on Hadoop's capabilities with in-memory processing, which enhances speed, especially for iterative tasks. Spark includes several specialized libraries, including Spark SQL for SQL-based data processing, Spark

Streaming for real-time analytics, and MLlib for machine learning. Unlike Hadoop's batch-oriented processing, Spark's in-memory architecture enables faster data processing and is suited to a broader range of tasks, including machine learning and stream processing. While Spark was originally designed for Java, it also supports a Python API known as PySpark.

In combination, Hadoop and Spark offer a robust ecosystem for big data management, with Hadoop handling data storage and Spark providing the advanced computational framework needed for modern analytics. Thus, Spark is ideal for tasks requiring fast data processing and interactive querying, as it offers flexibility and high performance, while Hadoop remains foundational in organizing and storing large-scale data.

2.1.6 Prisma

Prisma is an open-source Object-Relational Mapper (ORM) tool that simplifies database management for JavaScript and TypeScript applications. By abstracting SQL complexities, Prisma provides a high-level Application programming interface (API) to interact with popular databases like PostgreSQL, MySQL, and SQLite. Developers define data models using a schema file, from which Prisma generates optimized SQL queries, ensuring type safety and enhancing code reliability.

Prisma also includes tools to manage database schema migrations, which simplify the process of updating or versioning database structures. This schema-driven approach not only reduces manual intervention in query generation but also ensures that database interactions are type-safe, minimizing runtime errors. Prisma is particularly advantageous for applications that handle complex data relationships and require frequent database interactions, as it enables more efficient and intuitive database management, improving overall developer productivity.

2.2 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform that provides a broad set of services, including computing power, storage, and networking, through the internet. Businesses and developers use AWS to build and run a wide variety of applications, from simple websites to complex enterprise systems. One of its key benefits is the ability to scale resources up or down based on demand, which eliminates the need for managing physical hardware and allows for greater flexibility. AWS operates on a global network of data centers, offering services that ensure high availability and reliability for users across different regions. AWS enables companies to access the resources they need with minimal upfront investment, through its modular services, such as EC2 for virtual servers and Simple Storage Service (S3) for data storage. AWS also promotes a pay-as-you-go pricing model, where users only pay for the resources they use, making it affordable for both small startups and large enterprises. Its extensive suite of tools allows users to implement everything from basic hosting solutions to advanced analytics and AI models.

Below, some of the AWS tools used to build the system's architecture will be presented.

2.2.1 Identity and Access Management (IAM)

AWS Identity and Access Management (IAM) allows you to securely manage access and permissions to AWS resources. With IAM, you can create and control users, groups and roles, assigning detailed permissions to specify who can access what resources and with what privileges. For example, you can grant a user access to certain AWS services or specific actions on a database, while restricting other operations.

IAM uses the concept of "least privilege", allowing you to configure very precise accesses and monitor activities via logs. It is a fundamental tool for ensuring security and control in the AWS infrastructure.

2.2.2 Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) is a main AWS service that allows to launch and manage virtual servers in the cloud, called instances. EC2 offers a range of instance types to meet different computational needs, allowing users to select the optimal balance of Central Power Unit (CPU), memory, and storage for their workloads. Therefore, when you create an instance it is possible to choose the specific hardware configurations that best suit their specific requirements, offering flexibility in performance and cost management. EC2 enables on-demand access to computing power, with the ability to deploy and manage instances without the need for physical servers.

EC2 also integrates well with other AWS services, allowing to build reliable and scalable cloud-based systems. It supports multiple operating systems, such as Linux and Windows, and offers both temporary and persistent storage options, depending on the type of application.

For instance, in this project, an EC2 instance was used to host a snapshot of the production database, allowing safe testing and development with real data without affecting the live environment. This demonstrates how EC2 can help isolate and manage testing or development environments effectively.

2.2.3 Relational Database Service (RDS)

Amazon RDS is an easy-to-manage relational database service optimized for total cost of ownership. It is easy to configure, use and scale according to demand. Amazon RDS automates several database management tasks such as provisioning, configuration, backups and patching. Amazon RDS allows customers to create a new database in some minutes and offers the flexibility to customize databases to their needs by choosing from 8 engines and 2 deployment options. Customers can optimize performance with features such as Multi-AZ with two readable standbys, optimized writes and reads, and AWS Graviton3-based instances, with a choice of pricing options to manage costs effectively.

You have the option to enable automated backups or create manual backup snapshots as needed. These backups can be used to restore your database efficiently and reliably using Amazon RDS's restoration process.

Beyond the security features included in your database package, you can manage access by utilizing AWS IAM to assign user roles and permissions. You can also enhance database protection by placing them within a Virtual Private Cloud (VPC). Moreover, security groups can be used to control what IP addresses or Amazon EC2 instances can connect to your databases.

Table 2.1 compares the database management models between Amazon EC2 and Amazon RDS, highlighting customer and AWS responsibilities for various features.

2.2.4 Lambda

AWS Lambda is a serverless computing service that allows code to be executed without having to manage servers or infrastructure directly. Launched

Feature	**EC2 management**	**RDS management**
Application optimization	Customer	Customer
Scaling	Customer	AWS
High availability	Customer	AWS
Database backups	Customer	AWS
Database software patching	Customer	AWS
Database software install	Customer	AWS
Operating system (OS) patching	Customer	AWS
OS installation	Customer	AWS
Server maintenance	AWS	AWS
Hardware lifecycle	AWS	AWS
Power, network, and cooling	AWS	AWS

Table 2.1: Comparison of Amazon EC2 and Amazon RDS database management models

in 2014, Lambda allows developers to execute functions in response to specific events, such as changes to a database, HTTP requests, or file updates in an Amazon S3 bucket.

The serverless model eliminates the need for manual provisioning, management, or scaling of resources, as Lambda takes care of these tasks automatically. Users only pay for the code execution time, measured in milliseconds, and the number of requests, making it a highly cost-efficient option for many applications. The service supports several programming languages, including Python, Node.js, Java, Go, and C#, making it flexible for a wide range of use cases, such as real-time data processing, application monitoring, and automation of repetitive tasks.

To generate a lambda, first, you create your function by uploading your code and choosing the memory, timeout period, and AWS IAM role. Then, you specify the AWS resource to trigger the function, which can be a particular Amazon S3 bucket, Amazon DynamoDB table, or Amazon Kinesis stream. When the resource changes, Lambda will run your function, launching and managing the compute resources as needed to keep up with incoming requests.

2.2.5 Glue

AWS Glue is a fully managed extraction, transformation and loading service, designed to facilitate the preparation and integration of data for analysis. AWS Glue automates the processes of data discovery, cataloguing, cleaning, transformation and movement between different sources such as data lakes, relational databases and other storage resources. The service is designed to simplify the work of preparing data for analysis and modeling by eliminating the need to configure and manage servers. The three main features offered by Glue are:

Jobs ETL

ETL Jobs in AWS Glue are the main operating units that perform the extraction, transformation and loading processes. An ETL job reads data from a source, transforms it as required (such as cleaning, merging or format conversion) and loads it into a destination, such as a data warehouse or data lake. AWS Glue automatically generates Scala or Python code to perform these operations, but also offers the possibility of customising scripts. These jobs can be executed on demand or scheduled at regular intervals, integrating with other AWS resources.

Data Catalog

The AWS Glue Data Catalogue is a centralised metadata repository that organises and manages information on datasets from different sources. It stores data schemas, formats and partitions, facilitating access and queries via tools such as Amazon Athena and Amazon Redshift, without requiring manual configuration.

Crawler

AWS Glue Crawlers automate data discovery and cataloguing by analysing sources to automatically identify schemas and partitions. They update or create tables in the data catalogue, reducing manual work and simplifying metadata management.

In summary, AWS Glue, through the use of Jobs ETL, Data Catalogue and Crawler, provides a powerful and scalable platform for large-scale data management, optimising workflows and ensuring easy integration with other AWS services.

2.2.6 Athena

Amazon Athena is an interactive query service that allows data analysis directly on files stored in Amazon S3 using standard SQL. Athena is a serverless solution, which means that it does not require the management of infrastructure or servers: users only pay for the queries executed, based on the volume of data processed.

The service is optimised to work with large datasets and common data formats such as CSV, JSON, Parquet and Apache ORC[1], Apache Iceberg[2] allowing structured and semi-structured information to be analysed efficiently. Thanks to the AWS Glue Data Catalogue described in section 2.2.5, Athena can easily access previously defined metadata and schemas, reducing the time needed for data preparation.

Athena lends itself well to ad-hoc data lake analysis, reporting and data monitoring scenarios, without having to load data into a traditional database. Its features make it ideal for big data analytics environments, where speed of execution and ease of use are crucial, and it integrates well with other AWS services, such as Amazon QuickSight for data visualisation.

2.2.7 Simple Storage Service (S3)

Amazon S3 is an object-based storage solution specifically designed for handling large volumes of data with high durability and availability. It stores data as objects in "buckets" and allows users to define access controls and lifecycle policies for efficient data management. S3 includes several storage classes, such as Standard, Intelligent-Tiering, and Glacier, which are suited to different access patterns and cost needs. For instance, it's possible to archive infrequently accessed data to lower-cost storage classes over time, allowing organizations to optimize their storage expenses. It supports direct querying through Amazon Athena and enables efficient ETL processes via AWS Glue.

Data is automatically scaled and protected, with robust security measures including encryption options and access management through IAM. Therefore, S3 enables real-time data analysis and is widely used as the foundation of data lakes.

[1] `https://orc.apache.org/`
[2] `https://iceberg.apache.org/`

2.2.8 QuickSight

Amazon QuickSight is a cloud-based BI service from AWS, designed to create dashboards, interactive visualisations and reports from large volumes of data. QuickSight empowers users to analyse data in a simple and intuitive way, providing tools to create charts, tables and advanced visualisations that help make data-driven decisions. The tool supports two modes of data access:

- SPICE (Super-fast, Parallel, In-memory Calculation Engine) mode, which uses an in-memory engine for fast analysis performance and is available for a fee,
- and Direct Query mode, which is free but generally slower, as it queries directly on data sources.

QuickSight allows dashboards to be shared with other users and offers support for access from mobile devices, making it a flexible tool for real-time data analysis and visualisation.

Finally, one of QuickSight's most innovative features is its integration with generative artificial intelligence, which allows analyses to be built quickly via natural language prompts. Users can enter simple questions or queries in natural language and get automatically generated charts, tables and insights. This makes the analysis of datasets more accessible even for those without technical expertise, speeding up the decision-making process.

2.2.9 Step Functions

AWS Step Functions is a fully managed workflow orchestration service that allows different AWS services to be coordinated in sequential or parallel workflows. Using AWS Step Functions, complex processes can be defined as states, with each step in the workflow representing an activity, such as executing a job, waiting for an event or calling the API of any AWS service, such as AWS Lambda, Amazon S3 or AWS Glue.

The service uses a visual representation based on a state machine model, allowing users to build and monitor workflows intuitively. Alternatively, workflows can be defined via Amazon States Language (ASL), a JSON language that allows the user to specify the behavior of each state, conditions for passing between states, and error and exception handling criteria.

AWS Step Functions is ideal for automating distributed processes such as data pipelines, application management and microservice orchestration. It also enables workflows with error handling, retries and conditional operations, ensuring process reliability and resilience. Its extensive support for long-running tasks makes it a powerful tool for coordinating the execution of complex tasks in a secure manner.

2.2.10 Event Bridge

AWS EventBridge is a fully managed event routing service, designed to connect applications using real-time event streams. It enables the creation of event-based architectures, where various services or applications automatically react to events generated by other AWS applications or services. Each event represents an action or change of state, such as changes in a database or updates in an S3 bucket, and is routed to appropriate destinations according to predefined rules.

Specifically, the system built uses EventBridge Scheduler, a feature that allows events to be scheduled at regular intervals or at specific times. This service is useful for performing recurring tasks or planned actions, such as starting jobs on AWS Lambda or other resources. In practice, the EventBridge Scheduler functions as a serverless 'cron', allowing time-based workflows to be automated without having to manage a dedicated infrastructure.

2.3 Serverless Framework

The Serverless Framework is an open-source tool that simplifies the deployment and management of AWS Lambdas by centralizing resource definitions in a single YAML configuration file. It automates the deployment of AWS Lambda functions together with other cloud resources, such as API Gateway endpoints and DynamoDB tables. This approach reduces the complexity involved in setting up permissions, configuring event triggers, and scaling applications automatically. The framework's command-line interface enables rapid deployment, monitoring, and version management, simplifying the development cycle and allowing developers to focus on application logic rather than infrastructure. In this project, the Serverless Framework efficiently handles the deployment of Lambda functions minimizing the need for manual configuration.

2.4 Table formats

2.4.1 Apache Parquet

Apache Parquet[3] is an open-source, column-oriented data file format designed for efficient data storage and retrieval. It provides high-performance compression and encoding schemes to manage complex datasets in bulk and is supported by many programming languages and analytics tools. Initially used exclusively in the Hadoop ecosystem, Parquet is now employed by platforms such as Apache Spark[4] and various cloud services to meet the demands of data warehousing.

Parquet's main features include:

- **Columnar storage**: Unlike row-based formats like CSV or Avro, Parquet stores the values of each column next to one another, allowing for better compression and faster querying, especially when accessing only a subset of columns.
- **Self-describing**: Each Parquet file contains metadata, such as the schema and structure, facilitating interoperability between services that write, store, and read Parquet files.
- **Efficient compression**: By leveraging the fact that columnar data tends to be of the same type, Parquet achieves more effective compression than row-based formats. This reduces storage needs and accelerates data transfers.
- **Flexible encoding schemes**: Parquet supports various encoding schemes, such as Run-Length Encoding (RLE), Dictionary Encoding, and Delta Encoding, which further optimize compression and performance.
- **Schema evolution**: Parquet allows schemas to evolve over time by adding or removing fields without affecting existing data, making it ideal for dynamic environments.

[3] https://parquet.apache.org/
[4] https://spark.apache.org/

Row Format vs. Columnar Format

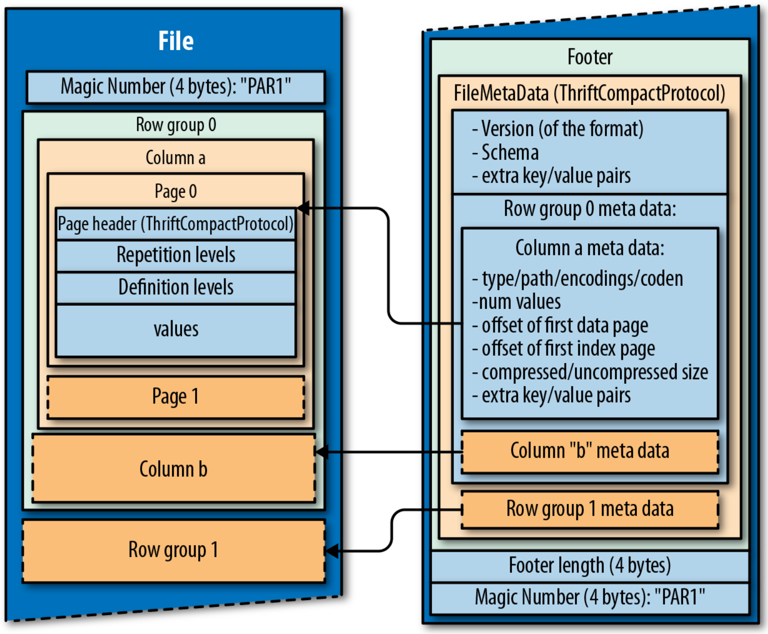

Figure 2.1: Parquet File Layout

As you can see in Figure 2.1, each Parquet file contains a header, one or more data blocks, and a footer. The data itself is stored in the data blocks, while the footer holds metadata about row groups, columns, the Parquet format version, and a 4-byte magic number. The data is organized into:

1. **Row groups**: These logically partition the dataset into rows. Each row group contains a column chunk for every column in the dataset. Row group size can be pre-configured: larger groups improve sequential Input/Output (I/O) but need more buffer memory. A recommended size is between 512 MB and 1 GB.

2. **Column chunks**: Each column chunk represents contiguous data for a specific column within a row group.

3. **Pages**: A page is the smallest indivisible unit in Parquet, used for compression and encoding. The division into pages allows for more efficient compression and parallelized reads. Page size can be pre-configured: smaller data pages allow more precise reads, like single-row lookups, while larger pages reduce space and parsing overhead. The recommended page size is 8KB.

This structure is optimized for analytical queries that only require a subset of columns, reducing I/O by reading less data. Moreover, Parquet files contain statistics like the minimum and maximum values for each column, enabling engines to skip irrelevant data blocks during queries.

Compression Techniques

In Parquet, compression is performed at the column level, supporting various encoding methods, including:

- **Plain encoding**: The default encoding, used when no more efficient method is applicable.
- **Dictionary encoding**: Frequently occurring values are stored in a dictionary, and the data is replaced with the corresponding keys, reducing storage size. This method is applied dynamically when advantageous.
- **Run-Length Encoding (RLE)**: When consecutive values are the same, they are stored as a single value along with their count. Parquet combines bit-packing with RLE to achieve better compression.
- **Delta Encoding**: Instead of storing raw values, the difference between consecutive values is stored, which is useful for sequential data.

