POLITECNICO DI TORINO

MASTER's Degree in DATA SCIENCE AND ENGINEERING

MASTER's Degree Thesis

A configurable data platform for streaming delta and full data ingestion

Supervisor **Candidate**

Prof. PAOLO GARZA **MICHELE GALLINA**

2023/2024

Summary

The thesis project focuses on creating a cloud-based platform to manage large amounts of data in a secure, efficient, and dynamic way to meet current and future needs. The work was carried out using Apache Spark within Databricks and analyzing which framework best suited the various requirements. The platform is entirely cloud-based. Specifically, it was used Microsoft Azure. Being built using a cloud service allows for easy scaling, both up and down, to quickly respond to changes in data volume or adjust the processing time required. The use of Databricks provides a highly versatile platform based on Apache Spark, natively integrated with many frameworks to enable the creation of a system capable of meeting needs ranging from data ingestion and processing to the creation of complex dashboards and even the use of AI models.
The thesis work, in particular, focused on creating a data platform for a security company to ingest data from two sources: a relational database and an IoT sensors network. Once the data are stored on the platform, they undergo a quality improvement process to be made available to meet business needs. The platform was also designed to be as configurable as possible to make it easily extensible. Three company requirements were selected on the business side, and a solution was proposed for each.

Acknowledgements

Prima di procedere con la trattazione, vorrei dedicare qualche riga a tutti coloro che mi sono stati vicini in questo percorso di crescita personale e professionale.

Un sentito grazie al mio relatore Prof. Paolo Garza per la sua disponibilità e tempestiva risposta ad ogni mia richiesta.

Un ringraziamento speciale va al mio Tutor Andrea Settimo che mi ha proposto il progetto su cui lavorare, guidato nelle ricerche ed è sempre stato disponibile per rispondere a ogni mia richiesta o chiarimento.

Non posso non ringraziare l'azienda Cluster Data Reply, che mi ha dato la possibilità di effettuare questo lavoro e mi ha accolto nel suo team in questi mesi, in particolare vorrei ringraziare Massimiliano Rizzi.

Ovviamente uno speciale ringraziamento va ai miei genitori che mi hanno sempre supportato (e sopportato) in tutti gli alti e bassi del percorso accademico, e con loro tutti i miei famigliari.

Un grazie speciale va anche a tutti i miei amici e in particolare a Filippo, Paolo, Christian, Sofia, Andrea, Francesca, Mattia e Gianluca che in questi anni mi sono stati vicini.

Un grande grazie a Elena che in questo ultimo anno mi è sempre stata accanto e mi ha supportato in questo ultimo periodo del mio percorso accademico.

Come potrei non ringraziare i membri del Gruppo Speleologico Mineralogico Valsesiano con cui ho condiviso molti momenti di svago e che hanno programmato numerose attività in funzione dei miei studi.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

API

Application Programming Interface

DLT

Delta Live Tables

IoT

Internet of Things

REST

REpresentational State Transfer

RU/s

Request Units per second

SCD

Slowly Changing Dimension

SQL

Structured Query Language

SSP

Structured Streaming Programming

Chapter 1

Introduction

Data has become a highly valuable asset for companies, institutions, and organizations. In many cases, important decisions are made based heavily on data. Collecting and using data often enables a company to maintain or obtain a competitive advantage by introducing new technological solutions and optimizing internal processes. For example institutions can utilize data to offer personalized services to citizens and to create digital systems capable of quickly identifying those who do not comply with the law.

It is important to ensure high-quality and valuable data because all data-driven services require these features; to achieve this, it is essential to use robust and flexible data management platforms capable of ingesting data from multiple sources, processing it, and making it available in the most suitable form for each specific use. In many cases, a lot of data are involved, so these platforms need to manage large amounts of data efficiently. Moreover, the data platforms should allow activities to be easily scalable. Without this infrastructure, also the most promising data source can become unusable, leading to lost opportunities or costly errors. Machine learning systems are an example of data-driven services. These systems require properly managed databases. In a nutshell, without data management platforms, all services based on artificial intelligence would not be feasible.

The thesis worked on designing and deploying a cloud-based data platform using the most advanced technologies for processing large data volumes in enterprise environments. They provide scalability, security, and high performance, enabling data ingestion from more sources. In other words, these technologies guarantee flexibility and adaptability in the management of complex data ecosystems. Specifically, the platform can ingest data from two different sources. The first is a relational database: a centralized source

of structured data. The second is a network of IoT sensors: a distributed source of semi-structured data.
The entire platform was developed using a managed cloud service to ensure quick and easy adaptation to possible changes in requirements and data volume in game. Moreover, the managed cloud removes the need to handle the physical infrastructure. The design was focused particularly on configurability and modularity to allow for easy extension of the platform capabilities while ensuring maintainable code. In other words, a source code with customizable behavior at runtime was extensively utilized together with primitive classes used to inherit methods and parameters. It was also used in association with classes that define methods employed in various parts of the platform where it was not convenient to exploit inheritance.

1.1 Platform context

The platform developed during this thesis work is a prototype of a data management system for a security company. The company provides various services, including alarm systems, surveillance, cash-in-transit, and others. To make the project feasible, and given its demonstrative nature, the developed platform manages only a subset of all company data. This approach allowed for an accurate exploration of available frameworks to select those best suited to the project's objectives.

The main objective of the prototype was to demonstrate the feasibility of having a single system that integrates multiple aspects of data management, from data ingestion and processing, to meet specific business needs, including data archiving with different levels of quality.

Three distinct business requirements were selected: making aggregated data available, providing a pseudo real time notification service, and creating interactive dashboards. The data analysis activities were minimal, as they were not among the main objectives.

Given the prototype's demonstrative nature, synthetic data were used to ensure clients' privacy and comply with data protection laws. Synthetic data was utilized in a way that reliably simulates real-world scenarios for which the platform was designed, while ensuring high privacy standards during the development and demonstration phases.

Chapter 2

Technologies

To build a cloud-based data platform, it is required to use different technologies that work at different platform layers. This chapter explains all the technologies used; chapter 3 explains how they work and interact in the specific case of thesis work.

2.1 Cloud Platform: Microsoft Azure

Microsoft Azure[1] is a cloud computing platform developed by Microsoft designed to build, manage, and deploy applications across a global network of data centers. It offers various services, from basic computing, storage, and networking to more advanced technologies such as artificial intelligence, machine learning, and internet of things. The core idea behind Azure is to allow companies to access high-performance computing resources without needing expensive physical infrastructures, letting them to quickly scale up or down based on their specific needs. In other words, users can rent computing resources over the Internet rather than relying on their on-premises hardware. This flexibility is a significant advantage, as it enables companies to reduce costs while maintaining the ability to adjust their computing power in response to changing business demands. This aspect was essential in the thesis because it enabled the construction and testing of the infrastructure using low-cost and low-performance resources. On the other hand, the high-performance resources were allocated only to benchmark the platform's performance when it ingests and manages high data volume. With this approach, the building costs were significantly reduced, compared to always utilizing resources capable of managing the platform's operational data flow. Furthermore, it was possible to quickly assess the impact of varying the

computational resources on the platform's overall performance.

Microsoft Azure's services are categorized into different types, known as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). With IaaS, companies can rent virtual machines, storage, and networking components, giving them complete control over the infrastructure. PaaS, on the other hand, provides a more streamlined experience, offering pre-configured frameworks that help developers to build and deploy applications faster. SaaS includes ready-made software solutions that run on Azure's infrastructure, allowing companies to use fully managed applications without worrying about construction and maintenance. For the project, a mixture of the two primary types of service, Infrastructure as a Service and Platform as a Service, was used to combine the necessity for complete control over the cloud resources and the use of pre-configured services.