Parquet also supports compression algorithms such as Gzip, Snappy, Brotli, and LZO.

Performance Benchmarking

A benchmarking study conducted by Cloudera compares Parquet, Avro[5], and CSV across various tasks [8]. The results are summarized in Table 2.2.

From the table, it is evident that Parquet consistently outperforms Avro and CSV in terms of storage space and query speed. For instance, Parquet files required only 750 MB of disk space for the narrow dataset, compared to Avro's 1 GB and CSV's 4 GB. Similarly, Parquet demonstrated a 97.5% compression ratio on the wide dataset, allowing it to process 3.5 times less data in the same operations, compared to Avro. While Parquet performed well across the board, its efficiency is particularly notable for read-heavy operations, like analytical queries involving group-by and column scans.

Another study on the choice of the most efficient format in the Apache Hadoop system between parquet, orc, avro, CSV, JSON [5] shows that Parquet

[5] `https://avro.apache.org/`

Metric	**Parquet**	**Avro**	**CSV**
Narrow Dataset (3 columns, 83.8M rows)			
Dataset to file format (s)	74	72	45.33
Row count (s)	5.33	5.33	45.33
Group by (s)	9	24	N/A
All data pass (s)	8.33	15.66	N/A
Disk space (MB)	750	1,000	4,000
Wide Dataset (103 columns, 694M rows)			
Dataset to file format (s)	138	180	68
Row count (s)	2.66	45.33	68
Group by (s)	31	54	N/A
All data pass (s)	33.33	140	N/A
Disk space (GB)	5	18	200

Table 2.2: Performance Comparison of Parquet, Avro, and CSV

is better than its competitors from the point of view of storage, all data search, unique row search and sorting. However, it has a similar performance to ORC in the grouping and filtering tasks.

However, Parquet can be slower to write than row-based formats like CSV or Avro, due to the overhead of generating metadata.

Use Cases and Limitations

Parquet is particularly well-suited for scenarios that require efficient compression and fast query performance on large datasets. It is often used for analytical queries that need to access a subset of columns, in data pipelines where schema evolution is crucial, and in cases where efficient data storage and retrieval are key considerations. However, one limitation of Parquet is the potential performance drop when querying full rows in datasets with a very wide structure, typically with more than 100 columns. In such cases, row-based formats like Avro or ORC may offer better performance, as they are optimized for retrieving entire rows.

2.4.2 Apache Iceberg

Apache Iceberg[6] is an open-source format for managing large tables in Data Lakes. Initially developed by Netflix and later open-sourced, Iceberg was designed to overcome the limitations of more traditional table formats such as Apache Hive, which do not scale well in complex and distributed Data Lakes. Iceberg natively supports table management on both Hadoop Distributed File System (HDFS) and cloud storage, such as Amazon S3, Google Cloud Storage, or Microsoft Azure.

The primary goal of Iceberg is to provide a table format that efficiently supports the management of distributed data, allowing for reliable large-scale read and write operations, ensuring data consistency, and facilitating operations such as dataset scanning and version control.

Like Apache Parquet described in the previous section 2.4.1, Iceberg is based on a layered architecture that separates metadata management from the data itself. This separate approach allows for more effective management of read and write operations and enables scalable modification operations. Metadata in Iceberg is crucial for table management. Iceberg uses a snapshot system to keep track of table versions, enabling rollback operations and ensuring full control of changes. The metadata includes information on schema, partitioning, data files, and delete files. Each data modification generates a new snapshot that can be consulted to view the table's evolution over time.

Since snapshots store details about the snapshot's timestamp, partition, and relevant data files, Iceberg supports versioning. Therefore, a snapshot provides a view of the entire dataset at a specific point in time.

Iceberg Features

Iceberg tables offer several benefits compared to other formats traditionally used in the data lake, including:

- **Schema evolution**: Supports commands to add, drop, update, or rename columns without causing side effects or inconsistency.
- **Partitioning**: Iceberg introduces dynamic and flexible partitioning. Traditionally, Data Lakes use file path-based partitioning, which can be inef-

[6] `https://iceberg.apache.org/`

ficient and difficult to manage. Instead, Iceberg implements a metadata-based partitioning strategy that reduces overhead and improves query performance. Two key improvements in partitioning are:

  - *Partition evolution*: Facilitates the modification of partition layouts in a table, such as data volume or query pattern changes, without needing to rewrite the entire table.
  - *Hidden partitioning*: Iceberg automatically handles partitioning by transforming column values (e.g., converting `event_time` into `event_date`) without requiring user-maintained partition columns. This allows queries to benefit from partitioning transparently, hiding the physical layout from producers and consumers. The separation of physical and logical partitioning enables the flexible evolution of partition schemes over time, improving performance without costly migrations.

- **Time travel**: Allows users to query any previous version of the table to examine and compare data or reproduce results using past queries.
- **Version rollback**: Quickly fixes discovered issues by resetting tables to a known good state.
- **Increased performance**: Ensures data files are intelligently filtered for faster processing through advanced partition pruning and column-level statistics.
- **Transactional consistency**: Helps users avoid partial or uncommitted changes by tracking atomic transactions with ACID (Atomicity, Consistency, Isolation, Durability) properties.
- **Table optimization**: Optimizes query performance to maximize the speed and efficiency with which data is retrieved. The main optimization is file compaction. This is particularly useful in Data Lakes, where data is often written in small files, causing fragmentation that can negatively impact query performance. Periodic compaction reduces the number of fragmented files and improves efficiency.

ACID Operations

One of the key contributions of Apache Iceberg to the Data Lake world is the implementation of support for ACID operations, a crucial innovation to ensure the consistency and correctness of data operations.

- **Atomicity**: Changes to the data are treated as atomic transactions, meaning multiple operations are applied as a block, ensuring they are either fully completed or not applied at all. This is particularly useful in scenarios requiring concurrent writes.
- **Consistency**: Thanks to advanced metadata management, Iceberg ensures that all operations on a table remain consistent, even in distributed environments.
- **Isolation**: Iceberg provides transaction isolation, ensuring that concurrent operations do not interfere with each other. Using a snapshot system, each read or write operation accesses a consistent version of the table.

These features make Iceberg ideal for managing highly reliable and resilient data pipelines.

Table Layout

The Iceberg table format offers similar capabilities to SQL tables in traditional databases. However, unlike such datasets, Iceberg operates in an open and accessible manner, allowing multiple engines (such as Spark, Dremio[7], etc.) to work on the same dataset.

[7] https://www.dremio.com/

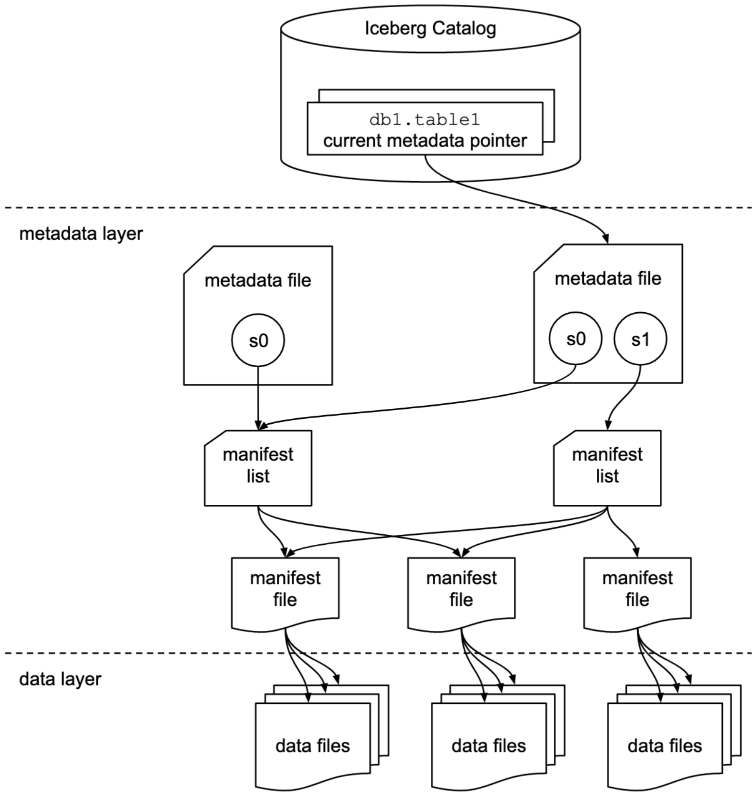

Figure 2.2: Iceberg Table Layout

Through metadata files, Iceberg tracks point-in-time snapshots by maintaining all deltas as a table. Each snapshot provides a full description of the table's schema, partition, and file information. Additionally, Iceberg intelligently organizes snapshot metadata hierarchically, enabling data processing engines to efficiently apply changes without redefining all dataset files, ensuring optimal performance at data lake scale.

The Iceberg table architecture consists of three layers:

- **The Iceberg catalog**: This is where services find the location of the current metadata pointer, which helps identify where to read or write data for a given table. Each table's references or pointers exist here, identifying the current metadata file.

- **The metadata layer**: This layer consists of three components: the metadata file, the manifest list, and the manifest file. The metadata file contains information about a table's schema, partition details, snapshots, and the current snapshot. The manifest list includes a list of manifest files along with information about the manifests that form a snapshot. Manifest files track data files and include other details and statistics about each file.

- **The data layer**: Each manifest file tracks a subset of data files, which contain information about partition membership, record count, and column upper and lower bounds.

3

System Development

3.1 System Architecture Design

The design of the system was a crucial part of my internship, consuming approximately 20% of the total hours dedicated to the project. It was crafted to meet the specific requirements of the business while also taking into account the current context and future needs of the company. UNOX, having collaborated with AWS since 2019, leverages innovative AWS technologies to maintain a competitive edge. For this project, we were guided by an AWS Solution Architect and an Enterprise Account Manager, who played an essential role in helping us build a cutting-edge system that adheres to AWS's Well-Architected Framework principles [3], ensuring Operational Excellence, Reliability, Performance Efficiency, Security, Cost Optimization, and Sustainability.

The AWS Solution Architect, in particular, provided detailed comparisons of the various AWS technologies available, enabling key decision-makers such as the team leader, the company's Chief Information Officer (CIO), and myself to gain a comprehensive understanding of the potential architectures we could build. During the early stages of my internship, we held recurring meetings to explore potential solutions that best aligned with our specific requirements and the nature of UNOX's business operations. These discussions helped us identify the ideal path to follow, although the initial solutions inevitably evolved as we encountered and faced practical challenges during the system's development.

A core aspect of the system design was to ensure a clear separation between the storage layer and the business analytics layer, effectively decoupling data producers (such as operational systems) from data consumers (like reporting and predictive analytics systems). This separation was essential to facilitate data science activities, where a data lake provides a convenient storage layer for experimental data, supporting both the input and output of data analysis tasks. The architecture also needed to support autonomous creation and use of data, without the need for coordination between programs or analysts, while at the same time enabling the sharing and reuse of massive datasets through a distributed computational framework.

3.1.1 Data Lake vs. Data Warehouse vs. Data Lakehouse

The initial idea for this project was to implement a data lake, a more innovative and flexible approach compared to traditional data warehouses. While both are used for storing large volumes of data, they serve different purposes and have distinct architectural characteristics.

Traditionally, a **data warehouse** is optimized for structured data, meaning data is cleaned, organized, and stored in a predefined schema, making it ideal for business reporting and analytics. However, data warehouses typically involve high setup and maintenance costs, and they require significant preprocessing to ensure data consistency before it can be used for analysis. Hence, they suffer from limited flexibility for advanced analytics, including machine learning tasks [11].

A **data lake**, on the other hand, offers a more flexible storage solution. It is capable of storing vast amounts of both structured and unstructured data in its raw form, allowing for greater adaptability. This means that data lakes are not bound by rigid schemas and can accommodate data from diverse sources without the need for heavy preprocessing. Data lakes are particularly suitable for data science, machine learning, and exploratory analysis, as they allow analysts and data scientists to directly interact with raw data, creating an environment where experimentation can thrive.

It enables autonomous data handling, and data producers do not need to coordinate directly with consumers. It also provides a shared storage framework, which facilitates collaboration between teams and allows for the re-use of large datasets without duplication or complex integration. A detailed analysis of the differences between data warehouses and data lakes is given in Table 3.1 [12].

Parameters	**Data Warehouse**	**Data Lake**
Data	Focuses only on business processes	Stores everything
Processing	Highly processed data	Mainly unprocessed data
Type of Data	Mostly in tabular form and structured	Can be unstructured, semi-structured, or structured
Task	Optimized for data retrieval	Share data stewardship
Agility	Less agile, has a fixed configuration	Highly agile, can be configured and reconfigured as needed
Users	Widely used by business professionals and business analysts	Used by data scientists, data developers, and business analysts
Storage	Expensive storage for fast response times	Designed for low-cost storage
Security	Allows better control of the data	Offers less control
Schema	Schema on writing (predefined schemas)	Schema on reading (no predefined schemas)
Data Processing	Time-consuming to introduce new content	Helps with fast ingestion of new data
Data Granularity	Data at the summary or aggregated level of detail	Data at a low level of detail or granularity
Tools	Mostly commercial tools	Can use open-source tools such as Hadoop or MapReduce

Table 3.1: Comparison between Data Warehouse and Data Lake

However, while data lakes excel in flexibility, they can sometimes suffer from challenges related to data governance, data quality, performance, and metadata management. As a result, organizations have adopted a two-tier architecture: storing data in lakes and then moving curated data to warehouses for structured analytics. The two-tier model (data lakes + data warehouses) introduces new complexities, including reliability and cost issues. Indeed, maintaining consistency between the lake and warehouse is complex, storing data in two places and running ETL processes increases costs.

This is where a more modern architecture, the **Data Lakehouse** [1], promoted by Databricks[1], a cloud platform built on Apache Spark that enables unified data analytics, machine learning, and big data processing, comes into play.

The lakehouse architecture combines the best features of both data lakes and data warehouses. It retains the ability of a data lake to store raw and

[1]`https://www.databricks.com/`

semi-structured data while incorporating some of the data management and performance optimization features of a data warehouse. This hybrid approach allows for real-time analytics and ACID transactions on large datasets by adding structured layers of metadata to the raw data. Actually, although we call it a data lake for simplicity, the system developed in this project can be interpreted as a data lakehouse. This is because we have integrated several layers that provide pre-aggregated tables in Parquet or Iceberg formats, which are directly usable for advanced analysis. These formats not only offer significant performance benefits through better compression and faster query times but also enable ACID operations. This structured approach allows us to maximize the system's potential for advanced business intelligence while maintaining the flexibility and scalability inherent in a data lake.

3.1.2 System architecture overview

The architecture of the entire system, as illustrated in Figure 3.1, was designed to meet the company's specific requirements, leveraging cloud technologies to handle large volumes of data while ensuring performance, reliability, and cost-efficiency. This event-driven design follows a predefined pipeline, where each event triggers the execution of a specific Amazon service with tailored parameters. The entire flow is designed to run at a defined frequency, ensuring up-to-date data availability for analysis. The solution automates the process of ingesting, transforming, and analyzing data from various sources, providing a centralized platform for data storage and business intelligence.

The system starts with data being collected from two main sources: PostgreSQL for operational data and MongoDB for IoT data generated by the ovens. Each of these data sources follows a custom extraction process. For PostgreSQL, AWS Glue is used to execute ETL jobs that extract the data, convert it into optimized formats, and load it into Amazon S3. For MongoDB, the system uses AWS Lambda to manage event-driven data extraction, process the data and store it in S3 in a scalable and efficient way.

Once the data is in S3, it is organized into three layers:

- **Raw**: Where the data is stored as extracted, without any transformation.
- **Curated**: Where the data is cleaned, formatted, and partitioned for better query performance.

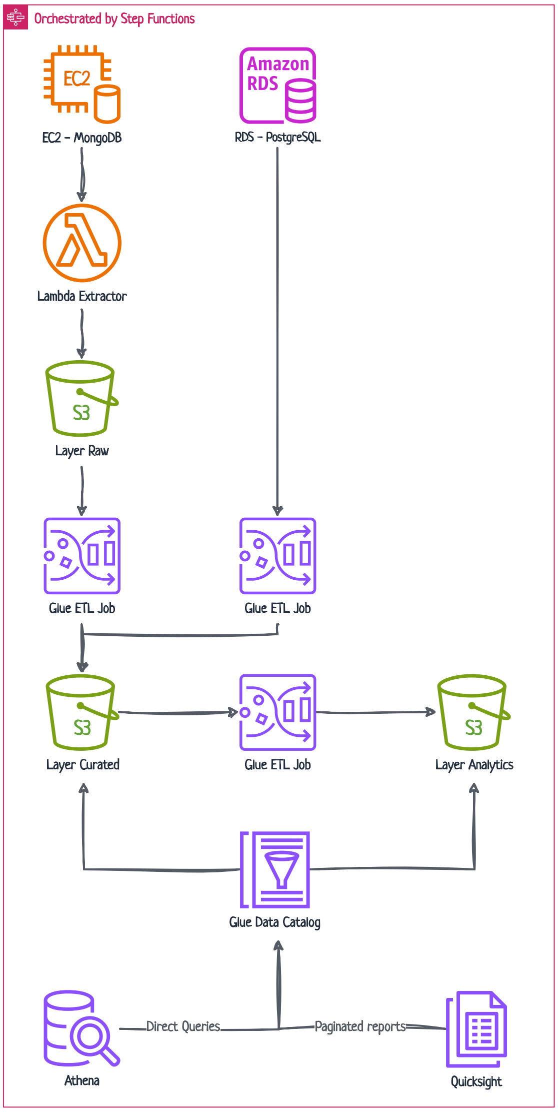

Figure 3.1: The System Architecture

- **Analytics**: Where the data is pre-aggregated and optimized for specific use cases, such as business reports.