Another important aspect to consider is the security of the infrastructure. Microsoft Azure makes a strong effort to protect data. It offers built-in tools for protecting data, managing user identities, and detecting potential threats. The platform also complies with international standards, which are crucial for organizations that operate in highly regulated sectors. These robust security measures ensure that companies can operate in the cloud confidently because they know that their sensitive information are protected. In the context of the thesis, the platform manages personal data of employees and clients, as well as other non-personal yet equally sensitive information, such as the alarm system codes. So, it was important to build the data platform on an infrastructure that guarantees data security by default.

To conclude, two additional Azure features are briefly outlined. They are not directly relevant to the thesis project but could potentially become significant in a broader context of project generalization to address other business needs: the global level of Azure and the hybrid cloud to manage very sensible data efficiently.
Azure works globally; Microsoft has established over 60 Azure regions across the globe, which gives the possibility to run the applications and store data close to the users. So, the geographical distribution ensures fast performance, low latency, and reliable service continuity, even if one region incurs issues. These are very important features for a company that works on an international level.
Azure also supports hybrid cloud environments, allowing organizations to combine their on-premise infrastructure with cloud resources. This feature is particularly useful for organizations that handle sensitive and strategic

data alongside tasks involving less critical information because it allows for differentiated data management based on processing risks.

2.2 Azure Technologies

The section explains all the Microsoft Azure cloud technologies and components required and used to build the data platform.

2.2.1 Virtual Network

Azure Virtual Network (VNet) [2] is a fundamental service to build Microsoft Azure's cloud infrastructure; it plays a central role in providing connectivity and security. It replicates the functionality of traditional on-premises networks in a virtualized context and offers the developer control over their networking environment in the cloud. VNet enables the creation of private, isolated networks where Azure resources can reside and communicate securely with each other, and with on-premises systems, or external services. The architecture of VNet ensures both secure communication and robust integration with existing IT infrastructures. One of the core functions of Azure VNet is its ability to isolate and segment resources into subnets. Subnets are essential for organizing resources logically and controlling traffic flow within the network. Organizations can improve security and traffic management by segmenting a VNet into multiple subnets. Each sub-network can be assigned a range of IP addresses and access policies can be defined; this ensures that communication between the different segments is regulated and each entity can only communicate with the minimum number of entities required to function properly.
So, the VNet component is fundamental to building a cloud-based data platform because it efficiently enables secure communication between different components and the ability to provide secure access to data by external business applications. It ensures that each platform resource can be accessed only by legitimate actors, providing a robust and effective security system.

2.2.2 Storage Account

The Azure Storage Account [3] component provides scalable, secure and highly available storage for various data categories. It allows users to store a wide range of data objects efficiently. Azure Storage Account supports different storage types: Blobs, Queues, and Tables, providing flexibility for

various storage needs. In the context of the thesis project, the type Blob with hierarchical names for the objects was used.

Blob storage, also known as Binary Large Objects (Blob), is used to store unstructured data. The Blob Container is a component of the Blob service, acting as a logical grouping for blobs (files) within a storage account. Each container can store unlimited blobs and is designed to organize and manage data. A container provides a versatile and scalable storage solution. It offers various features that can be configured to better meet user needs. The two most important features to configure for a container are outlined below.

- **Blob Storage Type** [4]: Depending on the usage, there are three types of Blob storage available. Block Blobs are the most commonly used, ideal for storing binary files or text and for uploading large files efficiently. Append Blobs are optimized for append operations, so they are helpful when data needs to be appended rather than overwritten; an example is a logging scenario. Finally, page Blobs are designed for virtual machine disks because they are optimized for fast random read and write operations.

- **Blob Access Tiers** [5]: Blob containers support multiple access tiers, which allow the optimization of costs and performance based on data access patterns. The Hot tier is for frequently accessed data, ensuring quick retrieval. The Cool tier is designed for infrequently accessed data and balances cost with retrieval latency. The Archive tier is designed for long-term storage with rare data access. It offers the lowest cost but with the longest retrieval times. Also, hours can be needed to get the first byte.

The storage type and access tiers chosen for the project will be explained later because they require considering the behavior of frameworks not yet introduced.

Another important storage configuration is data redundancy. Azure storage provides multiple options for data redundancy, ensuring high availability and durability of stored data. Locally Redundant Storage (LRS) replicates data within a single region, while Geo-Redundant Storage (GRS) ensures data replication across different regions, safeguarding against regional failures. Zone-Redundant Storage (ZRS) replicates data across availability zones within a region, providing enhanced resilience. The decision on the type of redundancy chosen must be made considering the consequences of losing data and the costs involved in redundancy. It is not a strictly technical decision, so this aspect was not considered in the thesis due to the prototype nature of the data platform. Therefore, the less expensive solution was chosen.

Before concluding the overview of storage, it is important to consider the security aspects. Like all the other Azure components, access policies can be configured at different levels of the Azure Storage to ensure that only legitimate entities can access the data with the minimum level of permissions required for proper operation. The other important security aspect concerns data encryption. Azure Storage employs an encryption mechanism using 256-bit Advanced Encryption Standard (AES) offering high levels of data protection [6]. The encryption process operates transparently, meaning data are encrypted and decrypted automatically without user intervention. The encryption is enabled by default for all types of storage accounts and cannot be disabled, guaranteeing that data are always secured. Therefore, the secure nature of Azure Storage ensures that data are always encrypted from the moment they are stored. Moreover, encryption does not entail any additional economic costs. By default, new storage accounts are encrypted using Microsoft-managed keys, a feature that provides immediate and reliable protection. However, users can also manage encryption keys if they prefer to retain full control over encryption processes.
In the thesis project, it was decided to utilize Microsoft-managed keys to streamline the encryption management process, considering the prototype nature of the work. Self-managed encryption keys could be a potential option to improve security in a production environment, but they must be managed correctly.

2.2.3 Cosmos DB

Azure Cosmos DB [7][8] is a highly configurable, globally distributed database service that ensures low-latency performance, scalability, and data consistency models. It is a versatile option for a wide range of applications, from real-time analytics to IoT, thanks to its support for numerous data models and APIs, which ensure flexibility.

Azure Cosmos DB is provided as a fully managed service of Microsoft Azure. It removes the burden of database management, updates, and patching operations. Its serverless nature and automatic scaling up and down ensure that it always addresses your needs, providing cost-efficiency and peace of mind.

Another characteristic of Cosmos is its global availability; it is built as a globally distributed architecture that uses Azure datacenters, which allows data replication to maintain high performances regardless of geographical location. Different replication models are available to satisfy various users'

needs. The geographical replication feature is unnecessary for the thesis work because the company works only in Europe, so West Europe was chosen as the location.

Cosmos DB provides three different classes of databases: NoSQL, relational, and vector. A NoSQL document-based database was needed for the thesis work. Cosmos also offers other types of NoSQL databases through API, but they will not be explained in detail. The Cosmos documents are JSON-like documents and support the JSON data types. The documents are basic units of information. A document stores data in a semi-structured format consisting of key-value pairs. As a general best practice, each document should contain information about an existing real-world entity. The documents are organized in collections (also called containers; the name depends on which API you use), and each collection should contain related documents. A collection is similar to a table in a relational database but does not enforce a schema, allowing for a flexible document structure. The collections are grouped in databases, the less granular Cosmos entity. Also, if each document can have a different schema, to ensure high performance and easy internal management, the documents have a bunch of mandatory fields [9]:

- id: The unique identifier for the document within a logical partition (The logical partitions are explained later); it can not exceed 255 characters and helps with the lookups.
- _rid: The unique hierarchical identifier for the resource stack in the resource model. It is utilized internally for positioning and navigation, and it is automatically generated by the system.
- _ts: The timestamp of the last document update, it is automatically generated by the system.
- _self: The unique and addressable URI of the document, it is automatically generated by the system.
- _etag: It is a tag for ensuring optimistic concurrency control and is automatically generated by the system.
- _attachments: It specifies an addressable path for the attachments resource and is automatically generated by the system.

To complete the organization of documents, it is important to explain the concept of partition [10]. Two types of partitions exist: logical and physical. Azure Cosmos DB uses both to scale individual containers horizontally. The

logical partitions are the subdivisions of documents inside a container; they are built based on the distinct values of the document partition keys. Thus, all the documents belonging to a partition share the same partition key value. The physical partitions are the physical division of the documents of a container and are fully managed internally by Cosmos. In simple words, one or more logical partitions are mapped to a single physical partition; in this way, Cosmos automatically manages the scale-up of a container. Oblivious to ensure an efficient scale-up, the users need to pay attention to the choice of the partition key field used in each container. The partition key field role can be assigned to any field (also nested) in the documents. It is decided when the container is created; the designed partition key field must exist in all documents of the container and can not be changed. The only way to change the partition key field is to create a new container, assign the new partition key, and transfer the data from the old container.

Cosmos DB supports different types of indexes, and by default, it automatically indexes every document field for all items in the container without defining any schema or configuring indexes [11]. It is possible to do it efficiently because Cosmos transforms every document into a tree representation. The default indexing feature plays an important role during the document design phase.

Lastly, an important Cosmos DB feature for the thesis work is how the document can be accessed and managed. Cosmos DB allows data to be interacted with many different APIs [12].

- API for NoSQL: The native API for NoSQL data document format. It gives the best end-to-end experience with the latest features available.
- REST API: It provides access through REST protocol over HTTPS using SQL queries.
- API for PostgreSQL: is a managed service for running PostgreSQL.
- API for MongoDB
- API for Apache Cassandra
- API for Apache Gremlin
- API for Table

The availability of all the above APIs ensures the possibility of opening to open-source ecosystems and allows the use of already developed and mature

skills. The reason for which it is crucial in the thesis work will be clear in Chapter 3.

2.2.4 Event Hubs

Azure Event Hub [13] is a cloud-native data-streaming solution capable of streaming millions of events per second with low latency from any source to any destination and decoupling event producers from event consumers. It is available on Azure as a platform-as-a-service solution that developers can use to ingest and store streaming data or as a notification system for external services that need to process data as soon as it is available.

Event Hubs support natively Advanced Message Queuing Protocol (AMQP), Apache Kafka, and HTTPS protocols; the latter can only be used to write data to an event hub. SDKs are also available for Python, .NET, Java, and JavaScript, providing low-level integration to ensure high performance and efficiency.

A best practice is that each Event Hub represents a specific data stream about a topic. Multiple Event Hubs can be organized within an Event Hub Namespace, which serves as a container. The Event Hub Namespace provides common configuration options, such as DNS-integrated network endpoints, access control, network integration management, geo-disaster recovery, and common access key settings across all the Event Hubs in the Namespace, allowing easy management.

Similarly to Cosmos DB, the Event Hubs use a partition key field to scale up; the main difference respect to Cosmos is that the scaling is not self-managed but should be managed by the developers.

An architecture that uses Event Hub is composed by [14]:

- Producer applications: They produce and write the data over the Event Hubs.
- Namespace: It is the management container for one or more Event Hubs.
- Event Hubs: They allow the organization of the event. Each Event hub can be seen as an append-only distributed log that can include one or more partitions.
- Partitions: They’re used to scale an event hub. They function similarly to motorway lanes. It is possible to add more partitions if more streaming throughput is required.

- Consumer applications: They are the applications that consume the data written into the Event Hubs

The Event Hubs also have checkpoints to prevent customer applications from reading the same data multiple times. Moreover, the data written over the event hub are not permanent like the ones written on the storage but have a retention time after they are deleted.

The usage of the Event Hub in the thesis work will be explained in Chapters 3 and 5.

2.2.5 Managed identities for Azure resources

In a data cloud platform that uses several resources, a challenge involves managing the credentials, keys, secrets, and certificates to allow secure communication between the platform components. Managed identities for Azure resources [15] removes the need to manage these credentials. It provides an automatically managed identity in Microsoft Entra ID for applications that support Microsoft Entra authentication. The applications use managed identities to obtain Microsoft Entra tokens without managing any credentials. Of course, it is possible to manage the credential classically in the cases in which something does not support Microsoft Entra ID.

There exist two types of managed identities: system-assigned and user-assigned. The system-assigned identities can be used for the Azure resources that allow to enable a managed identity directly on the resource. In these cases, a special type of service principal is created with the same name and life cycle as the resource. On the other hand, with user-assigned managed identities, the developers can create a managed identity as standalone Azure resources. Each of them can be assigned to one or more Azure resources. In these cases, a particular type of service principal is created and managed separately from the resources that use it. So, the life cycle of the identity is separate from the resources that use it and must be manually deleted when it is no longer used.

Because the *Managed identities for Azure resources* is integrated with all the components of the thesis project and ensures high-security requirements, it was used in all situations requiring an authentication system. Moreover, system-assigned identities were used.

2.2.6 Key Vault

The Key Vault [16] is the Azure component designed for security storing secrets. A secret is anything that you want to control access to tightly, such as passwords, certificates, and API keys. So, the Key Vault provides an easy service to use that allows access to the secrets only to legitimate entities. It is obvious that the authentication process into the Key Vault is very important. It can be done in three ways:

- Managed identities for Azure resources.
- Service principal and certificate.
- Service principal and secret.

Managed identities for Azure resources authentication service was used to manage the authentication in to the Key Vault because it provides high security in combination with an easy usages, and it is compatible with the resources used.

2.3 Databricks

Databricks [17] [18] is a unified cloud-based data platform for big data analytics, data engineering, machine learning, and data visualization. The developer of Apache Spark created Databricks to provide a robust environment for big data processing and storage. Moreover, it automatically manages the infrastructure maintenance. So, in simple words, Databricks provides an environment that simplifies data workflows and allows easy collaboration between data engineers, data scientists, and data analysts.

Databricks is fully integrated with cloud services such as Microsoft Azure, Amazon Web Services, and Google Cloud. It is also compatible with various analytics tools and data storage. So Databricks is not only a robust environment in which to perform big data tasks but is also a very flexible platform that addresses very different user needs.

Databricks also allows the usage of skills already developed and mature; it supports four programming languages: Python, SQL, R, and Scala. The platform supports notebooks that combine code in different languages with visualizations and text. The Notebooks enable team members to collaborate and work in real time.

The following sections explain the technologies used over Databricks in the thesis project.

2.3.1 Apache Spark

The elaboration core of Databricks is built on Apache Spark [19]. It is an open-source framework for Big Data that efficiently processes massive amounts of data and eliminates low-processes management such as synchronization and orchestration. The idea behind Spark is to use disk storage as little as possible and process the data in memory until feasible. So, the computation speeds up with respect to the previous frameworks, such as Map-Reduce for Hadoop.

Spark is a framework that offers multiple data processing activities, such as batch processing, real-time streaming, machine learning, and graph analytics. Since the framework was born for big data processing, all the activities were tailored to be executed as much as possible in a parallel way. The functionalities of the Spark Core Component can be used through an application programming interface based on the Resilient Distributed Dataset (RDD) data abstraction. On top of the Spark Core Component, the Spark SQL [20] Component is built. It supports structured data processing by introducing the abstraction data type called DataFrame and a specific language (DSL) to manipulate DataFrames in different programming languages. For the thesis project, the SQL component of Spark was used via the Python programming interface.

Spark also supports streaming analysis with different features for RDD and DataFrames. The following section explains the Spark Structured Streaming Programming framework, which enables streaming analysis of structured data and uses DataFrames.