The system relies on AWS Glue Data Catalog to manage metadata, enabling easy access and query capabilities. For queries, AWS Athena is used to allow SQL queries directly on the data stored in S3, while AWS QuickSight provides interactive dashboards and visualizations for business users to explore and analyze the up-to-date data.

To enhance security, each AWS service involved in the event-driven pipeline has tailored IAM policies. These policies restrict access so that each service can only interact with the specific resources it requires, minimizing potential exposure to other parts of the Amazon ecosystem. The system utilizes stringent access rules to reinforce security by ensuring only the minimum necessary permissions are granted for each AWS event, helping protect sensitive data and maintaining a secure environment.

In the following sections, a detailed description of the entire workflow will be provided, including how data ingestion, integration, and cataloguing are performed, as well as how queries and reports are generated. This chapter will also cover how orchestration and scheduling are managed through AWS Step Functions, ensuring that each component of the system works seamlessly and in the correct sequence.

3.2 Data Sources

Before going into the implementation details of the solution, it is necessary to analyse the two data sources in order to better understand certain choices made during the design phase.

3.2.1 PostgreSQL Data

The Postgres database in this system contains crucial operational data, including various datasets related to Unox ovens and their usage. It is managed by an AWS RDS instance (described in section 2.2.3) that uses a `db.t3.xlarge` configuration, providing 4 vCPUs, 16 GB Random Access Memory (RAM), and 100 GiB of gp2 storage. The database runs PostgreSQL version 12.19, and backups are automatically created every 14 days to ensure data safety.

PostgreSQL hosts several databases, among which the most heavily used is the `ddc` database. The name `ddc` refers to the Data Driven Cooking (DDC) platform, an intelligent cooking system, developed by the Digital Experience team, that leverages data to optimize and enhance cooking processes. DDC offers advanced features for oven owners, enabling them to efficiently monitor and control their devices.

The `ddc` database comprises 72 tables, with the most critical ones being:

- **Device**: Contains detailed information about all network-connected devices produced by Unox.

- **Company**: Stores information regarding the companies that own Unox ovens.
- **Device group**: Facilitates grouping of devices within a company, allowing management differentiation based on factors such as location, model, or other criteria.
- **Device recipe**: Tracks the current recipe loaded on a device.

The relational schema of the main tables is shown in the figure 3.2. In this diagram, diamonds represent relationship tables, while arrows indicate foreign key relationships pointing to the primary key of the referenced table.

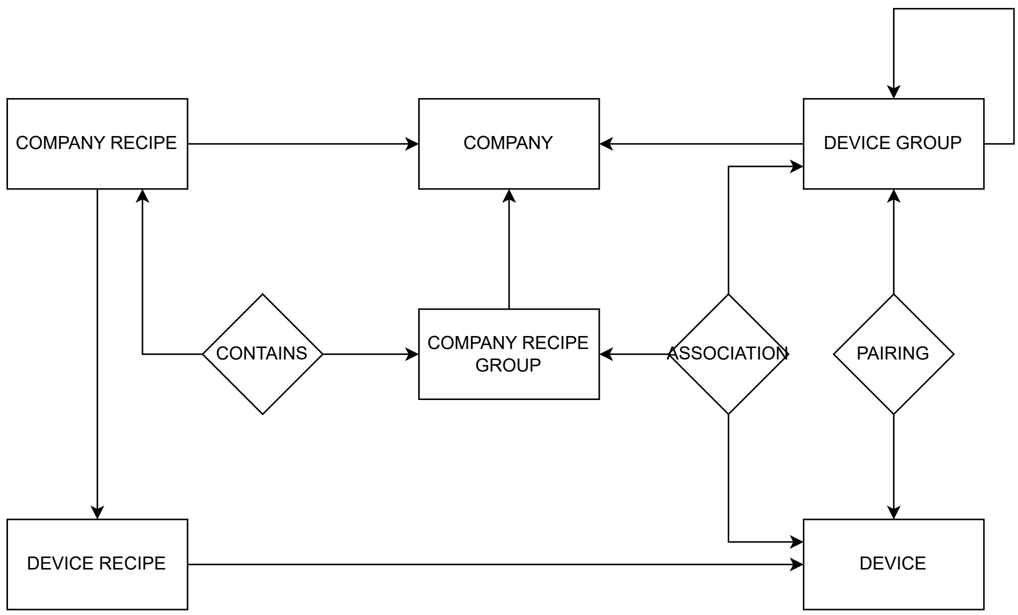

Figure 3.2: Main tables relational schema

As seen in the diagram, recipes related to a company can also be differentiated based on the device group. This means that recipes can apply at different levels: specific to a device, to a group of devices, or to an entire company. Furthermore, recipes can be created by the community or by Unox itself, stored in the `community_recipe` and `chefunox_recipe` tables, respectively.

In addition to these, there are other tables related to the primary ones, such as those containing data on the parameters, profiles, or settings of a company or device.

Two more tables, `device_recipe_history` and `device_ip_info_history`, contain historical data. The former records all recipes created since the installation of a device, while the latter stores the history of IP addresses and associated information for a device. These are the largest tables in terms of storage, with `device_recipe_history` occupying 22 GB and `device_ip_info_history` storing 5 GB of data.

3.2.2 MongoDB Data

Since AWS RDS does not support the MongoDB engine, this database is installed on an AWS EC2 machine. The EC2 instance used is a `c5.4xlarge`, with 16 vCPUs and 32 GiB of RAM. The C5 instances are optimized for compute-intensive workloads and offer high performance at a low cost, providing an optimal balance between price and computational power.

Unfortunately, the MongoDB database is deployed on a single replica set and is not configured as a *sharded* cluster. A replica set consists of a group of MongoDB instances that maintain the same dataset, providing redundancy and high availability. Furthermore, without a sharded cluster, the system cannot horizontally scale across multiple nodes, which limits its ability to handle large-scale datasets and high read/write throughput efficiently.

MongoDB organizes data into collections and documents, which can be thought of as equivalent to tables and rows in a relational database, respectively. A document in MongoDB is a flexible, schema-less structure that can be represented as a JSON-like object, where there are no constraints on the data types or mandatory fields.

The database contains 20 collections, of which 13 have been deemed useful for analysis and inclusion in the data lake. Table 3.2 describes all the collections, including the approximate number of documents and the storage space they occupy.

All the collections store time series data, which means that they record sequences of data points indexed in time order. Consequently, each collection includes an `idDevice` field, indicating which device produced the specific document, and a timestamp field that records when the data was generated. In addition to these, each collection contains specific fields that characterize its data.

As we specified earlier, the data belonging to these collections are generated

Collection	Total # of Documents	Size	Storage Size
alarm	43 M	3.7 GiB	1.9 GiB
end_of_prog	240 M	60 GiB	24.7 GiB
end_of_prog_aggregated	29 M	16.2 GiB	15.8 GiB
events	1500 M	152.6 GiB	78.2 GiB
evereo_sess	215 K	48.4 MiB	20 MiB
request	820 K	60.7 GiB	38.8 GiB
sd_events	75 M	11.2 GiB	5.2 GiB
sd_haccp	220 M	90.3 GiB	23.6 GiB
sd_messages	220 M	130 GiB	25.3 GiB
sd_variables	190 M	505 GiB	156.2 GiB
sdata	170 K	80.6 MiB	29 MiB
variable_logs	970 M	7.3 TiB	1.8 TiB
working_minutes_logs	180 K	29.3 MiB	8.6 MiB

Table 3.2: MongoDB collections used for analysis, with document counts and storage sizes.

by the various sensors installed in the devices. When the device is connected to the internet and ready for transmission, the collected data is sent to the back-end, which is responsible for uploading it to the database. The back-end implements a retention mechanism, ensuring reliability in case of database unavailability or upload issues. If a problem occurs, the device is alerted that the data upload was unsuccessful, meaning the device must reattempt the upload once it is ready to transmit again. This mechanism provides robustness, preventing data gaps and duplicates across MongoDB collections. Each device, therefore, uploads data independently, with unpredictable upload intervals.

After successfully uploading the documents to MongoDB, the system updates the `last_log_date` timestamp in the `device` table of the DDC database. This field reflects the timestamp of the most recent document uploaded for the respective device. Observing this `last_log_date` column, we can confirm that all data with earlier timestamps has been successfully uploaded. However, there may still be additional data recorded by the device that has not yet been uploaded to MongoDB.

It is worth noting that the `variable_logs` collection is the largest in terms of data size and will be described in detail in the following section.

Variable Logs

The `variable_logs` collection represents a time series that captures telemetry data from the ovens. Every 30 seconds, each oven sends data for every available sensor. Each sensor is represented by a variable, and there are sensors that measure various parameters, such as temperature, humidity, fan speed, microwave activity, and other values related to the engine or power supply. For each of these groups, multiple sensors may exist. For example, temperature can be measured in several locations, including the cooking chamber, the core probe, the control board, or the power board. There may also be temperature values such as the one set by the user or the one recommended by algorithms. Additionally, measurements may be sent by accessories connected to the oven.

Each oven can send up to 30-35 measurements every 30 seconds to the backend, which handles the writing to the database. However, only some of these measurements are actually sent, depending on the oven family, model, and the type of program that is running.

Unfortunately, the document structure in this collection is not very intuitive. Each document represents a 6-hour slot of measurements for a single sensor, divided into 6 one-hour samples, with each sample containing 120 measurements (one every 30 seconds). This format is optimized for the oven's internal algorithms and is maintained for backward compatibility.

From an analytical perspective, this pattern is somewhat inconvenient. When querying data for specific time periods, a single query returns 720 measurements, and to locate a precise measurement, one must navigate through the nested structure of the JSON document.

To address this issue, a view of the collection was previously created, called `variable_logs_clean`, using an aggregation pipeline composed of 7 stages. This view transforms the data structure into a more accessible format by unpacking each document so that there is a single document for each individual measurement, associating it with the specific timestamp of that measurement. Non-existent measurements are excluded from the view, thus preventing the creation of unnecessary documents. This solution results in a significant increase in the number of documents, but each document is much lighter in terms of storage.

For convenience, in the data lake, the view `variable_logs_clean` will be used as the source for oven telemetry data instead of the original collection.

3.3 Data Ingestion

As outlined in the introduction, the data ingestion process differs depending on the two main data sources used in the system: PostgreSQL and MongoDB. These differences arise from the unique requirements of each database and the technologies used to connect and extract data.

For PostgreSQL, AWS Glue establishes a connection using Java Database Connectivity (JDBC). JDBC is a standard API that facilitates communication between a client and a relational database, enabling Glue to execute queries, extract data, and transform it for storage in a structured format like Parquet [15]. This process is crucial for handling structured data efficiently.

In contrast, for MongoDB, the ingestion process involves a direct connection using a MongoDB-specific driver. This type of connection allows the system to interact with MongoDB. Unlike JDBC, the driver is tailored for the unique characteristics of MongoDB, allowing for the handling of unstructured or semi-structured data.

Due to the need for multiple sequential jobs to export data effectively, it is essential to be able to select specific data subsets for each job. In PostgreSQL, AWS Glue jobs handle this through incremental data ingestion allowed by Bookmarks, while for MongoDB, a different approach was required. To achieve scalability and flexibility in handling the unstructured data from MongoDB, AWS Lambda was chosen as the ingestion tool.

3.3.1 PostgreSQL Data

Connection to the database

Before building the actual extraction job for PostgreSQL, it was essential to configure the connection to the database. AWS Glue facilitates the management of connections through a dedicated section where various parameters can be defined. However, prior to configuring this, it was necessary to analyze the company's networking architecture on AWS.

In AWS, networking is managed through a VPC, which is a virtual network dedicated to an AWS account. A VPC can contain multiple subnets, which are smaller segments of the VPC used to organize and isolate resources within different parts of the network. Subnets can be public or private, depending on whether they are associated with an Internet Gateway (IGW) that allows communication with the Internet. Security Groups act as virtual firewalls, controlling inbound and outbound traffic for AWS resources. Routing Tables are used to manage the paths that data packets take to reach various destinations, such as other AWS resources or external networks via the Internet Gateway.

In this specific setup, the PostgreSQL RDS instance is distributed across three subnets, all of which belong to the same VPC. These subnets share a single routing table that contains an Internet Gateway, allowing communication with external networks, including the Internet.

To configure the connection to the RDS instance, several parameters had to be specified:

- **Database credentials**: This includes the connection type, host, username, password, and port number.
- **Networking details**: The VPC, subnet, and security group had to be defined to ensure that Glue could securely connect to the RDS instance.

Once these were set up, it was necessary to create a VPC Endpoint in the routing table. A VPC endpoint enables a private connection between AWS services (in this case, RDS and Glue) without the need to traverse the public internet. In this context, the VPC Endpoint was crucial for allowing Glue to access the RDS instance directly and securely within the VPC. This reduces latency and enhances security by keeping traffic within AWS's private network.

ETL Job description

After setting up the connection to the PostgreSQL database, the next step was to create the actual ETL (Extract, Transform, Load) job for data extraction. AWS Glue provides a powerful environment for automating ETL workflows, and each Glue job operates as a script written in either Python or Scala, leveraging Apache Spark as the underlying engine for distributed data processing.

The script can be generated visually or written manually, and it is executed in a serverless environment, meaning there is no need to manage the infrastructure, as Glue handles the allocation of resources dynamically. The ETL script operates using a GlueContext, a specialized context that integrates AWS Glue-specific features into Spark's ecosystem. GlueContext provides the necessary methods to read from and write to various data sources, like databases, S3 buckets, and data catalogs.

Data in Glue is stored in a structure known as a DynamicFrame [6], which is similar to Spark's DataFrame but with added flexibility for semi-structured or schema-less data. A key difference between a DynamicFrame and a DataFrame lies in the level of schema enforcement. While a DataFrame in Spark is strictly tied to a predefined schema, a DynamicFrame is schema-aware but more flexible, allowing it to adapt to evolving data structures. DynamicFrames are especially useful in ETL processes that involve reading from semi-structured sources where data may not adhere to a rigid schema.

As outlined in section 2.2.5, AWS Glue simplifies the creation of ETL jobs through a fully visual interface. Glue jobs are structured as workflows, which consist of multiple steps that can be executed either sequentially or in parallel. The visual interface allows users to design a flowchart, where each node represents a specific step in the job. These nodes are categorized as Source, Transformation, or Target, depending on the role they play in the ETL process.

For example, in the simplest case of moving a dataset from one location to another, only two nodes are required: a Source node to define the data source and a Target node to specify the destination. Additional parameters such as bookmark usage (to track job progress), execution type, timeout settings, and the database connection are configured globally at the job level. AWS Glue automatically transforms the visual ETL workflow into a Python or Scala script. Therefore, each time the job is executed, it runs the script that was generated based on the visual design. Each node in the flowchart corresponds to a function call, either from Apache Spark or from AWS Glue's specific libraries.

While visual ETL tools are convenient and intuitive for building basic workflows, they have limitations. They do not always expose all the parameters or advanced options available in the GlueContext or SparkContext. This is particularly restrictive when more granular control is needed, such as fine-tuning the performance of data extraction or transformation steps. For this reason, a custom Python script was written for the PostgreSQL extraction. This approach allowed for full utilization of Glue's and Spark's capabilities, ensuring optimal performance and flexibility. Using a custom script also allowed for more advanced data filtering, error handling, and optimization techniques.

Glue Bookmarking in JDBC Sources

In the context of AWS Glue, *bookmarks* serve as a mechanism to track the progress of a job by saving the last processed primary key. Specifically, for JDBC sources, the bookmark stores the value of the last primary key that was successfully processed during the job execution. This information is stored in an internal JSON file in Glue's storage system. When the job is run again, it checks whether a bookmark is present for each table. If a bookmark exists, the job filters the data by selecting only those rows where the primary key is greater than the one saved in the bookmark, ensuring that only new rows are imported.

However, this functionality presents certain limitations. If a row in the source table is updated, but its primary key remains unchanged, the updated row will not be selected during the next job execution, leading to outdated data in the data lake. On the other hand, if the primary key changes during the update, the updated row is imported into the data lake, but the old version of the row remains, resulting in duplicated data. Consequently, the use of bookmarks is only advantageous in cases where the source table is append-only, meaning no updates or deletions are performed.