2.3.2 Spark Structured Streaming Programming

Spark Structured Streaming Programming [21] is fully integrated into Databricks; it is built over the Spark SQL component and uses the Spark SQL engine. The Spark Structured Streaming Programming framework provides scalable and fault-tolerant stream processing; the streaming computation can be expressed in the same way as the batch computation done over static data. The Spark SQL engine performs the streaming computation incrementally, ensuring end-to-end exactly-once fault tolerance through checkpointing and write-head logs.

With the Spark Structured Streaming Programming, it is possible to choose between micro-batch and continuous query processing methods. Micro-batch is the default one, and with it, the queries are processed utilizing micro-batch, ensuring latencies at most 100 milliseconds and it is guaranteed exactly

once fault-tolerance. On the other hand, continuous processing ensures 1 millisecond as latency but guarantees only at-least-once processing. Also, in this case, the data operations are expressed in the same way, allowing developers to choose easily the processing method based on the application requirements. In the thesis project, the micro-batch processing method was selected.

Spark Structured Streaming Programming also allows interaction with different program languages; Python was always chosen for the project.

2.3.3 Databricks Account and Workspace

The Databricks Account is the top hierarchical entity that manages access to Databricks in the cloud.
A Databricks workspace is the deployment in the cloud of Databricks. It is an implementation of Databricks in the cloud that provides access to Databricks resources. A Databricks account can be associated with one or more Databricks workspaces, depending on the organization's needs [18].

2.3.4 Unity Catalog: The Centralized Data Governance

Unity Catalog [22] [23] is developed and integrated into Databricks. It is an open-source data governance solution. It is designed to provide both centralized access control and data discovery across Databricks workspaces.

In the Unity Catalog, the Metasore records metadata about the data assets and regulates access to them. The Metastore is the Unity Catalog's top-level metadata container. A Workspace has one Metastore for each geographical region. The database objects in the Metascore are accessible through three levels of hierarchy: catalog, schema, and object. Keep attention that the catalog level is not the Unity Catalog. Not only database objects but also other things are registered in the Metastore, but they will not be cited in this work.
The Catalogs [24] are the basic level of the Unity Catalog data governance model. They are used at the top level of the data isolation scheme.
The Schemas [25], also known as Databases, are the second level of the data governance model in Unity Catalog. They contain the database objects.
At the third level, there are the database objects.

- Tables: The data are organized into rows and columns. A table can be managed or external. For a managed table, the whole table lifecycle is managed by the Unity Catalog. For an external table, Unity Catalog does

not handle access to the table's storage location made externally from Databricks but only handles access to the table made within Databricks.

- Volume: Volumes logically organize unstructured and non-tabular data into cloud storage. Also, a volume can be managed or external; the implications are the same as the tables.
- View: The views are queries text from one more data source of the Metastore. The result of a view is computed in runtime when it is queried and is not stored in the Metastore.
- Functions: The functions are saved logic that returns a set of rows or a value.
- Model: The Models are AI models registered in the unity catalog as functions.

The storage location of the managed database objects can be defined at each level of the hierarchy. If a storage location is not specified for a level, it inherits it from the upper level.

2.3.5 Lakehouse and Delta Lake

The tables on Databricks are built on top of Delta Lake [26] [27], an optimized storage layer. It is open-source software that adds file-based transaction logs for ACID transactions and scalable metadata management to Parquet data files. With Delta Lake, a single copy of data may be used for batch and streaming operations thanks to the full compatibility with Apache Spark APIs. The standard format for all Databricks operations is Delta Lake. All tables on Databricks are Delta Tables if there are no other specifications. The Delta Lake provides some important features to the tables

- ACID Transactions: Delta Lake provides Atomicity, Consistency, Isolation, and Durability (ACID) properties. In this way, it ensures data integrity also in case of failures and concurrent data operations.
- Scalable Metadata: Delta Lake makes it easy to manage petabyte-scale tables with billions of files and partitions.
- Time Travel and Audit History: Delta Lake supports time traveling, which allows access and restoration of previous versions of data. It is possible because Delta Lake maintains a complete audit trail by logging every modification.

- Schema Enforcement and Evolution: Delta Lake enforces a schema on the table to prevent bad data from causing data corruption. At the same time, Delta Lake also supports schema evolution, which allows changes to the schema without failures. Of course, schema evolution should be enabled only for tables that need it.
- Efficient Data Storage and Access: Delta Lake improves storage efficiency by compacting small files into larger ones. It reduces the number of files and improves read and write operations.

2.3.6 Workflows and Jobs

Workflows [**Databricks:Workflows**] is an orchestration Databricks tool. It is used to configure Databricks jobs. The jobs are the first layer of the orchestration mechanism. Each job comprises one or more tasks and manages dependencies between the tasks. A task represents a unit of logic and can consist of a notebook, a JAR, SQL queries, a Delta Live Table pipeline (see below), another job, or a control flow task.
Parameters can be defined at the job level (global parameters that are pushed down in all tasks) or in a specific task. Another important feature is the trigger, which can be defined at the job level. The last important feature used in the thesis project is the possibility of managing jobs programmatically via REST API.

2.3.7 Delta Live Tables

Delta Live Tables (DLT) [28] [29] is a declarative framework developed by Databricks for building reliable and maintainable data streaming processing, especially ETL pipelines. This framework allows defining transformations by removing orchestration, cluster management, monitoring, error handling, and failure recovery.

Using Spark Structured Streaming Programming implies defining the data pipeline using a series of separate Spark tasks; on the other hand, Delta Live Tables allows the definition of streaming tables and materialized views that the system creates and keeps updated. Databricks automatically manages tables built with the Delta Live Tables framework; in this way, it automatically determines what updates are needed and the current state of the tables, and it performs maintenance and optimization tasks. The data transformations are done based on queries defined for each processing step. With Delta Live Tables, it is also possible to enforce the schema and define the expected data

quality; moreover, it is possible to define how to handle records that do not match the expected data quality. The tables created by the DLT framework are Delta Tables with the same features and guarantees provided by Delta Lake. The DLT framework supports only Python or SQL programming languages for declaring the operations.
The result of the declarative queries should create streaming tables, materialized views, and views.

- Streaming table: A streaming table is a Delta Table that adds support for streaming and incremental processing operations. The Streaming Tables are designed to process growing datasets by manipulating each row only once. They are designed to manage append-only sources in which the required elaboration can be done incrementally as new data arrives.
- Materialized view: A materialized view contains pre-computed results tracked in the Catalog. Materialized views are updated according to the pipeline update schedule. Query results are modified and updated at each execution of the pipeline. Delta Live Tables provides materialized views without the difficulties of efficiently applying updates.
- View: The views in Databricks compute the result from the source when they are queried. The views are tracked in the Catalog and are convenient for executing intermediary queries to enforce data quality and divide complex queries. The views defined with the Delta Live Tables framework differ from standard views defined with Databricks SQL because they are not tracked in the Catalog but can be referenced only from the pipeline in which they are defined.

As said before, DLT is a declarative framework; an orchestration tool is needed to instantiate the declared objects: the DLT Pipeline. It is the main unit to configure and run DLT data processing. The Delta Live Tables Pipeline infers the dependencies between the tables, ensuring that updates occur in the proper order. Delta Live Tables examines the dataset's actual and desired states before creating or updating datasets. The DLT Pipeline also performs the processes of efficiency, monitoring, data quality enforcement, error handling, and failure recovery. This automatic management reduces time and costs to elaborate a streaming data flow in real-time with respect to the use of Spark Structured Streaming Programming. Moreover, it also reduces the deployment time for ingestion processes that do not require complex data elaborations. Each DLT Pipeline has a configuration section through which it is possible to set the collection of notebooks and files that

define the Delta Live Tables datasets and declarative queries. Other important settings are: where the tables will be saved in the Workspace, the cluster policy, and the trigger type. In the configuration section, it is also possible to define parameters available to all notebooks containing the Delta Live Tables code. One of the most important advantages of using DLT instead of Structured Streaming is the fully automatic handling of the Slow Changed Dimension (SCD) of types one and two. To do it with DLT, you only need to specify the field by which the input data must be ordered, the primary keys, and the SDC type; the framework fully manages all the other operations. It is much less complex than providing the same functionality with classic Spark code.