Given these constraints, specific solutions have been proposed depending on the behavior of the tables in the company's `ddc` database:

- For tables that only undergo append operations, the use of bookmarks is feasible, allowing the job to load only the newly inserted rows. This helps optimize the data ingestion process.
- For tables that undergo upserts (i.e., updates or inserts), a more efficient solution is to replace the entire table during each job execution. This involves deleting the previous version of the table in the data lake and replacing it with the updated version from the source. Since the total memory occupied by these tables is less than 300 MB, this approach is manageable from a performance and cost perspective.
- A special case is the `device_recipe` table, which deletes a row and reinserts it with a new primary key whenever an update occurs. Given the size of this table (approximately 1 GB), the most effective solution is to replace the entire table in the data lake after each job execution to ensure data consistency.
- The `device` table also behaves similarly to `device_recipe`, but with an additional requirement for *Time Travel*. Time Travel is a feature that would be useful in this context to retain historical data, such as older firmware versions installed on the devices or previous IP addresses. In this case, the

use of Apache Iceberg is proposed as a solution. Iceberg is a table format designed for large-scale datasets, providing capabilities such as schema evolution, partitioning, and Time Travel. During each job execution, the entire `device` table (around 70 MB) would be loaded, and, if the table is already present in the data lake, a `MERGE INTO` operation would be performed.

The `MERGE INTO` operation in Iceberg works by combining data from two tables based on a matching condition. Specifically, it checks for rows that already exist in the target table and updates them with the new data from the source. If a row in the source table does not have a matching row in the target table, it is inserted as a new row. This process ensures that both the updated and new rows are correctly handled without creating duplicates.

While performing a `MERGE` between two nearly identical tables (with only a few updated rows) is not the most efficient operation, a proposed optimization is to load only the rows with a recent `updated_at` timestamp. This column, which indicates the time at which a row was last updated, can be used to filter rows that have been updated within a certain time frame. For example, a query like `SELECT * FROM device WHERE updated_at >= NOW() - INTERVAL '2 DAYS'` would select only the rows that have been updated in the last two days, reducing the amount of data that needs to be processed during the `MERGE INTO` operation.

However, due to inconsistencies in the `updated_at` column, it was ultimately decided to perform the `MERGE` with the entire table as the source, ensuring that no updates are missed.

Before diving into the specifics of the implemented script, it is worth mentioning that the entire solution could have been replaced by a Change Data Capture (CDC) system [10], for example using AWS Database Migration Service (DMS). AWS DMS allows for continuous replication of data changes from a source database to a target location, capturing inserts, updates, and deletes in real-time. By leveraging CDC, it would be possible to automatically detect and replicate any change in the PostgreSQL database to the data lake, eliminating the need for periodic full-table exports. However, this solution was discarded due to the high costs associated with its constant execution, as DMS would require a continuous running process to capture all changes.

Glue Script

The exporting code is divided into three distinct phases, each processing a subset of tables sequentially based on their characteristics and behavior. Below is a detailed explanation of each part of the script.

First Phase: Processing Upsert Tables The first part of the script processes tables that can undergo upsert operations (update or insert). The list `tableNames` contains the name of tables to be processed, which was generated by querying the `information_schema.tables`, a system table that holds metadata about all tables in the database. From this set, append-only tables were excluded.

The following code snippet illustrates how each table is processed:

```python
for tableName in tableNames:
    logger.info("TABLE: " + str(tableName))
    table = glueContext.create_dynamic_frame.from_options(
        connection_type = "postgresql",
        connection_options = {
            "useConnectionProperties": "true",
            "dbtable": tableName,
            "connectionName": "Postgresql connection production",
        }
    )

    objects_to_delete = s3.list_objects_v2(Bucket="datalake-postgres"
, Prefix=tableName+"/")
    if 'Contents' in objects_to_delete:
        delete_keys = {'Objects': [{'Key': obj['Key']} for obj in
objects_to_delete['Contents']]}
        s3.delete_objects(Bucket="datalake-postgres", Delete=
delete_keys)

    out = glueContext.getSink(path="s3://datalake-postgres/"+
tableName+"/", connection_type="s3", updateBehavior="
UPDATE_IN_DATABASE", partitionKeys=[], enableUpdateCatalog=True,
transformation_ctx="write_"+tableName)
    out.setCatalogInfo(catalogDatabase="postgres", catalogTableName=
tableName)
    out.setFormat("glueparquet", compression="snappy")
    out.writeFrame(table)
```

Code 3.1: First phase Postgres extraction

The `glueContext.create_dynamic_frame.from_options()` function reads the PostgreSQL table and converts it into a `DynamicFrame`, which can then be manipulated and written to other destinations. Since these tables undergo upserts, it is necessary to delete the existing tables in Amazon S3 to replace them with the updated version. This is done using the `list_objects_v2()` and `delete_objects()` methods from the `boto3` client. `boto3` is the AWS Software Development Kit (SDK) for Python, which allows to interact with AWS services, such as S3.

After clearing the existing data, the updated `DynamicFrame` is written back to S3 in Parquet format using `glueContext.writeFrame()`. The data is automatically added to the AWS Glue Data Catalog, making it available for further querying and processing. The use of the `Snappy` compression format ensures efficient storage.

Second Phase: Processing Append-Only Tables The second phase processes tables that are append-only, meaning that new rows are only added, and no updates or deletions occur. This phase is similar to the first one, but with a few key differences. First, in this case, the use of bookmarks is enabled, allowing the job to only load new rows since the last execution. This is achieved by configuring the `transformation_ctx` parameter, which ensures the bookmark functionality tracks the last processed row and continues from that point during the next run.

Furthermore, since some of these append-only tables (as discussed in section 3.2.1) contain a large amount of data, it was necessary to optimize the read operations. This was done by using the `hashfield` and `hashpartition` parameters. These parameters enable partitioning of the data based on a hash of the specified field, which allows for parallel processing, improving the performance of reading large tables. The `hashfield` determines the column used for hashing, and `hashpartition` defines how many partitions the data should be split into for parallel execution.

Third Phase: Processing the Device Table with Time Travel The third phase of the script processes the `device` table, which involves upsert operations and requires *Time Travel*. To enable Time Travel, the table is stored in Apache Iceberg format, which supports versioning and allows efficient retrieval of data from different historical snapshots.

Below is the snippet that handles the `MERGE INTO` operation for the `device` table:

```
MERGE INTO glue_catalog.postgres.device t
USING merge_source s
ON t.id = s.id
WHEN MATCHED AND (t.id_firmware <> s.id_firmware OR t.board_serial <>
    s.board_serial OR t.id_board_model <> s.id_board_model OR t.
   last_ip <> s.last_ip OR t.city <> s.city OR t.connection_kind <> s
   .connection_kind OR t.bridge_firm <> s.bridge_firm OR t.cloud_pin
   <> s.cloud_pin OR t.mirror <> s.mirror) THEN
    UPDATE SET *
WHEN NOT MATCHED THEN
    INSERT *
```

Code 3.2: MERGE INTO for Postgres extraction

The `MERGE INTO` operation compares the rows from the source table (designated as `s`) with the target table (designated as `t`). If a matching row is found (based on the `id`), and any of the key fields (such as firmware, board serial, or IP address) has changed, the row in the target table is updated. If no matching row is found, the new row from the source table is inserted into the target. This ensures that the table in S3 remains synchronized with the PostgreSQL source, while also maintaining historical data for Time Travel purposes.

3.3.2 MongoDB Data

For the IoT data stored in MongoDB, a completely different approach was adopted. AWS Glue does not support bookmarks for connections other than JDBC and S3, which ruled out the possibility of using Glue's native bookmarking functionality. Additionally, several challenges made it impractical to create a custom bookmark system in Glue. One such limitation is that the functions provided by the Glue context do not support *pushdown predicates*, which are conditions applied at the data source level to filter the data returned. Without pushdown predicate support, every time a collection is read, it must be fully imported and then filtered afterward. Importing several terabytes of data with each job execution is clearly unfeasible.

In principle, `PySpark` provides a `read()` function that allows passing an aggregation pipeline as a parameter, allowing to drain data. An aggregation pipeline is a MongoDB framework that processes data through multiple stages, each applying specific transformations or filtering conditions to the dataset. While this approach would work for ongoing operational tasks, during the initial ingestion phase, it would still be highly inefficient due to the large data volume. Handling this with multiple aggregation pipelines, each importing a subset of data, would require sequential execution in Glue, as asynchronous operations are not supported in this environment, significantly increasing execution time.

Given these constraints, it was decided to use AWS Lambda functions, which offer scalability based on workload, parallel execution, and support for asynchronous queries. AWS Lambda is a serverless computing service that runs code in response to events, automatically managing the infrastructure required to execute the code. A notable limitation of Lambda is its maximum execution duration of 15 minutes, which needs to be considered when handling large-scale data processing tasks.

To deploy the Lambda functions, the `Serverless Framework` was used. This framework enables simplified management of serverless applications by automating the setup, deployment, and scaling of resources, without needing to manually provision or manage servers. The functions were written in Node.js with TypeScript, chosen for its robust asynchronous code handling capabilities and strong type-checking, which enhances code reliability and maintainability.

Two ORMs were utilized. For PostgreSQL, `Prisma` was selected to handle bookmarking data, as it allows connections to multiple databases in the same environment. Prisma simplifies data access by generating a type-safe client, which improves both the efficiency and reliability of database interactions. For MongoDB, the original MongoDB driver developed for Node.js was used, offering direct and optimized support for MongoDB's functionalities and data handling requirements.

To interact with AWS services, the JavaScript `SDK v3` was utilized. AWS SDK v3 is the latest version of the Software Development Kit provided by Amazon, specifically designed to facilitate interaction with AWS services. It offers modular packages, each dedicated to a specific service, allowing developers to import only the required functionalities. This approach optimizes application performance by reducing the overall bundle size, which is especially beneficial for serverless environments where minimizing execution time and memory usage is crucial.

The architecture for the Lambda functions consists of two TypeScript-based functions: *master* and *worker*. To initiate the data extraction process, the *master*

Lambda function is invoked. This function is responsible for identifying which devices have sent data since the previous job execution and for invoking multiple *worker* Lambdas accordingly. The *worker* Lambdas, then, handles the export of new data for the identified devices, allowing the system to bypass the 15-minute execution timeout limitation by distributing the workload across multiple functions. Each *worker* Lambda imports data from a single collection and processes a subset of devices. Figure 3.3 illustrates the architecture of the Lambda-based solution used for the data ingestion process.

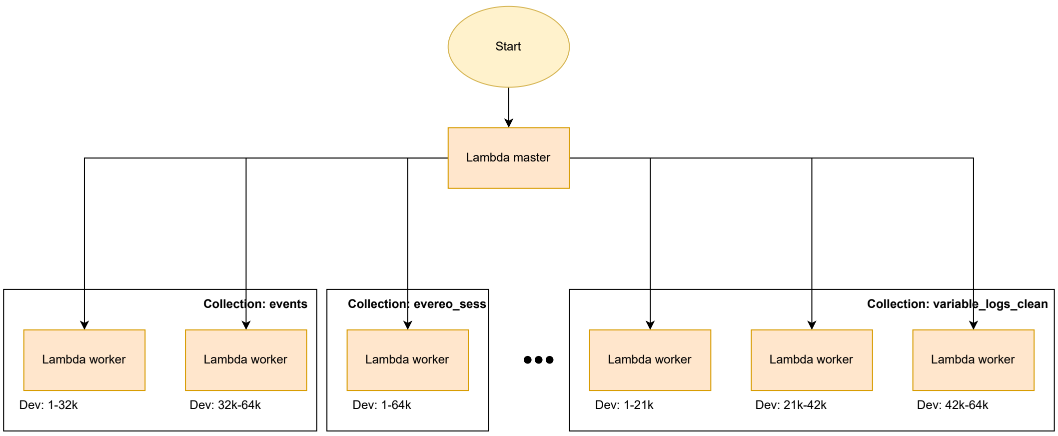

Figure 3.3: Lambdas architecture

To implement custom bookmarking, a dedicated support database was created in an AWS RDS instance to avoid overloading the production database. This support database includes a table named `lastts`, designed to track data ingestion progress. The `lastts` table contains three columns: `idDevice`, which identifies each specific device, `collection`, indicating the dataset collection, and `lastts`, a timestamp indicating the latest ingestion point for the device. Together, `idDevice` and `collection` form the primary key, ensuring uniqueness for each device and collection combination.

In addition, the support database includes a table named `metrics` for tracking processing metrics. Each time a worker completes processing a list of devices, it logs a new entry in the `metrics` table with details such as `First device processed`, `Last device processed`, `Devices to query`, `Documents uploaded`, `Worker computation time`, `Total runned queries`, `Devices processed`, `Collection`, and `Number of errors`. These metrics provide valuable insights into job performance, allowing for monitoring and optimization of the data ingestion pipeline.

Master Function

The *master* function begins by querying all devices in the `device` table to obtain their `last_log_date` timestamps. These timestamps represent the latest data logged by each device, which is essential for determining if any new data needs to be processed. For each collection in the system, the function also retrieves the `lastts` timestamp for each device from the `lastts` table. This `lastts` timestamp indicates the most recent data ingestion point for that collection. If a device's `lastts` does not exist, or if its `last_log_date` is greater than its `lastts`, the device is added to an array called `devicesToQuery`, marking it as ready for further processing.

Once the `devicesToQuery` array is populated for each collection, the function organizes these devices into batches for parallel processing. The batching is controlled by `workerSize`, which is calculated based on the total number of devices and the preset number of workers specified in `workersPerCollection`. This object defines the desired number of workers per collection, with collections containing a higher volume of documents assigned a greater number of workers to manage the load more effectively. Alternatively, it is possible to use the *split mode* option, which instead of batching devices according to a predefined number of workers, specifies a target number of devices per worker, dynamically adjusting the number of workers invoked according to the number of devices identified for processing.

The code snippet below demonstrates the batching and invocation of workers:

```
const workerSize = Math.ceil(devicesToQuery.length /
    workersPerCollection[operationalMode][collection])
for (let i = 0; i < devicesToQuery.length; i += workerSize)
    invocations.push(invokeWorkerWithTimeout(devicesToQuery.slice(i,
    i + workerSize), collection))
```

Code 3.3: Device batching

The master function manages all worker invocations using `Promise.allSettled(invocations)`, which executes all promises concurrently and returns an array of results, one for each promise. In JavaScript, a promise represents an operation that will complete asynchronously in the future, either successfully or with an error. Each result from `allSettled` indicates whether the corresponding promise was fulfilled or rejected. This approach ensures that all worker executions complete, regardless of individual failures, without interrupting the entire process.

The following functions handle each worker invocation:

```
async function invokeWorkerWithTimeout(dev: number[], collection: string) {
  const timeoutPromise = new Promise<never>((_, reject) =>
    setTimeout(
      () => reject(new Error(‘Timeout for worker ${collection}${dev[0]}‘)),
      WORKERS_TIMEOUT,
    ),
  );

  return Promise.race([invokeWorker(dev, collection), timeoutPromise]);
}

async function invokeWorker(dev: number[], collection: string) {
  const params = {
    FunctionName: ‘datalake-mongo-extractor-${process.env.NODE_ENV}-worker‘,
    Payload: JSON.stringify({
      body: {
        devices: JSON.stringify(dev),
        collection: collection,
      },
    }),
  };

  const command = new InvokeCommand(params);
  const response = await lambdaClient.send(command);

  // Handle worker Lambda response
  if (response.StatusCode !== 200) {
    throw new Error(‘Invocation failed for a worker‘);
  }

  const payload = JSON.parse(new TextDecoder('utf-8').decode(response.Payload));

  if (payload.errorMessage) {
    throw new Error(
```

```
      `${payload.errors} error(s) in worker ${payload.id} (${payload.
    log_stream_name})`,
    );
  }

  return payload;
}
```

Code 3.4: `invokeWorkerWithTimeout` and `invokeWorker` functions

The `invokeWorker` function constructs and sends the invocation command for each worker Lambda. If the response `StatusCode` is not `200`, an error is thrown, indicating that the invocation failed. Additionally, if there are specific errors reported in the worker's payload, these are also logged as errors.

In this process, three primary types of errors may occur. Each is handled independently by logging the error to avoid affecting other workers or the master function:

- **Worker invocation failure**: If `InvokeCommand` returns a `StatusCode` other than `200`, the invocation did not succeed.
- **Error within the worker**: If the worker encounters any issue, the master function receives a payload containing the number of errors that occurred. Full error details are logged within each individual worker, accessible via AWS CloudWatch.
- **Worker timeout**: If a worker does not complete within 15 minutes, the issue is managed by `Promise.race()` to which two promises are passed: the actual worker invocation (`invokeWorker`) and a timeout promise. If the worker exceeds the predefined `WORKERS_TIMEOUT` duration, the timeout promise rejects, ensuring the master function is aware that the worker did not complete within the expected time.

In this setup, all master and worker logs are available in AWS CloudWatch, enabling easy access to error details and execution summaries. `CloudWatch` is an AWS tool for monitoring and managing log data from AWS services.

Once all worker responses are received, the master function logs summary statistics for each collection. This includes progress as a percentage of documents uploaded relative to the total documents in each collection. The uploaded document count is calculated by summing the values in the `docs_uploaded` field within the `metrics` table for a specific collection in the support database. The total document count is estimated using the MongoDB driver's `estimatedDocumentCount()` function.