Until now, Delta Live Tables has been described as a powerful framework for handling ETL pipelines; this is true, but it also has limitations that it would be best to be aware of before adopting it. The following limitations result from experimentation carried out during the thesis work.

- The first problem is related to debugging. When you define DLT-declared queries and datasets in a notebook, it is impossible to run the notebook directly. The only way to run the code is to attach them to a DLT Pipeline and run it. Problems emerge when something goes wrong; for example, the final output is not what is expected, or runtime and compiling errors occur. In the last two cases, error messages are provided in a less friendly way than the classic Spark errors; moreover, the solution requires more time because running only a part of the code is impossible. However, bigger problems occur where the final output is not what is expected since it is impossible to directly inspect intermediate results by adding display and print operations. Even adding prints to track code execution is ineffective, let alone using more advanced tools like the built-in Databricks interactive debugger. In simple words, these types of issues could cause the lengthening of development time, especially in situations in which a large number of data transformations are involved and there is a lot of interaction between different data sources.

- Related to the previous point, using DLT to perform complex data transformations may not be a good choice. Moreover, some useful methods available in Spark Structured Streaming Programming are not available with the DLT framework. An example is the foreachBatch method, which allows batch functions to be applied to the output data of every micro-batch of a streaming query.

- DLT Pipelines allow to define pipeline's parameters, but pushing down

the parameters of a job in which the DLT Pipeline is a task is impossible. It is a strong limitation of the orchestration mechanism. Consider that the job's parameters are pushed down when tasks run another job.

- DLT does not allow the managing of cluster libraries. Installing and using libraries that are not the Spark default libraries available over the cluster is impossible. For example, it is not possible to read and write from CosmosDB, but it is possible to read, write, and manage files on the storage account. Depending on the project, it may be required to devise complex and not intuitive work around that need to be orchestrated in the proper way.

- Used shared custom libraries that ensure the reuse and maintainability of the code could become tricky with DLT. The only way to use them is to define these libraries into Python files that are located in the same directory of the DLT notebook that uses them. Obviously, this limitation can lead to code duplication or poor code organization, especially in situations where many notebooks need to be managed.

- Related to the previous problem, in DLT notebooks, it is impossible to use magic commands to execute other notebooks; this limits the possibilities of code reuse to only Python files in which one or more classes are defined and can be imported into the DLT notebook.

- The DLT tables can be defined only once; this means that a DLT table can be the target of a single operation, so it is unfeasible for two distinct data sources to update it.

- All the tables defined in a DLT pipeline must be stored in the same Schema (database) of the Metastore; the destination Schema is defined in the DLT Pipeline settings. This behavior can result in a limitation of logical data organization. In Chapter 3, this problem emerges.

2.3.8 Auto Loader

The Auto Loader [30] is a tool for ingesting new data files as they arrive in the cloud storage. It is a data source for the Spark Structured Streaming Programming or Delta Live Tables framework. The Auto Loader can also scale to properly manage near real-time ingestion of millions of files per hour. It supports multiple data file formats such as JSON, CSV, XML, PARQUET, AVRO, ORC, TEXT, and BINARYFILE. The data loaded by the Auto Loader

can reside in numerous cloud storage, such as Azure Blob Storage, Azure Data Lake Storage Gen2, Amazon S3, Google Cloud Storage, ADLS Gen1, and Databricks File System.

To ensure that each file is processed exactly once, as a file is discovered, its metadata are written in a persisted and scalable key-value store located in a checkpoint folder over the cloud storage; this behavior also enables an easy and fast recovery in case of failures, and guarantees the exactly-one load of each data file. Therefore, Auto Loader is a powerful, versatile, and fault-tolerant tool for ingesting data files in real time.

2.4 Theoretical concepts

The section explains the theoretical concepts used to build the platform.

2.4.1 Slowly Changing Dimensions

The Slowly Changing Dimensions (SDC) [31] refers to the methods used to track the changes in the dimension records of a data warehouse. There exist different types of SCD [32]. The ones used in the thesis work are explained below.

- SCD type 0: This type is assigned to attributes that never change and have durable values. For example, the birth date of a person.
- SCD type 1: In SCD Type 1, the new data overwrites the existing. So, the old value is lost, and no history is maintained.
- SCD type 2: SCD type 2 maintains a complete history of values. When the value of an attribute changes, the current record is marked as the old record, and the time window for which it was valid is also recorded. Then, a new record is created with the new value. This record is marked as current. To provide these functionalities, at least three columns must be added: one boolean that tracks if the record is the current one and two timestamp columns to store the record's start and end time validity. When a new record arrives, the table's primary keys discriminate between an update of an existing record or a new value.
  In systems where an update or a new record may arrive several times, a hash field calculated on all the record fields may be added to the table. In this way, to decide if a new record is an update or a fake update, it is only necessary to compare the two hash fields when the keys match.

Chapter 3

The Data Platform

This chapter deals with the specifics of the problem and how these were addressed. Furthermore, it also explains the motivations behind the technical choices, how the platform was structured, the code organizations, and the orchestration mechanism.

3.1 Problem specification

The objective of the thesis work was to build a prototype of a data platform able to manage the ingestion of multiple data sources, perform data engineering tasks, make available data that satisfy different business needs, and provide data visualization and notification services. In addition, robustness and resilience were required for each task. Below, there is a brief list of the requirements.

- **Data sources**: The company has multiple data sources, in particular, a relational database source and numerous decentralized sources. For the prototype project, it was decided to handle nine tables of the relational databases and the messages that arrive from the alarm plants. These messages concern alarm activation, deactivation, and intrusion detection. The relational database must be considered a scarce resource, so as few reads as possible should be performed.

- **Configurability and modularity**: The modularity of the structure and the configurability are two important requirements. It was required to build the platform with configurable modules to allow easy platform widening and easy data management tasks. In addition, there was a

requirement to design the platform to allow easy and fast addition of new data management tasks to meet new business needs.

- **Data destinations**: It was required to serve three final data destinations. The platform must be able to upload the output of the data engineering tasks to a NoSQL database accessible through APIs. The other data output consists of providing notification services. Two notification services were required for the prototype: one for when an alarm system is triggered and signals an intrusion, and one for managing the messages about the arm and disarm of an alarm plant. The last data output consists of an interactive dashboard to visualize the trend of the number of alarms fired over a time interval. The visualization should take into account the clients with the highest number of fired alarms.
- **Data ingestion**: The platform must provide three data ingestion methods: full, delta, and streaming. Additionally, the possibility to switch from one method to another easily and quickly through configuration parameters was required.
- **Real-time and non-real-time needs**: The platform must also satisfy real-time and non-real-time needs.
- **Orchestration mechanism**: Due to the complex interaction of different components of the system, it was required to have a centralized orchestration mechanism to manage various situations easily. The centralized mechanism must ensure data integrity.

3.2 The Architecture

The Medallion [33] design pattern was chosen for the platform architecture. This means organizing the data ingestion into different consecutive stages. In this way, a progressive improvement in data structure and quality is achieved. Additionally, the data are written in a persistent storage at each stage. The use of the medallion approach for managing an ingestion process allows the possibility of recreating the tables from raw data or any specific stage at any time without reading from the data sources or recomputing the preceding stages. Moreover, it allows to have an easy incremental ETL process and an easy data model.