Since `variable_logs_clean` is a view and does not support direct document counting, progress is calculated based on the percentage of devices processed relative to the total number of devices. The following function, `countTotalKeys`, counts the already processed devices:

```
async function countTotalKeys() {
  let totalKeyCount = 0;
  let continuationToken;
  let isTruncated = true;

  while (isTruncated) {
    const params: ListObjectsV2CommandInput = {
      Bucket: AWS_CONFIG.TARGET_BUCKET,
      Prefix: 'variable_logs_clean/',
      Delimiter: '/',
      ContinuationToken: continuationToken,
    };

    const data = await s3Client.send(new ListObjectsV2Command(params)
    );

    // Sum KeyCount from the current response
    totalKeyCount += data.KeyCount ?? 0;

    // Handle pagination
    continuationToken = data.NextContinuationToken;
    isTruncated = data.IsTruncated ?? false;
  }

  return totalKeyCount;
}
```

Code 3.5: `countTotalKeys` function

This function iterates through paginated S3 results to count the total keys, providing an approximate measure of processed devices. This estimation is more approximate than document counting and is mainly useful for bulk loads.

Finally, the master function returns a JSON response summarizing the number of successfully executed workers and overall progress, completing the orchestration for data processing and upload.

Worker function

As previously seen, the master function invokes each worker function with two parameters: the `collection` to process and a list of `devices`. The worker function begins by adding all devices to an *async queue* with a concurrency level of 5, meaning that up to five tasks can be executed concurrently. An *async queue* allows tasks to be processed asynchronously, enabling the execution of multiple tasks in parallel. However, it should be noted that, due to Node.js's single-threaded nature, this parallelism is achieved via asynchronous handling rather than true parallel threads.

Each task in the queue processes a single device, querying and loading its data into the data lake in manageable portions. Given the high volume of data, it is necessary to retrieve records in smaller segments. Starting from the earliest available data in 2015, the function performs sequential queries, each covering a 60-day period until it reaches the device's `last_log_date`. Due to AWS Lambda's 15-minute maximum execution time, all queries must complete within 10 minutes, leaving a 5-minute buffer to finalize the last query without risking timeout.

Each execution task in the queue involves a `while` loop, iterating as long as the total runtime is under 10 minutes. In each iteration, the loop defines the query period's boundaries: `from` and `to`. The `from` value is set to the device's `lastts` if available; otherwise, it defaults to October 2015 (the beginning of MongoDB data storage). The `to` value is determined as the minimum between `from + 60 days` and `last_log_date`. If `from` is greater than or equal to `to`, the loop ends, indicating all data has been imported for that device.

Query Construction

Once the time boundaries are defined, the worker constructs a MongoDB query tailored to the specific `collection`. Since each collection has its own structure and timestamp field names, a custom query format is created for each one. For instance, in the `variable_logs_clean` collection, the query is structured as follows:

```
variable_logs_clean: (device, from, to) => ({
    idDevice: device,
    day: {
      $gte: new Date(from.getTime() - (from.getTime() % 21600000)),
      $lte: new Date(to.getTime() - (to.getTime() % 21600000)),
    },
```

```
    time: { $gt: from, $lte: to },
})
```

This query filters by `idDevice`, with `from` and `to` as the start and end times, respectively. The time field is represented by both day and time, this is because, as we said, `variable_logs_clean` is a view and day is used to exploit the day index in the real `variable_logs` collection.

The MongoDB query then runs, with results stored in `queryResult`. After obtaining the data, it undergoes two main aggregation steps to ensure consistency and format the records properly:

- **Common aggregation**: Applied to all collections, this step moves non-standard fields (those not defined in the Parquet schema for that collection) into a `payload` field. This standardizes each document by consolidating unusual fields.
- **Variable_logs_clean aggregation**: Specific to the `variable_logs_clean` collection, this aggregation combines all measurements with the same timestamp from a single device into one document.

Data Loading to the Data Lake

Once the documents are ready for data lake ingestion, the worker function performs three crucial steps: updating the `lastts`, converting JSON documents to Parquet format, and uploading the data to an S3 bucket. It is essential that these three operations are executed robustly, meaning that if any one of them fails, the other steps should not complete. To ensure this robustness, these operations are encapsulated within a transaction managed by Prisma.

In Prisma, *interactive transactions* provide a way to bundle multiple database operations into a single logical unit. Within an interactive transaction, either all operations are successfully completed and committed, or if any operation fails, the transaction is rolled back, undoing any changes made during the transaction. In this context, if either the conversion to Parquet or the upload to S3 fails, the transaction is not committed, effectively rolling back the `lastts` update. Consequently, the next execution will attempt to re-import the same data. Additionally, if a transaction-level error occurs, the worker ensures any partially uploaded files are removed, maintaining data consistency.

For the conversion of JSON documents to Parquet format, the `parquetjs`[2] library was utilized. `parquetjs` allows for efficient transformation of JSON data

[2]https://www.npmjs.com/package/@dsnp/parquetjs

into Parquet format, a columnar storage format optimized for high-performance querying. The library supports schema definitions, so each collection's schema is specified, ensuring consistent data structure in the output files. Furthermore, both the conversion and upload processes are performed using *streaming*, allowing the function to handle large data volumes without overwhelming memory.

Streaming in Node.js enables data to be processed in chunks, where each data chunk flows continuously from one stage of the pipeline to the next without waiting for the entire dataset. In this implementation, the streaming pipeline begins with a `Readable` stream that reads data from `documentsAggregated`, applies the Parquet transformation, and finally streams the transformed data to the S3 destination. Here is the code that accomplishes this:

```
const destination = new PassThrough();
const reader = Readable.from(documentsAggregated);
const pt = new ParquetTransformer(parquetSchemas[params.body.
    collection]);

await new Promise<void>((resolve, reject) => {
  pipeline(reader, pt, destination).catch((err) => reject(err));

  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: AWS_CONFIG.TARGET_BUCKET,
      Key: `${params.body.collection}/${device}/${from.getTime()}.
    parquet`,
      Body: destination,
      ContentType: 'application/parquet',
    },
  });
  upload
    .done()
    .then((res) => resolve())
    .catch((err) => reject(err));
});
```

In this code:

- `destination` is a `PassThrough` stream that serves as the final output in the pipeline.
- `Readable.from(documentsAggregated)` creates a readable stream from the JSON documents, allowing them to be processed sequentially.

- `ParquetTransformer` applies the Parquet schema transformation, converting the JSON data into Parquet format in real-time.
- `pipeline()` connects the readable stream, Parquet transformation, and `destination` stream in a sequence, with errors caught and handled via `reject`.

The transformed data is then uploaded to S3 using the SDK's `Upload()` function, which reads from `destination` and streams data directly to the S3 bucket, saving memory and speeding up the process.

Once the queue is empty, either due to reaching the time limit or because all devices have been processed, a new entry is added to the `metrics` table in the support database. This entry records various metrics measured during execution, such as the number of processed devices, total queries executed, and uploaded document count. These metrics are also logged to AWS CloudWatch, where they can be monitored for performance analysis and troubleshooting.

Finally, the worker function returns a response to the master function, including the `statusCode`, the unique Lambda identifier, and any error counts detected during execution.

3.4 Data Integration

As described earlier, the extracted data is directly stored in two different S3 buckets. AWS S3 is an object storage service that organizes data within containers called *buckets*, each capable of storing virtually unlimited objects. A bucket does not function as a typical directory; rather, each file within S3 is identified by a unique *key*, which may include prefixes resembling folder structures. In this system, PostgreSQL data is stored in a bucket named `datalake-postgres`, where each table is assigned a distinct prefix corresponding to its table name. MongoDB data, on the other hand, is stored in a bucket named `datalake-mongodb`, organized with prefixes based on collection names and device IDs.

As introduced in Section 3.1.2, the data lake structure is composed of three layers: *raw*, *curated*, and *analytics*. For PostgreSQL data, it was decided to combine the raw and curated layers, as data can be cleansed during extraction through the Glue ETL job. Maintaining an additional extraction job and S3 bucket for raw data was deemed unnecessary, given that PostgreSQL tables are relatively lightweight and do not justify the complexity of a separate raw layer.

For MongoDB, however, an ETL job is required to transform data from the raw layer to the curated layer. The structure of this job is similar to that described in Section 3.3.1, with each collection processed sequentially. It comprises a source node (reading from the raw layer), a transformation node, and a target node (writing to the curated layer). For this transformation, Glue bookmarks are employed to track progress when reading from S3. Unlike JDBC bookmarks, which record the last processed primary key, S3 bookmarks store the last modified timestamp of the files read.

Currently, this ETL job performs four key functions:

1. conversion from Parquet format to Iceberg,
2. Snappy compression,
3. partitioning,
4. and deduplication of the `alarm` collection.

Transforming MongoDB tables to Iceberg format is beneficial primarily for two reasons: Iceberg enables optimized table storage through table optimizations and enhances partitioning capabilities. Table optimizations are covered in detail in Section 3.5, as these are applied directly within the data catalog. Instead, Iceberg's partitioning facilitates effective partitioning based on metadata for `timestamp` columns, as well as customizable partitioning by year, month, or day.

Partitioning of collections was implemented to improve query performance by reducing the scope of data scanned [13]. Specifically, all collections are partitioned by `idDevice`, and larger collections are additionally partitioned by year. By leveraging Iceberg's hidden partitioning, data can be partitioned based on timestamps down to the year, month, or day level. To determine the optimal time granularity for partitioning, several tests were conducted (see Section 4.2), as daily partitioning is not always ideal. While partitioning by day minimizes the amount of data scanned for short-term queries, it significantly increases I/O operations for long-term queries. This results in multiple small files for daily partitions, whereas a yearly partition would only require a single file and a single I/O access.

In the context of company data, the daily volume for each device and collection does not justify creating a separate file. Additionally, most queries target extended time periods, so partitioning by year was chosen to minimize I/O overhead, thus improving response latency. The downside of this choice is a potential increase in scanned data for certain queries, which may elevate query costs (particularly when using AWS Athena, where costs are based on scanned data size).

Deduplication of `alarm` documents

Deduplication of documents in the `alarm` collection is necessary because `alarm` is the only collection in the data lake that may undergo updates. Each document in `alarm` includes fields such as `id` (document identifier), `idDevice` (device identifier), `code` (alarm code), `activeTS` (alarm activation timestamp), `resetTS` (alarm reset timestamp), among others. Here, `resetTS` is always greater than or equal to `activeTS`, but a document may initially lack a `resetTS` if the alarm is active when recorded in the database. The document is subsequently updated with a `resetTS` once the alarm is deactivated. Consequently, documents are added to the data lake if either `activeTS` or `resetTS` is greater than the `lastts`. Depending on the status of these timestamps during data ingestion, three scenarios can arise:

- `activeTS` and `resetTS` are both greater than `lastts`: No deduplication is required.
- `activeTS` is greater than `lastts` and `resetTS` does not exist: The document will be re-imported when `resetTS` becomes available, requiring deduplication.
- `activeTS` is less than `lastts` and `resetTS` is greater than `lastts`: The document is imported, but since it is already present in the data lake, deduplication is required.

Glue ETL manages deduplication through two methods:

- For duplicate documents imported within the same job, deduplication is achieved via a `groupBy` operation in PySpark, aggregating by `resetTS`:

```
    .groupBy("id", "class", "activationMail", "activeTS", "code"
    , "iddevice", "resetMail", "payload")
    .agg(max("resetTS").alias("resetTS"))

```

- For duplicate documents imported by different jobs, deduplication is handled using Apache Iceberg's `MERGE INTO` operation. If a document with the same `id` already exists in the Iceberg-formatted S3 target, the row is updated with the latest `resetTS`:

```sql
MERGE INTO glue_catalog.mongodb.alarm t
USING merge_source s
ON t._id = s._id
WHEN MATCHED THEN
    UPDATE SET t.resetTS = s.resetTS
WHEN NOT MATCHED THEN
    INSERT *

```

Finally, the analytics layer was designed to facilitate specific data aggregations for targeted analyses. A base job and an S3 bucket were established for this layer, providing flexibility for future recurring analyses. For instance, this layer can support tasks such as joining multiple tables with column-level aggregations.

3.5 Data Cataloguing

The construction of the data lake is completed by cataloging the various tables. The AWS Glue Data Catalog serves as a centralized metadata repository that organizes, describes, and indexes datasets stored within the data lake, making them accessible and queryable through services like AWS Athena. This cataloging process facilitates data exploration, searchability, and consistency across the data pipeline. For convenience, two distinct catalogs were created in this setup: `Postgres` and `Mongodb`, each corresponding to a database within the Glue Data Catalog. Each database contains all the respective tables present in the various data lake layers.

There are two primary methods for creating a catalog (i.e., a database and tables) in Glue:

- **Using the Glue Crawler**: Glue crawlers automate catalog creation by scanning specified data locations, identifying data formats and schemas, and creating or updating tables. Crawlers work by inspecting a dataset, inferring its schema, and periodically refreshing the catalog entries to ensure metadata remains current.

- **Direct Table Creation within Glue Jobs**: Glue jobs can automatically create tables in the catalog when writing data to S3. This method, triggered at the time of data ingestion, enables Glue to create or update tables based on the data format, schema, and location specified within the job.

For the curated and analytics layers, the second method was applied, allowing tables to be cataloged as soon as data is written to S3. However, for MongoDB's raw layer, using a crawler is mandatory to capture the unprocessed data format, as direct cataloging through Glue jobs is not supported. Since the raw layer is considered less relevant for analytical use, the curated layer was cataloged instead, eliminating the need to crawl the raw layer tables with each system execution, thereby reducing both time and cost.

Tables stored in S3 are available in either Parquet or Iceberg formats. In terms of cataloging, the Glue Data Catalog handles both formats similarly, recording schema, partitioning, and metadata, allowing queries to be executed regardless of format.

Table Optimization for Iceberg in the Glue Data Catalog

One of the powerful features in the Glue Data Catalog for Iceberg tables is *table optimization* [14]. The most effective optimization, among those offered by Glue, is the *table compaction*. This feature addresses performance and storage efficiency by compacting small files within the table, a process especially useful for managing data updates. Each system execution potentially generates new files as data is added or updated, which could lead to numerous small files. File *compaction* in Iceberg combines these smaller files into larger ones, reducing file fragmentation and optimizing read performance by lowering the number of I/O operations needed for queries.

Compaction is particularly advantageous in this context, as it consolidates updated data added with each execution cycle, thus enhancing query performance and minimizing storage costs. This process not only improves the data lake's efficiency but also helps manage the storage footprint of frequently updated collections, making it easier to manage large datasets effectively over time.

3.6 Direct Queries

For on-demand data queries, AWS Athena provides a flexible querying tool that supports ANSI SQL, enabling users to execute complex operations such as joins, window functions, and array manipulations. Query results can be viewed directly on the console or downloaded as CSV files for further use. Athena relies on the Presto and Trino engines, which are optimized for large-scale data processing and integrate optimally with the Glue Data Catalog, making all cataloged data accessible without additional configuration.

The Athena interface displays the tables in the `Postgres` and `Mongodb` catalogs, providing the possibility to preview, create, or delete them and run SQL queries with ease. Autocomplete suggestions speed up query creation, helping reduce errors, and users can save frequently used queries to streamline repetitive tasks.

A notable advantage of Athena is its integration with Iceberg, which brings version control and efficient update handling directly into the data lake. With Iceberg, Athena enables point-in-time queries that simplify historical data analysis without requiring duplicated data versions.

For more complex data transformations or iterative analyses, PySpark and Spark SQL offer complementary options. These tools enable large-scale distributed processing, allowing users to employ both Python and SQL syntax for diverse analytical needs, integrating with both Parquet and Iceberg data formats.

Athena also provides comprehensive query statistics, including execution time, data scanned, and cost metrics. These insights are crucial for query optimization, helping users adjust their queries to improve efficiency and reduce costs in future executions.

3.7 Report creation

For BI reporting, Amazon QuickSight has been primarily used to build and publish analyses. Creating an analysis with up-to-date data in QuickSight involves three main steps: preparing the datasets, building the analysis, and publishing it as a dashboard.

In QuickSight, it is not possible to directly use tables cataloged in the Glue Data Catalog; instead, specific datasets must be created. QuickSight distinguishes between *Data Sources* and *Datasets*. A Data Source represents the underlying connection to a specific data repository, such as an S3 bucket, RDS database, or data lake, while a Dataset refers to the actual data used in the analysis, which can integrate one or more data sources.

Datasets in QuickSight can undergo several preparatory transformations to align data with analysis requirements. Users can exclude unnecessary fields, edit existing fields, change data types, add calculated fields, and apply filters to refine the data. Datasets can operate in two distinct modes: *Direct Query* or *SPICE* [17]. In Direct Query mode, each query retrieves data in real-time from the source, whereas SPICE (Super-fast, Parallel, In-memory Calculation Engine) stores a cached version of the dataset, improving performance but requiring periodic refreshes to maintain updated data. To avoid redundancy, further details about SPICE can be referenced in Section 2.2.8. SPICE datasets need a defined schedule for refresh operations, which can be set as either `full` or `incremental`. Incremental refreshes are particularly efficient, as they only update records that have changed since the last refresh. For MongoDB datasets, enabling incremental refresh required adding a processing timestamp to each document during the ETL transformation in Glue, which provides the necessary update information for each record.

Deciding whether to store a dataset in SPICE or Direct Query mode requires careful consideration, as SPICE incurs a monthly cost of $0.38 per GB stored. With storage requirements for SPICE generally up to 5 times higher than for S3, and knowing that all 13 collections in the curated layer collectively require around 300GB in S3, using SPICE exclusively would cost approximately 570$. Due to these significant costs, only the most frequently accessed tables are stored in SPICE. The `variable_logs_clean` collection, which would contribute disproportionately to the cost due to its size, is retained in Direct Query mode.

With datasets prepared, analyses can be built. Each analysis in QuickSight can only contain a single dataset, so it is essential that the selected dataset includes all the necessary fields and data. In QuickSight, an analysis is structured as a collection of *sheets*, each of which acts like a separate page within the analysis. Each sheet provides a flexible layout where users can position multiple *visuals*, add text boxes, and even insert images to create a cohesive view of the data. Users can freely arrange these elements, allowing for a customized layout where visuals can be displayed side-by-side, stacked, or arranged in grids. Text boxes and images are added to provide context, explanations, or branding, supporting a more comprehensive presentation of data insights. This flexibility helps users organize and present data on different aspects within a single analysis e.g., a sales analysis might have one sheet focused on revenue trends, another on product performance, and a third on regional comparisons.

Within each sheet, visuals play a central role. A *visual* is a chart type selected to represent the data, with options ranging from line and bar charts to scatter plots, pie charts, and heat maps. Each visual type has parameters specific to its format, allowing users to control how the data is represented. Adding data to visuals is simple and intuitive: users can drag and drop columns from the dataset directly into the visual's parameter fields. For example, to create a line chart showing monthly temperature trends, one could drag a `Date` column to the X-axis and a `Temperature` column to the Value field. This drag-and-drop interface makes it easy to experiment with different columns and chart types, adapting the visuals to best fit the data and the analysis goals.

QuickSight also supports complex filtering options to control which data is shown in each visual. Filters can be applied at multiple levels of granularity, giving users flexibility in how they present their data. Filters can be applied at three levels:

- **Visual-level filters**: These filters apply only to the data within a specific visual. For instance, if a visual shows cooking programs, a filter can be set to display only the programs of a specific category, such as "Washings".
- **Sheet-level filters**: Filters applied to an entire sheet affect all visuals within that sheet. This is helpful when analyzing a particular subset across multiple visuals on the same sheet. For example, on a sheet with various charts showing sales data, a filter could limit all visuals to data from the last fiscal quarter.
- **Analysis-level filters**: These global filters apply to the entire analysis, affecting every sheet and visual. This approach is useful for setting consistent filter criteria across all sheets, such as showing only data for Gas ovens throughout the analysis.

QuickSight also allows users to create *calculated fields*, which generate new data columns based on existing fields. To create a calculated field, users write a formula using QuickSight's built-in functions [4], which include mathematical, statistical, and string operations. For example, a calculated field showing the duration of a program could be created using the formula:

```
dateDiff(endTS, startTS,"MI")
```

This formula, once defined, becomes a new column available for use in any visual within the dataset, making it easy to integrate custom metrics directly into the analysis.

Once an analysis is complete, it can be published as a *dashboard*, making it available to other stakeholders. Dashboards in QuickSight are interactive, enabling viewers to filter data, view specific metrics, and drill down into key insights. This interactivity makes dashboards a powerful tool for communicating complex data analyses in an accessible format and supports data-driven decision-making by delivering relevant insights to a wide audience.

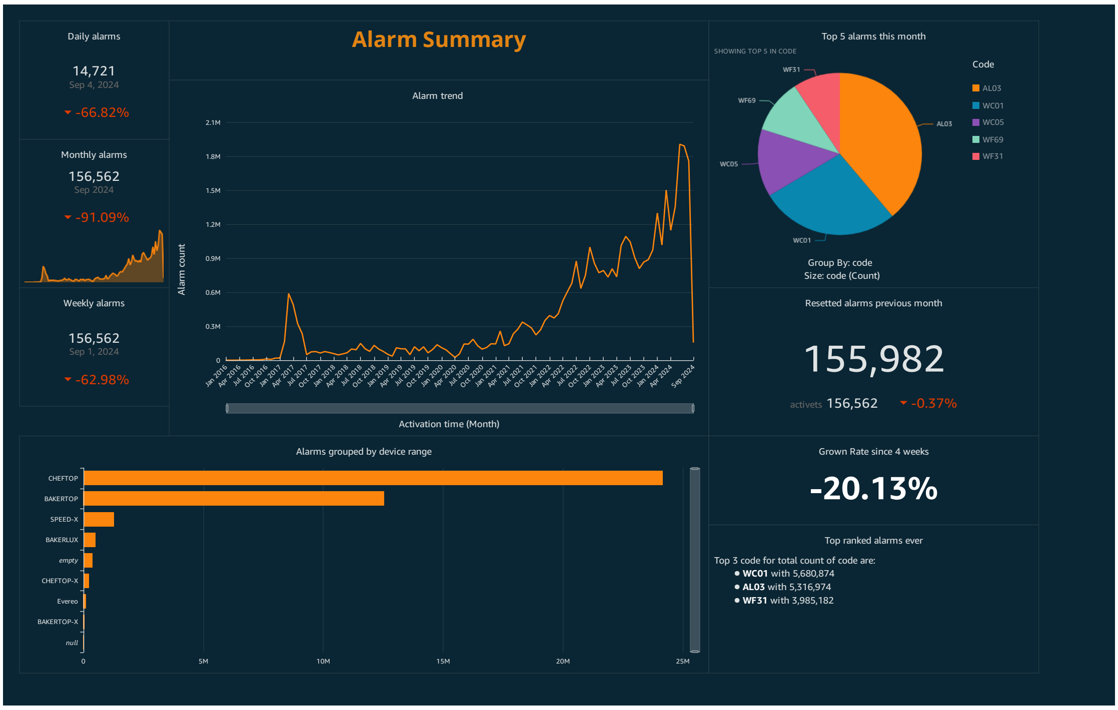

Figure 3.4: Quicksight Dashboard about Alarm Summary

3.7.1 Analysis with Generative AI

Incorporating generative AI into data analysis enhances user accessibility to data insights without requiring specialized data expertise. Amazon QuickSight Q, an innovative feature within AWS QuickSight, empowers users to interact with data using natural language queries. By enabling users to ask questions in plain language, Q lowers the technical barriers to data analysis, making advanced insights accessible to non-technical stakeholders and expanding the utility of business intelligence resources.

QuickSight Q integrates with the QuickSight dashboards, allowing users to query directly from dashboards without navigating complex query tools. This capability supports a range of question types, including queries about metrics, historical comparisons, and trend analysis. For example, a user can ask, How many ovens were produced in the last quarter? or How does current year performance compare to last year? QuickSight Q interprets these queries, executes the necessary computations, and presents results as visualizations or summary metrics directly in the dashboard interface. This minimizes time-to-insight by reducing dependency on SQL or other query languages. In addition to this, Q allows the creation and formatting of specific visual elements or calculated fields by means of instructions provided in natural language, thus speeding up and simplifying their management.

Quicksight Q not only offers AI functionality into dashboards but also allows *topics* to be constructed. Topics are structured data models tailored for natural language queries. Unlike dashboards, which present pre-configured visualizations, topics are designed to interpret and respond to dynamic, ad hoc questions by users. Each topic defines a specific dataset, along with custom metadata such as field definitions, synonyms, and data relationships optimized for natural language processing. Administrators configure topics to reflect business-specific terminology and metrics, allowing QuickSight Q to parse queries accurately based on relevant data fields. This setup enables Q to deliver precise responses, making it ideal for scenarios where users need on-demand insights beyond the fixed queries and visualizations typically available in dashboards.

Unfortunately, the cost of using these features is not moderate; in fact, activation of Q requires a fixed $250 each month in addition to the per-user costs described more precisely in Section 4.4.2.

In summary, by implementing Q, this system leverages AI-driven natural language processing to simplify and accelerate data analysis. This promotes a more data-driven culture throughout the organization.

3.8 Orchestration and Scheduling

To fully automatize the data system, orchestrating the execution of each component in a specific sequence is essential. Some services must be triggered at precise times to avoid overlaps, and certain operations need to follow a particular order so that the output of one stage can serve as the input for the next. This orchestration is managed using Amazon's Step Functions, a serverless workflow service that coordinates multiple AWS services into a series of steps defined by a state machine.

Step Functions enables complex workflows by chaining together a series of tasks, where each step can be a distinct AWS service task, such as a Lambda function or an activity integrated through APIs. The state machine, defined in ASL, specifies the order, conditions, and dependencies between steps, and it allows for retries, branching, parallelization, and error handling. This flexibility is especially advantageous when coordinating components that need to process and transform data sequentially and concurrently.

In this data import process, two separate workflows, one for *bulk load* and one for *operational* load are implemented. These workflows differ significantly in data volume: the bulk load workflow handles large amounts of data quickly, while the operational workflow supports smaller, more regular data imports. To address these requirements, two distinct Step Functions were created.

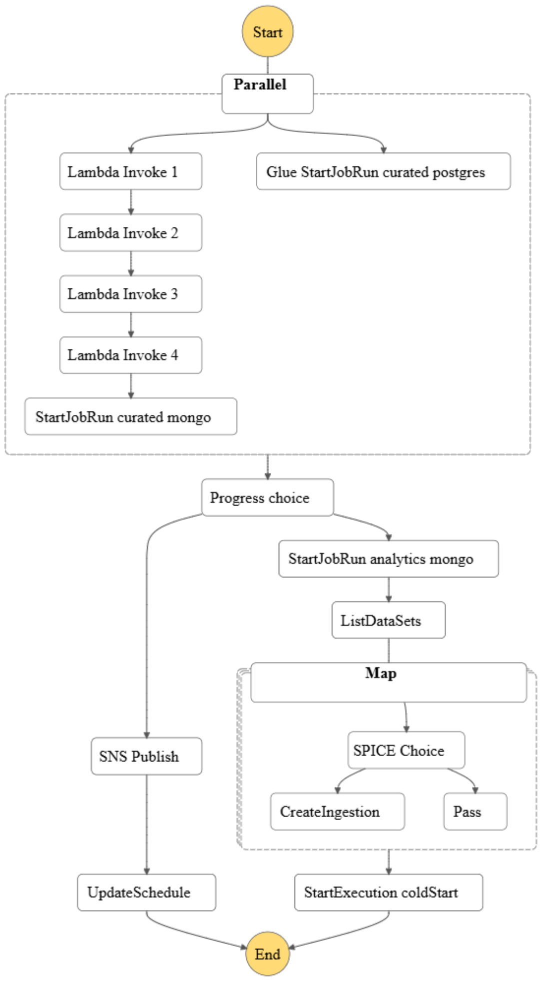

Figure 3.5: `datalake-cold-start` Step Function for Bulk load

The `datalake-cold-start` Step Function manages the bulk load. It starts with a parallel phase where two branches execute concurrently to optimize processing time. The first branch handles MongoDB data import and the second focuses on PostgreSQL data. In the MongoDB branch, four sequential invocations of the master Lambda function occur, each of which may trigger up to 30 worker Lambdas. Thus, this branch can potentially invoke up to 120 workers to facilitate data extraction. After these workers complete their tasks, a Glue job is launched to transform the extracted data for the curated layer. In the PostgreSQL branch, the Glue job dedicated to extracting PostgreSQL data is executed.

After the two branches complete, they merge at a decision point, where the system evaluates the progress. If the system has processed over 95% of the devices, the workflow transitions to the operational Step Function; otherwise, it re-triggers the same bulk load state machine.

The progress percentage is determined by calculating the ratio of processed devices to the total number of devices for the `variable_logs_clean` collection. This value is computed by the master Lambda, logged in CloudWatch, and included in the Lambda's response under the `progress` key. Within the workflow, the progress value from the fourth Lambda invocation is retrieved, passed to the MongoDB Glue job, and subsequently used as a parameter throughout the parallel stage, leading up to the decision state.

If the progress is below 95%, the initial ingestion phase must continue. The analytics layer Glue job is triggered, followed by a refresh of all QuickSight datasets. At this stage, the `ListDataSets` API in QuickSight retrieves a list of available datasets. This list is processed in a `Map` state, iterating over each dataset. For each dataset, `SPICE Choice` evaluates whether it meets specified conditions, such as containing a valid ID, name, and specific keywords corresponding to the datasets requiring a refresh. In this setup, all SPICE datasets must be manually included in these conditions. Only datasets that satisfy the criteria undergo an incremental refresh, with refresh parameters, including timestamp columns, configured directly in QuickSight during dataset setup.

After the QuickSight refresh completes, the state machine executes the `StartExecution coldStart` state, launching another state machine to continue the cold start phase.

However, if the progress reaches or exceeds 95%, the system sends a notification via Simple Notification Service (SNS). SNS is a fully managed messaging service that sends notifications from one publisher to multiple subscribers. In this workflow, an SNS message is configured with the subject "Data lake status" and the specific message "End of cold start phase. Starting operational mode". This notification informs by email both myself and the team leader that the bulk load phase has completed and that the system is transitioning to operational mode. Finally, an API function call updates an EventBridge Scheduler, which I will discuss in detail in the following section.

The EventBridge Scheduler The EventBridge Scheduler was employed to schedule the operational Step Function flow, as continuous data import was not necessary at this stage. During the bulk load phase, the state machine was configured to auto-trigger in cycles without any pause, making a scheduler redundant. However, the operational phase requires a more controlled data import frequency, typically determined by business needs, whether data updates are required hourly, daily, or weekly. In this case, as near real-time updates were not deemed essential, a daily schedule was chosen to balance data currency and cost-effectiveness.

EventBridge automates the execution of workflows by scheduling events based on pre-defined triggers. To set up this scheduler, a *cron expression* was defined, which is a time-based syntax specifying when the event should trigger. The cron expression used in this case, `0 1/2 * * ?  *`, instructs EventBridge to run the operational workflow every day at a specific hour, giving fine control over execution timing. This cron pattern consists of five fields that represent minute, hour, day of month, month, and day of week values.

When initially configuring the EventBridge event, it was set to `disabled`. The final task within the bulk load Step Function enables this event, activating the daily schedule. Once activated, the scheduler initiates the operational Step Function as described later, automatically running the workflow on a daily basis. The scheduler settings, including the cron expression, can be modified or temporarily disabled if business requirements change.

Operational Step Function The *operational Step Function* closely resembles the `cold-start` workflow in structure, with two key differences. Firstly, in the operational phase, only a single invocation of the Lambda function is required to handle MongoDB data import, as one execution is sufficient to process the data accumulated over a single day. This simplification reduces system overhead while keeping the data imports timely.

Secondly, the operational workflow omits the progress check. Once the system transitions to the operational phase, reverting to a previous stage is unnecessary, and thus, tracking the progress of data loading becomes redundant. Additionally, there is no need for the Step Function to trigger itself upon completion, as the EventBridge Scheduler handles regular daily execution.

4

Experiments and Analyses

4.1 Alarms Report use case

To evaluate the overall system performance, we focused on reproducing a report that the company has routinely generated and analyzed. By doing so, we were able to directly compare the new BI approach with existing processes, using AWS tools to replicate the report generation.

The collection named `alarm` in MongoDB records all alarms triggered by devices, with each alarm containing fields such as the device ID, alarm code, activation timestamp, termination timestamp, and other optional details. The required output for this report is a pivot table: a format used to summarize, analyze, and reorganize data by grouping it across specific dimensions. In this case, the rows represent different devices, while the columns include device details (serial number, model code, family, range, and version) and all potential alarm codes. For each alarm code, we need the total count of occurrences from the past week.

To obtain device-specific details like serial number, model code, and family which are not present in MongoDB's alarm collection we also needed to reference the `device` table stored in PostgreSQL.

Previously, this report was generated through a JavaScript script that used the MongoDB driver to access alarm data and the ORM Sequelize to access PostgreSQL data. The script would create a CSV file that was then automatically emailed to key stakeholders. The entire process was managed by an AWS Lambda function scheduled for weekly execution.

To reproduce this outcome using the new BI system, we required data from both the alarm and device tables, which are stored in the data lake's curated layer. To link alarm events with device information, we performed a left join between the two tables, aligning each alarm with its respective device information. We explored three possible solutions for executing this join:

- **Data Lake Join in Analytics Layer**: In this approach, we added steps in the Glue job to read the two tables from the curated layer, join them on the device ID, and write the result in the analytics layer using the Iceberg format. For the alarm table, Glue bookmarks ensure that only new data is read, while the device table is read in full each time. Additionally, the column `uploaded_at` is updated to allow QuickSight to refresh only new data. Since both the alarm code and device code are labeled: `code`, one of them was renamed for clarity. This joined table, identified as "alarm_device" in the AWS catalog, is then ready to be imported into QuickSight as a SPICE or Direct Query dataset. This solution is ideal for regularly scheduled analyses, though it requires AWS Glue knowledge, making it less accessible for end users.

- **QuickSight Dataset Join**: QuickSight allows datasets to be created by joining other datasets through its visual interface. With both alarm and device as source datasets, users can create a joined dataset in QuickSight. This approach requires both the source and joined datasets to be in SPICE mode. When the analysis is recurring, daily incremental refreshes are necessary for each dataset, with the joined dataset refreshed last. This solution is more user-friendly and accessible for non-technical users who may need quick analyses, though it incurs storage costs for three SPICE datasets, making it unsuitable for large tables.

- **Athena Join and View Creation**: For users preferring QuickSight's Direct Query mode, the join can be executed in Athena by creating a view: a saved query that functions as a virtual table. This view can then be registered in the AWS catalog and used in QuickSight without requiring data refreshes, as queries are executed live.