The data platform needs to manage two different data sources: a relational database and a network of IoT devices composed of sensors of the alarm

plants managed by the security company. The relational database contains data about the clients, the contracts, the alarm plants, and the sensors. The network of IoT devices consists of the anti-theft plants' sensors and the plants' control units. These two entities send two types of messages: when a plant is activated or deactivated and when a sensor detects an intrusion and fires an alarm. The messages are sent as JSON files (one file per message) and uploaded in a specific storage account container.
Given the prototypical nature of the thesis work, both data sources were simulated. The relational database was modeled with a storage account container to which CSV files for each table were uploaded. Specifically, each table was represented with a folder. The contents of each table were divided into multiple CSV files so that the arrival of new data could be simulated by uploading the files at different time instants. The simulation of incoming messages from burglar alarms is limited to the fact that the messages are uploaded by a Python script to the storage account container instead of being uploaded by the alarm systems.

The data platform was designed to handle two data ingestion processes: one for each data source type. In this way, it was possible to address the different challenges modularly. As said before, the architectural design was based on the Medallion pattern. So, the raw bronze and silver stages were built for each source type. Two programming paradigms were used to define two distinct ingestion processes to handle the two types of data sources. Figure 3.1 provides a high-level view of the data platform and its data flows. It is easy to see several stages in the data streams, not just the Bronze, Silver, and Golds as the traditional medallion approach [[33]]. The operations done up to the silver layer are not designed to satisfy business needs directly but to provide stable, valuable, and updated data from which to start the business logic processes. On the other hand, the operations performed in the business logic are the ones that directly satisfy the business needs. With this design, different efforts can be devoted to the various operations; moreover, an ingestion process can satisfy very different business needs: as soon as a new business need emerges, the developers start the work with the Silver stage data, not the raw source data. So, the ingestion process up to the Silver stage feeds multiple business logics: one for each specific business need. This also leads to using less computational resources with respect to a situation in which all the business needs have in their logic the ingestion process.

The requirements state that the platform should be as modular as possible. Modularity was achieved both by taking advantage of the inheritance made available by Python, and by developing code with parameters that

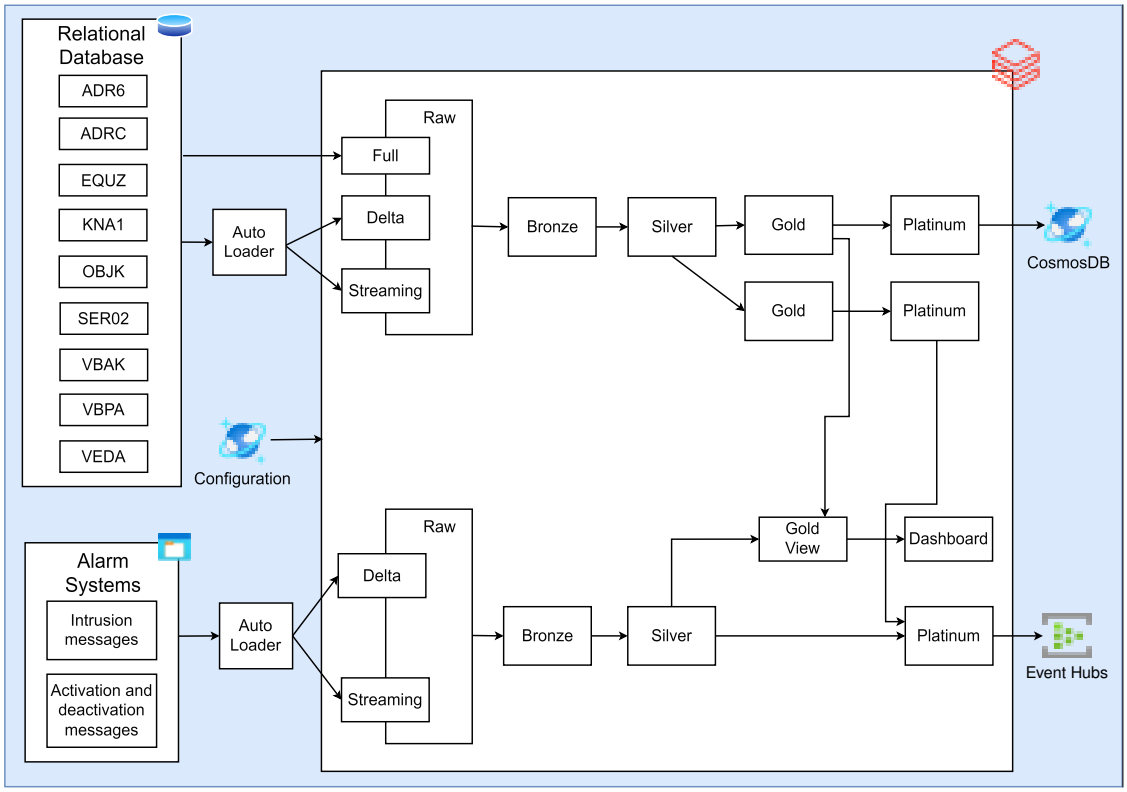

Figure 3.1: The Figure shows the high-level view of the data platform architecture and the data flows. The two data sources, the data destinations, and the tool that hosts the configurations are also shown.

can be easily configured at runtime. The configurations were stored in a CosmosDB database as JSON-like documents. The configuration documents were organized in collections, one for each type of configuration needed by the platform.

In the following sections, each platform part is explained in detail. First, the general code structure [3.2.1], followed by the two ingestion processes [3.3], and, in the end, the business logic tasks [3.4].

3.2.1 General code structure

The platform's core was built with Databricks. All the code is accessible and editable through the Databricks Workspace; it was also versioned in a company's git repository. The code comprises Python Notebooks and Python files organized in the directories shown in Figure 3.2.

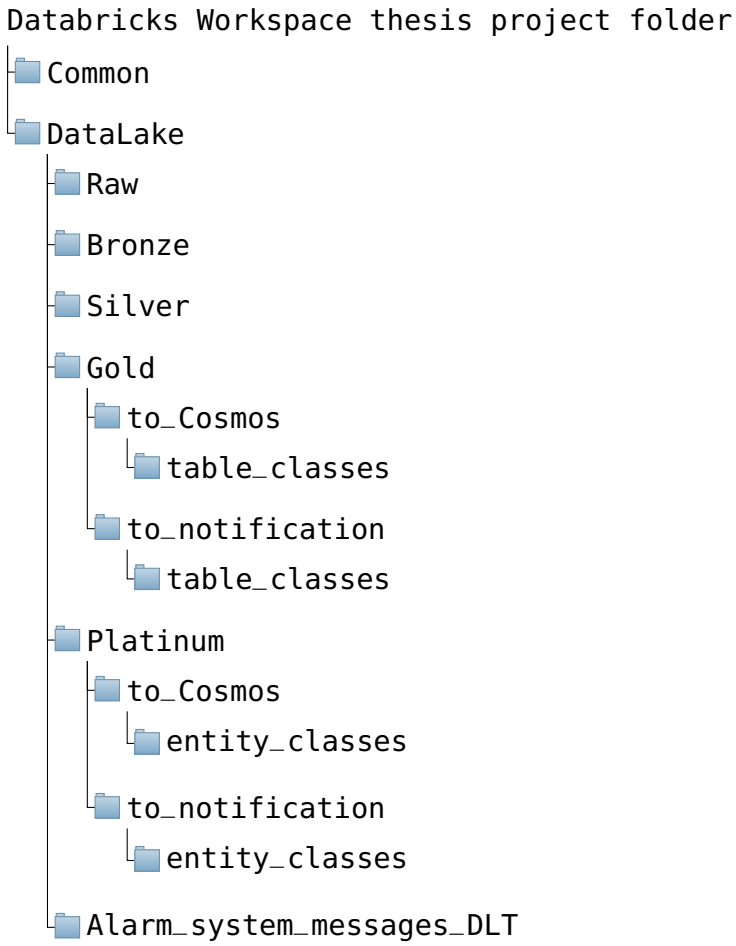

Figure 3.2: The tree shows the code organization in the Databricks Workspace at the folder level. What each folder contains is explained in Sections 3.3 and 3.4.