For this use case, the first solution was selected, as it best supports recurrent analyses with frequently updated data. After creating the joined dataset, further preparation was necessary to ensure all fields were analysis-ready. Device details like family, range, and version are embedded within the model code, which is a 14-character alphanumeric string. Specifically, the third character of the model code identifies the family and range, while the twelfth character denotes the version. A calculated field was created for each missing attribute, re-implementing the existing company function to derive these values from the model code directly within QuickSight.

```
ifelse(substring(split({code},'-',1),4,1)='L','MIND.Maps BIG',
substring(split({code},'-',1),3,1)='V','MIND.Maps',
substring(split({code},'-',1),3,1)='B','MIND.Maps',
substring(split({code},'-',1),3,1)='C','MIND.Maps Compact',
substring(split({code},'-',1),3,1)='F','SHOP.Pro',
substring(split({code},'-',1),3,1)='E','Evereo',
substring(split({code},'-',1),3,1)='S','SPEED.Pro',
substring(split({code},'-',1),3,1)='D','Digital.ID',
substring(split({code},'-',1),3,1)='L','Digital.ID',
substring(split({code},'-',1),3,1)='P','Digital.ID',
'')
```

Code 4.1: Family calculated field

Once the dataset was fully prepared, we developed the report using a pivot table with the following configuration: device ID, serial, device code, family, range, and version as rows; alarm codes as columns; and the alarm code counts as values. A filter was applied to the alarm activation date, defaulted to "Last 7 days." The layout was refined to fill the view window, with conditional formatting applied to highlight cells with alarm counts over 50 in orange and over 100 in red.

The resulting analysis was published as an interactive dashboard (shown in Figure 4.1) shared with stakeholders, allowing them to adjust the alarm activation date filter to specific relative or absolute periods. With the large number of columns and rows, only visible portions of the table are loaded initially, optimizing performance and accessibility. This setup allows stakeholders to access daily-updated data and view historical data on demand, eliminating the need for weekly emails and file downloads.

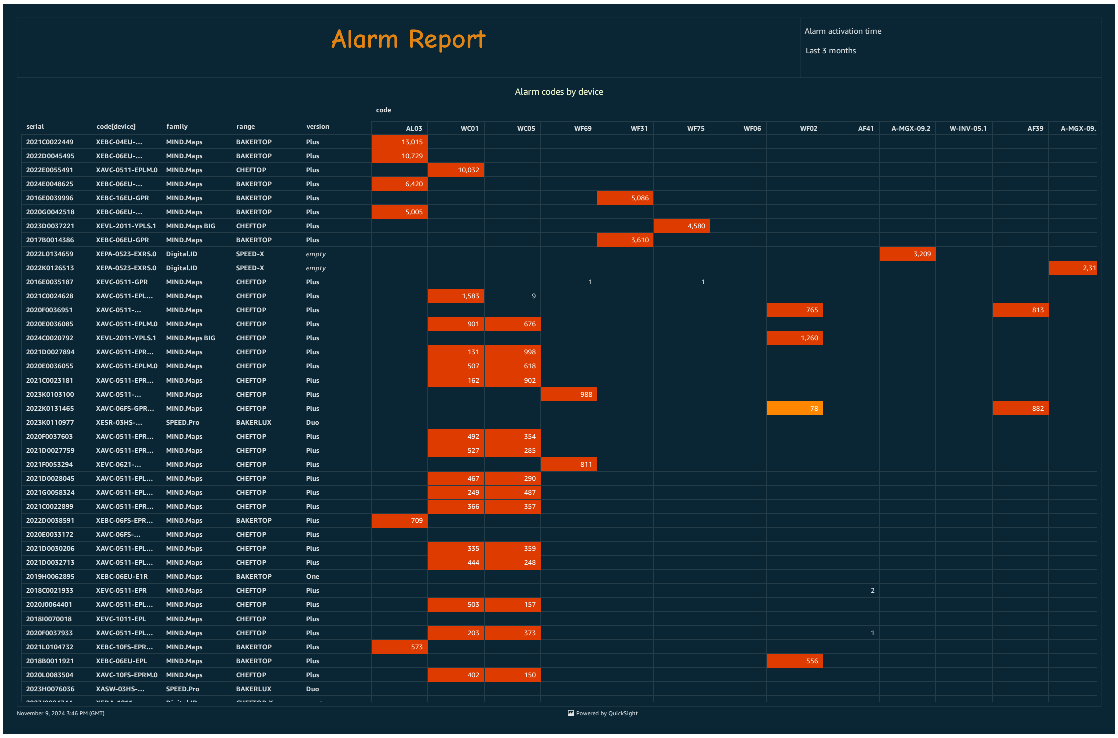

Figure 4.1: Alarm Report from Quicksight

Finally, generative AI was tested by creating a topic based on this dataset. It was necessary to define the scope of the questions, selecting and structuring the relevant data in order to narrow the field of questions that users can ask and make the answers more precise. Through a configuration process, it was possible to enrich the topic with synonyms and phrases commonly used to describe the various data fields. This allows QuickSight Q to recognise different formulations of the same question and still answer correctly.

4.2 Partitioning Strategy

The following experiment evaluates temporal partitioning strategies for the *mongodb* tables, comparing different levels of time granularity: no time-based partitioning, partitioning by year, and partitioning by month. Specifically, the `events` table was used for this test, which keeps track of all events occurring in the ovens; examples of frequent events are: door opening/closing, magnetron hour reset, dirt reset, etc. This evaluation aims to identify the impact of each strategy on query performance in terms of execution time and scanned data volume. In all cases, the data is firstly partitioned by device. Then, the analysis explores how adding various levels of time-based partitioning interacts with device-based partitioning, helping to identify the most efficient approach for complex queries involving both device ID and time filters.

Three types of queries were used for each partitioning strategy: a generic time filter (one year and one month), a filter by year, and a filter by month.

```
Generic Filter:
SELECT * FROM "mongodb"."events"
WHERE idDevice BETWEEN 1000 AND 5000 AND
ts BETWEEN TIMESTAMP '2023-01-01' AND TIMESTAMP '2024-01-31';

Year Filter:
SELECT * FROM "mongodb"."events"
WHERE idDevice BETWEEN 1000 AND 5000 AND
ts BETWEEN TIMESTAMP '2023-01-01' AND TIMESTAMP '2023-12-31';

Month Filter:
SELECT * FROM "mongodb"."events"
WHERE idDevice BETWEEN 1000 AND 5000 AND
ts BETWEEN TIMESTAMP '2023-01-01' AND TIMESTAMP '2023-01-31';
```

Code 4.2: SQL Queries used for Execution Time and Data Scanning Analysis

The execution time results, displayed in Figure 4.2, reveal some unexpected insights. Specifically, the queries on the non-partitioned table exhibit faster performance compared to the year- and month-partitioned tables. For instance, the generic filter query on the non-partitioned table completes in 53 seconds, compared to 58.3 seconds on the year-partitioned table and 54.2 seconds on the month-partitioned table. A similar trend is observed for year-specific and month-specific filters. This counterintuitive outcome can be attributed to the computational overhead of scanning partitions and managing the complexity of partitioned structures. Firstly, when data is partitioned, additional overhead arises from managing partition metadata, which can slow down query processing. Furthermore, partitioning the data into multiple files necessitates more I/O operations, as the database engine must access several files to retrieve the relevant data. Consequently, partitioning does not guarantee faster execution for all types of queries.

The data scanned, as shown in Figure 4.3, illustrates the effectiveness of partitioning in reducing the volume of data processed for targeted queries. For instance, the month-specific filter in the month-partitioned table scans only 44

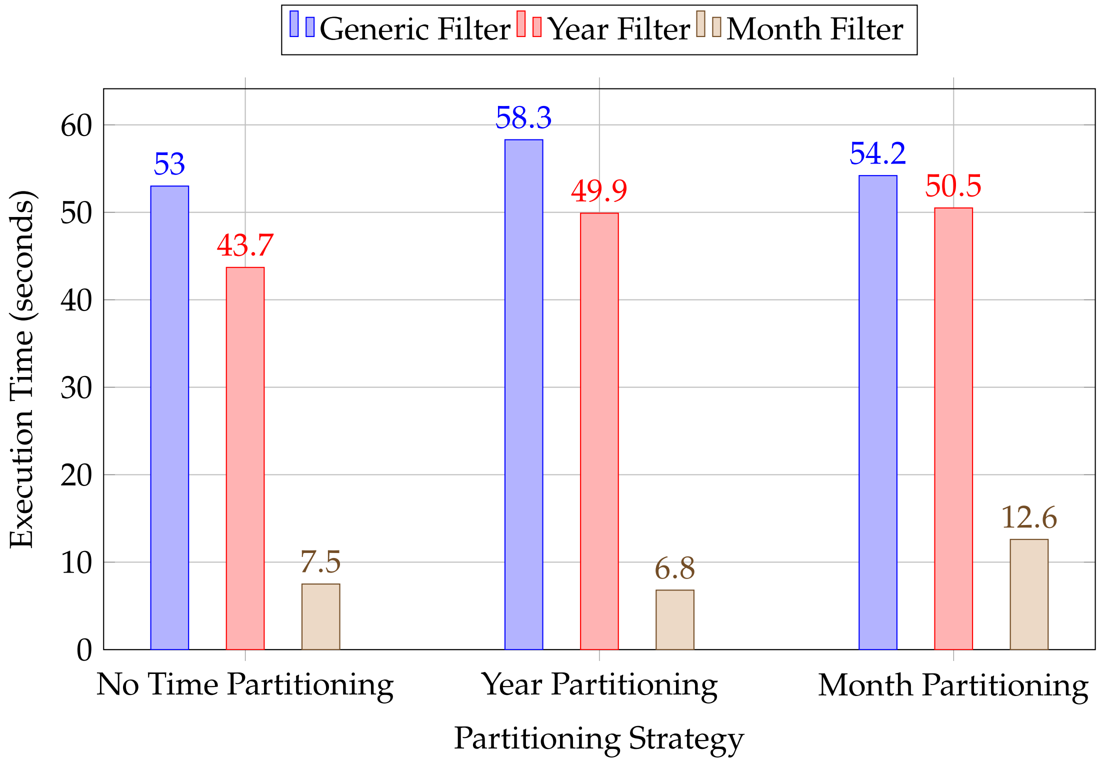

Figure 4.2: Query Execution Time by Partitioning Strategy for the `events` table

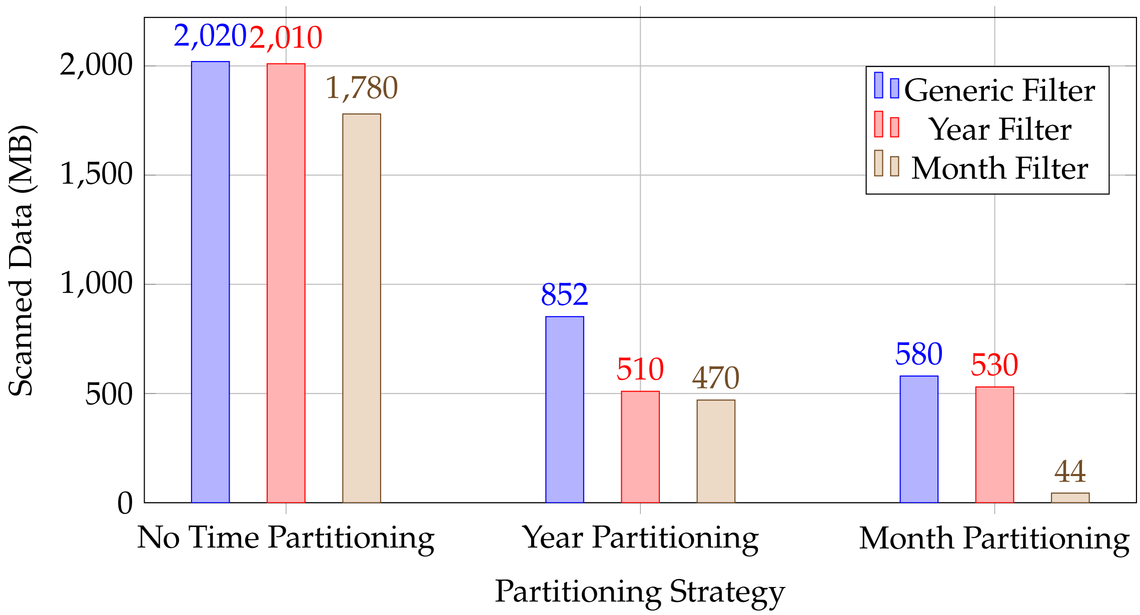

Figure 4.3: Scanned Data by Partitioning Strategy for the `events` table

MB, compared to 1,780 MB in the non-partitioned table and 470 MB in the year-partitioned table. This reduction demonstrates the value of partitioning in limiting data scans when filters closely match the partitioning strategy. Conversely, without partitioning, the table must scan a substantially larger amount of data, as all records are treated as a single, unstructured set.

The findings underscore the importance of selecting appropriate partitioning strategies based on the nature of the queries executed and the underlying data characteristics. While smaller partitioning intervals can enhance data efficiency and significantly reduce the volume of scanned data, they may also introduce a slight increase in execution time due to the necessity of additional metadata reads and higher I/O access counts. Therefore, a balance must be found between minimizing scanned data and optimizing execution times to achieve the best performance in data querying and processing.

Partitioning Strategy Results on "sd_variables" Table

The same test performed with events was re-executed on the table `sd_variables` to confirm the results obtained. "sd_variables" contains about 8 times fewer documents than events, but these are much heavier (see table 3.2). Specifically, "sd_variables" contains the IoT values generated by Digital ID devices, i.e. the highest oven range produced by Unox.

As shown in Figure 4.4, execution times for queries on "sd_variables" display only minor variations across partitioning strategies. In the case of year and month filters, the execution times remain relatively consistent across strategies, with a marginally faster performance for year-based and month-based partitioning compared to no partitioning.

In terms of scanned data, partitioning again shows a clear advantage in limiting the volume of data processed, especially for month-specific queries. For instance, the month-specific query scans only 170 MB in the month-partitioned table, while the no-partition approach scans up to 1,620 MB. These reductions in data scanning are similar to those noted in the "events" table, where finer-grained partitions significantly minimized data read volumes.

Based on these observations and tests, a year-based partitioning strategy has been chosen as an effective compromise. While finer-grained (month-based) partitioning minimizes scanned data, it often results in higher execution times due to the overhead associated with managing more granular partitions. Consequently, year-based partitioning offers a balanced trade-off, achieving acceptable query times while reducing data scans effectively for time-based filters.

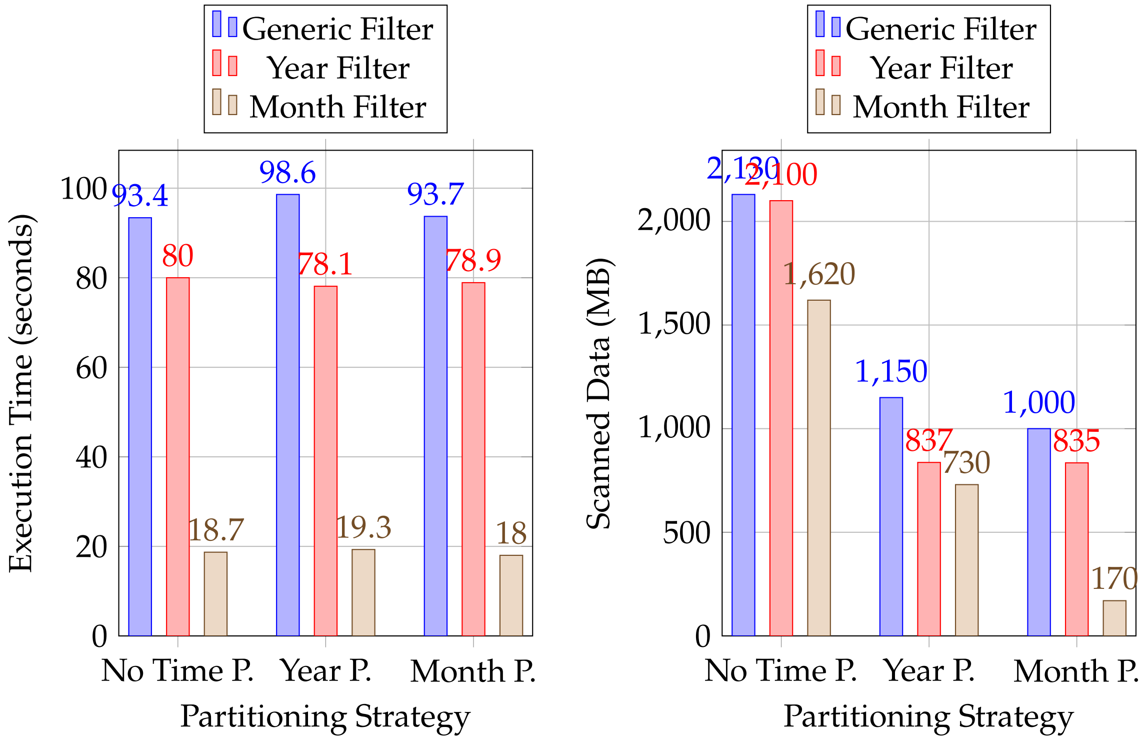

Figure 4.4: Query Execution Time and Scanned Data by Partitioning Strategy for `sd_variables` table

4.3 Benchmark of the BI System

This section presents a benchmark comparing query execution times on AWS Athena and the original MongoDB data sources. Since Postgres tables contain relatively small amounts of data and are generally not the bottleneck in business analyses, tests focused primarily on MongoDB tables. The queries on the original MongoDB source were executed using Studio 3T, a widely used GUI tool that facilitates query building, performance tuning, and data analysis. Before executing the benchmark, it was crucial to verify data consistency between the DBMSs and the data lake, especially considering frequent daily updates.