The `Common` directory contains notebooks with useful code to reach the required modularity. The following notebooks are available in the directory.

- `Mount`: It is responsible for setting up Databricks and allowing Unity Catalog access to specific storage account containers. It performs a mounting process for the containers. A secret key is required to mount them. The key is not written in the code but is read runtime from the Key Vault through a secure method of the dbutils library. By default, the library is available in the Spark cluster.

- `TableUtils`: Defines classes useful for interacting with both Delta Tables

registered in the metastore and tables written in the storage account containers reachable from Databricks but not registered in the Metastore. In particular, two classes are defined.

  - A primitive class (`TableUtilsRaw`) that makes available raw methods to read, write, and delete both tables registered and not registered in the Metastore.
    The write method takes in input a Spark DataFrame, the Schema name, and the table name. It allows writing the table in a customized storage location or as a managed table for which Unity Catalog manages all accesses. The table is a Delta Table registered in the Metastore in both cases. The choice is given through a boolean flag to provide uniform management of the memory paths. The method also allows writing the DataFrame content into the storage account without registering the table in the Metastore. It is also possible to personalize the writing mode (overwrite or append) and the file format (Delta, Parquet, and CSV).
    The read method returns a Spark DataFrame and has parameters to specify the Schema name, table name, and from which source read. If a table is not registered in the Metastore, the method automatically builds the path to read from using the input parameters.
    The method to delete a table uses the truncate or drop operation to provide the most efficiency available. The drop instruction is used only in cases in which the table schema changes. For tables not registered in the Metastore, the files are deleted directly from the storage account container. Also, this method has parameters to configure its behavior; the most important are the Schema and table name.
    For some parameters of the above methods, default values are set to facilitate the usage.
  - The class `TableUtils` inheritance the raw methods from `TableUtilsRaw` and wraps them to provide more complex functionalities. For example, it makes available methods to perform a write in full mode or in delta mode directly. Methods to give functionalities to write with a specific Slow Change Dimension (SCD) type. Or methods for backing up files before deleting a table. Also, other helpful methods are provided.

- `CosmosDbUtils`: It defines a class that provides methods to interact with a CosmosDB NoSQL database. The init method allows defining the

database name, the account endpoint, and the account key required to interact with the desired database. The parameters take values when the class is instantiated. The class provides methods for creating a container, writing one or more documents, and reading documents from a container. All input and output data are Spark DataFrames; each row corresponds to a database's document.

- `EventHubUtils`: It defines a class that provides parameters and methods to interact with an Event Hub. The init method allows defining the Event Hub Namespace, the access policy name, the access policy key, and the Event Hub name. Due to the interaction with the event hub being performed through REST APIs, the method builds and makes available the Event Hub URL, the headers, and the auth token necessary to authenticate the REST requests. So, the class defines internal methods to build these objects.
  The class has only a public method to write the content of a DataFrame into an Event Hub. This method takes in input a DataFrame and an optional string to define the type of data that the DataFrame contains. It converts the DataFrame to a list of JSON-like documents. Then, it prepares a data structure of type dictionary to write on the Event Hub. The dictionary has three fields: the operation's timestamp, the list of JSON-like documents, and the field that contains the value of the string given in input to the method or a null value in cases where the string has not been provided. As the last operation, the method performs the REST POST request to write the data.

- `ConfigurationManage`: It defines a class to read the configurations from the dedicated CosmosDB NoSQL database. In the init method, the CosmosDB utils class is instantiated. The required secrets are read securely from the Key Vault. The class provides methods for taking the configuration related to a specific medallion stage and making it available in an easy and ready-to-use format.

- `PrimitiveClassesMedallion`: It defines the primitive classes used by the medallion classes to inherit parameters, implemented classes, and methods. Each primitive class is explained in the section where it is used.

- `CommonUtils`: It is in charge of doing all the imports, including those of the classes defined in the other notebooks in the directory `Common`.

3.3 Ingestion

The ingestion activities involve the medallion stages up to Silver (Raw, Bronze, and Silver). In the project, all ingestion activities that read from sources not directly accessible by Databricks are done through Auto Loader instances.

The data ingestion can be done in three different manners: full, delta, and streaming. Data can also be loaded by one of the three available methods between different stages of the medallion. In all cases, the method to use is decided by a parameter of the job in charge of orchestrating the process.

Loading data in full mode means rewriting the stage data by reading all data from the source or from the previous stage. Consequently, the data operations required to transit from one stage to another are performed on all data. In addition, the full mode must also be used in all succeeding stages in order to maintain the data integrity. Thus, performing a full operation is expensive in terms of time and economics since it requires high computational power. Moreover, it requires accurate task orchestration to propagate the full to all subsequent stages of the data stream up to the business logic. Thus, full mode is utilized in situations where there has been a change in the logic, or where there have been serious errors for which no other recovery operations are available, or also in special situations where it is necessary to reload all data from a specific stage.
Loading data in delta mode means reading and processing only the data that have been changed or added in the time interval between the last reading and the current one. Consequently, this operation is much less expensive than the previous one because only the new data are read, processed, and written.
Loading data in streaming mode means always listening to the source, and as soon as new data arrives or a change is made, it is processed. This mode is useful for all situations that require real-time or pseudo-real-time processing. Obviously, processing a data stream in streaming mode is expensive since the servers must stay on all the time, ready to process what comes in.

3.3.1 Ingestion Relational Tables

This section explains the ingestion process of the tables from the company's relational database. Figure 3.3 highlights the data platform section in charge of the process. The medallion stages involved are Raw, Bronze, and Silver. In this platform portion, the data of each table remains separate and each one follows its specific flow. In other words, each table of the relational database corresponds to a specific data flow. The code and job that handle this section

are the same for all tables; what changes is a parameter of the job. Through this parameter, the configuration related to the specific table is loaded, which customizes the code's behavior. Since the same operations are performed to ingest all source tables of the relational database, the following paragraphs provide a general explanation of the process without referring to a specific table and how the configuration customizes the code's behavior for each data stream. Before explaining the single medallion stages, it is important to know that all the tables in the company relational database have a column that records, for each row, the timestamp of the last operation performed over the data. It is also essential to know that the company's relational database works only in append mode.