To ensure a reliable comparison, five representative queries were executed on both systems, with each query being run seven times on each system to calculate an average execution time. The test queries were executed on the `end_of_prog_aggregated` table, which contains a moderate number of documents with an average size compared to other tables.

The MongoDB performance is shown in two scenarios: with no optimizations and with optimizations. The available optimizations include:

- **Index Plan for a Query**: MongoDB caches the query execution plan, reducing the need to evaluate which index to use on repeated queries, which helps speed up execution.
- **Index Cached in Memory**: When an index is used, MongoDB caches it in memory, so subsequent queries using the same index can access it faster.
- **Data Cached in Memory**: MongoDB relies on the operating system's memory management system, which caches frequently accessed data. Once a document is read, it stays in memory, significantly reducing access time for future queries that request the same data.

The *index* in MongoDB is a data structure that speeds up query execution by enabling fast searching within the data [7]. When a query is run for the first time, MongoDB must scan through the dataset and load the necessary index into memory. Once cached, the subsequent executions of the same query are faster, as the data and index reside in memory. Each collection contains at least one index consisting of the *idDevice* and timestamp fields.

Therefore, the first execution of each query requires reading the data and index from the disk and moving them into memory, which takes an enormous amount of time. Moreover, considering the EC2 instance has only 32 GiB of memory, only a fraction of the datasets/indexes can be cached at any given time, making MongoDB's performance highly dependent on caching.

The following is a brief description of the five test queries executed on both systems:

- **Query 1**: Returns documents generated after January 1, 2024.
- **Query 2**: Groups documents by device and calculates the average energy consumed during cooking for each device.
- **Query 3**: Filters documents with daily cooking time greater than one hour, groups them by device, and calculates the total cooking time, then orders the results by descending cooking time.
- **Query 4**: Groups devices into five cooking times ranges, counts the number of devices in each range, and calculates the average cooking energy for each range, presenting the results in ascending order.
- **Query 5**: Performs a join between the "end_of_prog_aggregated" table and the `sdata` table (containing device configuration information) based on the device ID.

In the first four queries, 5000 devices were filtered to reduce execution time, while in the last query only 500 devices were filtered.

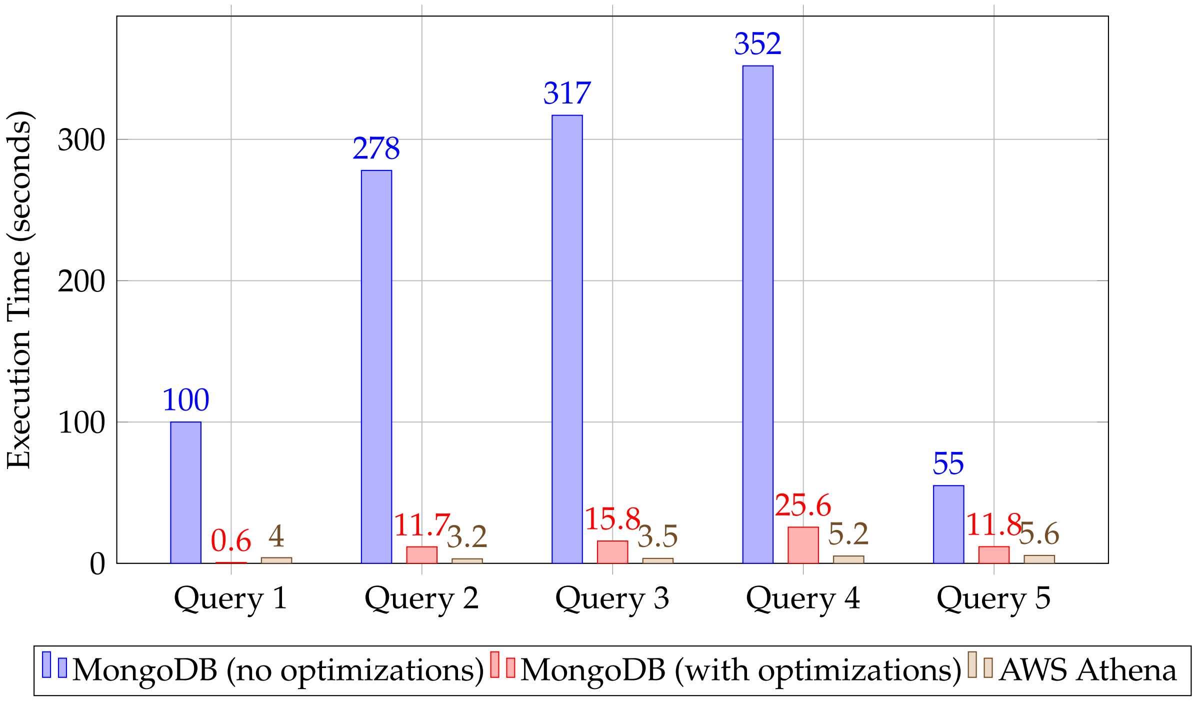

Figure 4.5: Query Execution Time on AWS Athena and MongoDB (optimized and not optimized) for `end_of_prog_aggregated` table

The results in Figure 4.5 clearly show the significant performance improvement with optimizations in MongoDB. In the "no optimizations" case, MongoDB's query times are significantly higher, as it must read both the data and indexes from disk. With optimizations enabled, the query times decrease dramatically, as both the data and indexes are cached in memory. However, when comparing MongoDB with AWS Athena, it is evident that Athena consistently outperforms MongoDB, even with all optimizations applied.

AWS Athena achieves execution times that are 10 to 70 times faster than MongoDB in its non-optimized state, which is the typical case in most business intelligence analyses, as queries are often run on-demand or with low frequency, preventing data and indexes from remaining in memory.

In conclusion, while MongoDB, with proper optimizations, can deliver competitive performance for smaller datasets, Athena outperforms MongoDB in large-scale data processing scenarios, especially when query performance and scalability are crucial.

4.4 Cost Estimation

Once the system had been tested and validated, it was essential to estimate the costs of each AWS service utilized. This estimation provides the company with an approximate understanding of the total solution cost. For simplicity, costs were divided into two phases: the bulk load and operational phases. The bulk load represents a one-time expense, while operational costs recur monthly and depend on user activity.

4.4.1 Bulk Load Phase Cost Summary

The following table summarizes the estimated costs for the bulk load phase:

AWS Service	Feature	Description	Cost ($)
Lambda	Elaboration	((46,000 requests × 450 s × 6 GB) - 400,000 GB/s) × 0.0000166667 USD	2,063.34
Glue	ETL mongo	5,000 DPU × 0.42 hours × 0.29 USD	604.18
	ETL postgres	20 DPU × 0.42 hours × 0.29 USD	2.43
S3 (Raw)	Storage	2,400 GB × 0.0245 USD	58.80
	PUT requests	3,000,000 requests × 0.0000054 USD	16.20
S3 (Curated)	Storage	300 GB × 0.0245 USD	7.35
	PUT requests	1,000,000 requests × 0.0000054 USD	5.40
S3 (Analytics)	Storage	34 GB × 0.0245 USD	0.83
	PUT requests	1,000,000 requests × 0.0000054 USD	5.40
RDS	Instance	730 hours × 0.074 USD	54.02
	Storage	50 GB × 0.137 USD	6.85
Total			**2,824.80**

Table 4.1: AWS Service Costs for the Bulk Load Phase

In this one-time phase, costs were incurred primarily by Lambda, Glue, S3, and RDS. Approximately 46,000 Lambda invocations were required to process the initial data load, involving complex orchestration between master and worker functions, each with an estimated average runtime of 7 minutes and memory usage of 6 GB. Glue costs stemmed mainly from the ETL processes handling data transformation, especially for MongoDB, which demanded approximately 20 Data Processing Units (DPU) per function. The RDS instance, slightly overprovisioned for initial testing, could potentially be downsized to reduce costs. Costs for GET requests in S3 were negligible and thus omitted.

4.4.2 Operational Phase Monthly Costs

The table below shows the monthly costs for the system's operational phase:

AWS Service	Feature	Description	Cost ($)
Lambda	Elaboration	((1,217 requests × 720 s × 6 GB) - 400,000 GB/s) × 0.0000166667 USD	80.93
Glue	ETL mongo curated	600 DPU × 0.42 hours × 0.29 USD	73.08
	ETL postgres	600 DPU × 0.1 hours × 0.29 USD	17.40
	ETL analytics	600 DPU × 0.05 hours × 0.29 USD	8.70
S3 (Raw)	Storage	2,400 GB × 0.0245 USD	58.80
	PUT requests	1,500,000 requests × 0.0000054 USD	8.10
S3 (Curated)	Storage	300 GB × 0.0245 USD	7.35
	PUT requests	1,500,000 requests × 0.0000054 USD	8.10
S3 (Analytics)	Storage	34 GB × 0.0245 USD	0.83
	PUT requests	1,000,000 requests × 0.0000054 USD	5.40
Athena	Queries	1,521 queries × 0.009765625 TB × 5.00 USD	74.27
QuickSight	Readers	10 Readers × 3.00 USD	30.00
	Authors	1 Author × 24.00 USD	24.00
	SPICE	(100 GB - (10.00 GB × 1 author)) × 0.38 USD	34.20
Total			**431.16**

Table 4.2: AWS Service Costs for the Operational Phase (Monthly)

These monthly operational costs are estimated based on an expected daily load of 40 Lambda invocations with a runtime average of 10 minutes each. Glue requires 20 DPUs daily for ETL processes. Athena costs vary depending on usage but are estimated with a baseline of 50 queries per day, scanning approximately 10 GB of data each. QuickSight includes 10 reader accounts and 1 author account, along with 100 GB of SPICE storage. The estimate does not include the use of generative AI with Quicksight Q, which would require an increase of at least $275 per month. RDS costs are negligible here, as daily upserts are minimal and could be handled by an RDS instance with lower performance or by leveraging free AWS options. In summary, the system has an initial cost of approximately 2800 USD for the *bulk load* phase and for the ongoing *operational* phase, the estimated monthly cost is around 430 USD. Operational expenses vary a lot based on actual system utilization by users.

5

Conclusions and Future Works

This thesis presents the design and development of a data lake and business intelligence solution on AWS, specifically tailored to improve data accessibility, integration, and analysis within UNOX S.p.A. Through leveraging AWS services such as S3, Glue, Athena, and QuickSight, the project succeeded in creating a robust architecture that automates data ingestion, transformation, and storage, thus enabling near real-time access to data analytics across multiple departments.

The system introduced not only provides a centralized data repository but also empowers non-technical users with tools to perform independent analyses. By implementing a data lake over a traditional data warehouse, UNOX gains flexibility in handling diverse data sources, supporting both structured and unstructured data formats. Furthermore, the architecture's reliance on serverless components, such as AWS Glue, has minimized infrastructure management efforts and optimized the system's scalability and cost-efficiency.

Performance tests confirm the solution's reliability and responsiveness, significantly reducing the time required for data queries and report generation. Financial analyses also indicate that the system is sustainable for long-term operation, with manageable monthly costs.

Despite the positive results achieved, several areas for improvement emerged during the implementation, particularly regarding the initial MongoDB data ingestion process. This bulk loading phase was identified as the most resource-intensive part of the pipeline in terms of both time and cost. The initial ingestion from MongoDB is markedly slow, taking several weeks to complete and incurring high costs relative to the workload managed. This inefficiency is primarily due to the long response times for MongoDB queries, which can take several minutes to retrieve IoT data generated over two months for a single device.

One approach to mitigate these costs is to dynamically adjust the memory allocated to each Lambda worker, based on the volume of data extracted. Currently, memory allocation remains static and relatively high to prevent overload issues, but refining this setting based on workload requirements could offer a slight reduction in cost.

An alternative to the existing ingestion method could involve employing a dedicated migration service, such as AWS DMS, to transfer the full historical database into Amazon S3. AWS DMS could handle bulk data migrations more efficiently, potentially reducing both the time and cost of initial data ingestion.

Regarding the operational phase, optimizations could be achieved by implementing a CDC or a streaming data transfer system. In this project context, CDC would continuously track and propagate changes from the source databases to the data lake, ensuring that only new or updated data is transferred, which would streamline data processing and reduce load times. A streaming transfer system would enable real-time data ingestion by pushing data as it is created, ideal for rapid data analysis and time-sensitive reporting.

From a storage perspective, further cost reductions could be achieved by categorizing data into different storage classes based on usage and tagging files when they are created in the data lake. For instance, Amazon S3's Intelligent-Tiering could automatically move data between storage classes based on access patterns, minimizing costs without manual intervention [2].

Additionally, exploring advanced machine learning integrations, especially for predictive analytics, could add further value to the BI platform.

Overall, the project represents a substantial step forward in the digital transformation of UNOX's data management capabilities, fostering a data-driven culture and enhancing decision-making processes across the organization.

References

[1] Michael Armbrust et al. “Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics”. In: *Proceedings of CIDR*. Vol. 8. 2021, p. 28.

[2] Prashant Singh Avijit Goswami and Rajarshi Sarkar. *Improve operational efficiencies of Apache Iceberg tables built on Amazon S3 data lakes*. 2023. URL: https://aws.amazon.com/it/blogs/big-data/improve-operational-efficiencies-of-apache-iceberg-tables-built-on-amazon-s3-data-lakes/.

[3] AWS. *AWS Well-Architected Framework*. 2024. URL: https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html.

[4] AWS. *Functions by category*. 2024. URL: https://docs.aws.amazon.com/quicksight/latest/user/functions-by-category.html.

[5] Vladimir Belov, Andrey Tatarintsev, and Evgeny Nikulchev. “Choosing a Data Storage Format in the Apache Hadoop System Based on Experimental Evaluation Using Apache Spark”. In: *Symmetry* 13.2 (2021). ISSN: 2073-8994. DOI: 10.3390/sym13020195. URL: https://www.mdpi.com/2073-8994/13/2/195.

[6] Swapnil Bhoite. *AWS Glue DynamicFrame transformations with example code and output*. 2022. URL: https://swapnil-bhoite.medium.com/aws-glue-dynamicframe-transformations-with-example-code-and-output-26e14d13145f.

[7] Rupali Chopade and Vinod Pachghare. “MongoDB indexing for performance improvement”. In: *ICT Systems and Sustainability: Proceedings of ICT4SD 2019, Volume 1*. Springer. 2020, pp. 529–539.

[8] Don Drake. *Benchmarking Apache Parquet: The Allstate Experience*. 2016. URL: https://blog.cloudera.com/benchmarking-apache-parquet-the-allstate-experience/.

[9] Tomislav Hlupi et al. "An overview of current data lake architecture models". In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1082–1087.

[10] Kushwanth Chowdary Kandala. *Change Data Capture(CDC) using AWS Glue in Batch ETL Process*. 2023. URL: https://www.linkedin.com/pulse/change-data-capturecdc-using-aws-glue-batch-etl-process-kandala-7dejc/.

[11] Athira Nambiar and Divyansh Mundra. "An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management". In: *Big Data and Cognitive Computing* 6.4 (2022). ISSN: 2504-2289. DOI: 10.3390/bdcc6040132. URL: https://www.mdpi.com/2504-2289/6/4/132.

[12] Athira Nambiar and Divyansh Mundra. "An overview of data warehouse and data lake in modern enterprise data management". In: *Big data and cognitive computing* 6.4 (2022), p. 132.

[13] Sivakumar Ponnusamy and Pankaj Gupta. "Scalable Data Partitioning Techniques for Distributed Data Processing in Cloud Environments: A Review". In: *IEEE Access* 12 (2024), pp. 26735–26746. DOI: 10.1109/ACCESS.2024.3365810.

[14] Srinivas Rishindra Pothireddi. *Optimization Strategies for Iceberg Tables*. 2024. URL: https://blog.cloudera.com/optimization-strategies-for-iceberg-tables/.

[15] Rajesh Remala, Krishnamurty Raju Mudunuru, and Sevinthi Kali Sankar Nagarajan. "Optimizing Data Ingestion Processes using a Serverless Framework on Amazon Web Services". In: *International Journal of Computer Trends and Technology* 72 (). DOI: 10.14445/22312803/IJCTT-V72I6P116.

[16] Jeffrey Richman. *Data Lake Architecture: Components, Diagrams, & Layers*. 2023. URL: https://estuary.dev/data-lake-architecture/.

[17] Roy Yung and Koushik Muthanna Koravanda Ganapathy. *Best practices for Amazon QuickSight SPICE and direct query mode*. 2024. URL: https://aws.amazon.com/it/blogs/business-intelligence/best-practices-for-amazon-quicksight-spice-and-direct-query-mode/.

Acknowledgments