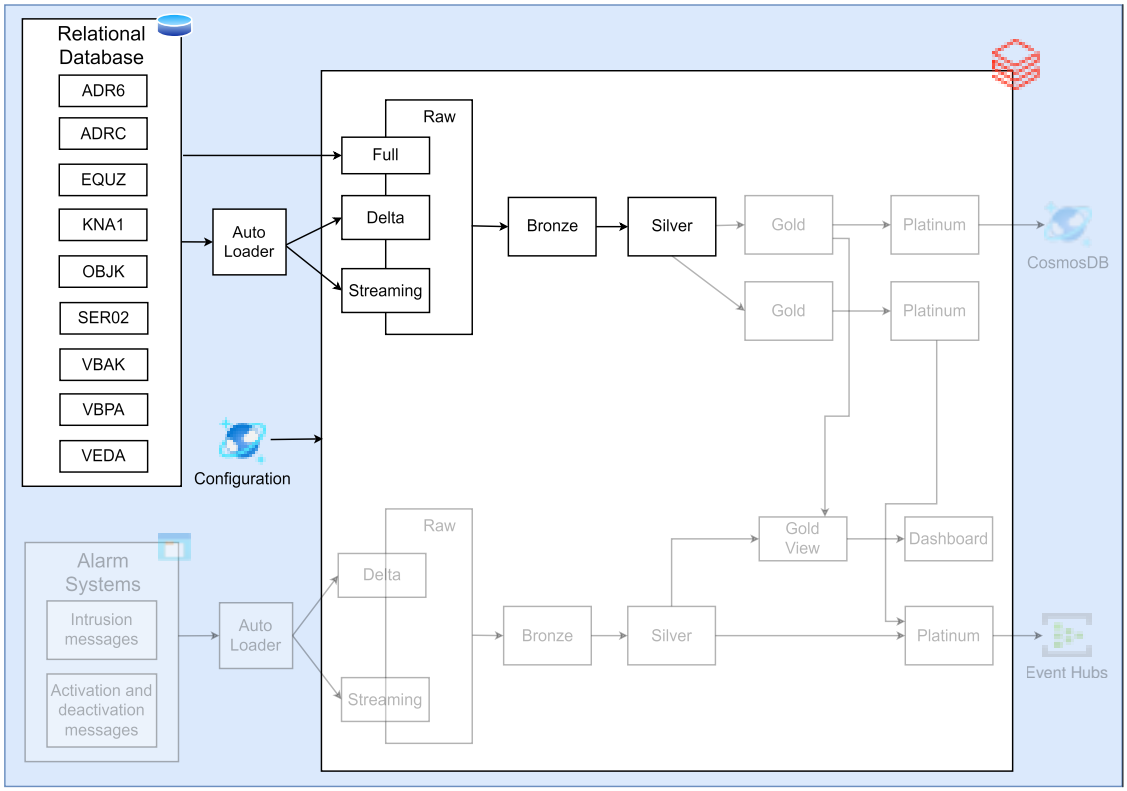

Figure 3.3: The Figure highlights the data platform section responsible for ingesting tabular data from the company's relational database.

In the Raw stage, the data are only copied from the table over the relational database into a storage account container accessible directly from Databricks. In this stage, the data are stored as tables written only in the storage account and not registered in the Metastore. The file format used is CSV. No data

manipulations are done. New data are appended to the table. The reason for this passage is to ensure the least possible usage of the data source.

In the Bronze stage, the data are read from the raw table, the schema is enforced, and the data type of each column is defined. The required cast operations are performed. The data are written on an external Delta Table registered in the Metastore, so the file format used is Delta, and a path is specified for writing the file on the storage account. Also, in this case, the new data are always appended to the table.

In the Silver stage, the data are historicized. The types of Slowly Changing Dimensions available are one and two. The data are written on an external Delta Table registered in the Metastore, so the file format used is Delta, and a path is specified for writing the file on the storage account.

The storage account for this platform section has one container for each stage, and each table corresponds to a directory in each container. The Metastore is organized similarly: the bronze and silver tables are registered in two separate Schemas (databases), and the raw tables are not registered in the Metastore.

At each stage, the code is organized into two notebooks: the class notebook and the main notebook. The notebooks referred to a stage are saved in the Workspace in the same directory following the code organization shown in Figure 3.2.

The class notebook defines the class used in the stage; it inherits methods and parameters from a primitive class defined in the common utilities [Section 3.2.1]. The primitive class is the same for all medallion stages discussed in this section.
The primitive class provides the implementation of two classes: one that contains the methods to manage tables and Delta Tables easily and one that helps to retrieve the configuration from Cosmos. This class loads the proper configuration and makes it available to the stage class. In addition, the primitive class also provides useful parameters that are ready to use and a method useful for managing the full load case. This method performs the backup of the table and checkpoint files before removing the table.
Raw, Bronze, and Silver classes set up a Spark Structured Streaming flow. The different load behavior is achieved by setting the proper trigger parameter in the streaming trigger setting. In particular, the trigger parameter can be set with a time interval or with a parameter that modifies the streaming behavior by loading only the unloaded data and then stopping the stream activity; in simple words, this parameter switches the streaming activity to

an activity that runs only one time and then stops. With this strategy, the same code can be used to handle the three load methods, facilitating code maintainability.
As said in Section 2.3.2, a Spark Structured Streaming activity uses checkpoints to keep track of which data was already read and which are new. These checkpoints are properly organized in a storage account container. To perform a load in full mode, the checkpoints of the specific data stream must be deleted. To give the possibility of recovering operations, both the table's and checkpoints' files are backed up and then deleted using the method described before.

In the Silver stage, the full operation is more complex. Two different procedures are available if it is executed on a table for which SCD type two is enabled: `full_refresh` and `full_append`. The `full_refresh` is the standard full in which all data are deleted and recomputed from the bronze table. The `full_append` is a special type of full available only for tables with SCD two; all data from the silver table are marked as not current, and then all data from the bronze table are reloaded, historicized and stored to the silver table. Also, in this case, the checkpoint files are backed up and deleted.
The other notebook available for each stage is the main notebook. It loads the job parameters, executes the notebook `CommonUtils`, instantiates the stage class, and executes the class method to load, transform, and save the data.

As said before, one logic manages all data processing of this platform part by customizing the code behavior by loading specific configurations for each data flow. For this part of the platform, the configuration is composed of one document for each flow. In other words, one document exists for each table ingested from the relational database.
In the Raw stage, the configuration specifies the table source and the parameters useful for performing the read.
For the Bronze stage, the configuration gives the schema and the casting operations to perform.
In the Silver stage, the configuration provides the type of SCD and the table's primary keys.
In both the Raw, Bronze, and Silver stages, the configuration specifies the trigger interval of the task when the delta load mode is used.
Still, on configuration, other parameters are specified at the job level that orchestrates this part of the platform. Specifically, through the job parameters, the type of loading mode to use, the name of the containers to store the backups and checkpoints, and the name of the table to ingest can be decided. The last parameter is useful for loading the correct configuration

from Cosmos. In addition, when the full load mode is used, it is possible to decide from which stage to perform the operation. Still referring to a full operation, in the job task that is responsible for executing the silver stage, it can be defined the type of full to perform (`full_refresh` or `full_append`). Obviously, the parameter has action only if SCD type two is enabled for the table.

In the end, to have the data in the silver tables updated and ready for the business logic tasks, it is necessary to decide what type of load method to use for each table and run the orchestration job for all tables with the proper parameters. This task is under the responsibility of the global orchestration mechanism explained in Section 3.5.

3.3.2 Ingestion IoT messages

This section explains the ingestion process of the messages that arrive from the network of IoT devices. Figure 3.4 highlights the data platform section in charge of the process. The medallion stages involved are Raw, Bronze, and Silver. The Delta Live Tables framework was used for this ingestion process because it was required to have a fast process. DLT is faster and more efficient in continuous streaming processing with respect to the Spark Structured Streaming framework. Obviously, using the DLT framework involved some challenges, particularly in the testing phase, in reading the configuration from CosmosDB, in organizing the tables in the Metastore, and in the code reuse, heredity, and organization. On the other hand, the orchestration activities and the historicization are much simpler to manage with respect to the Structured Streaming Programming framework. The DLT framework offers a fully managed SCD of types one and two. In particular, the framework provides an automatic reorder and management of the data that does not arrive chronologically. Moreover, the framework automatically manages duplicate messages by taking the first copy that arrives and discarding the others. This feature is very important in a system that elaborates messages from a network of IoT devices. It is important to know that the messages always record the timestamp of the event, the ID of the IoT devices to which the message is referred, and an identifier of the type of message it is. The ID is unique in the company's network. All the devices use the same time zone to produce the timestamps.

Now, a high-level description of what each stage does is provided; then, we enter the details about the code structure and organization, the configuration

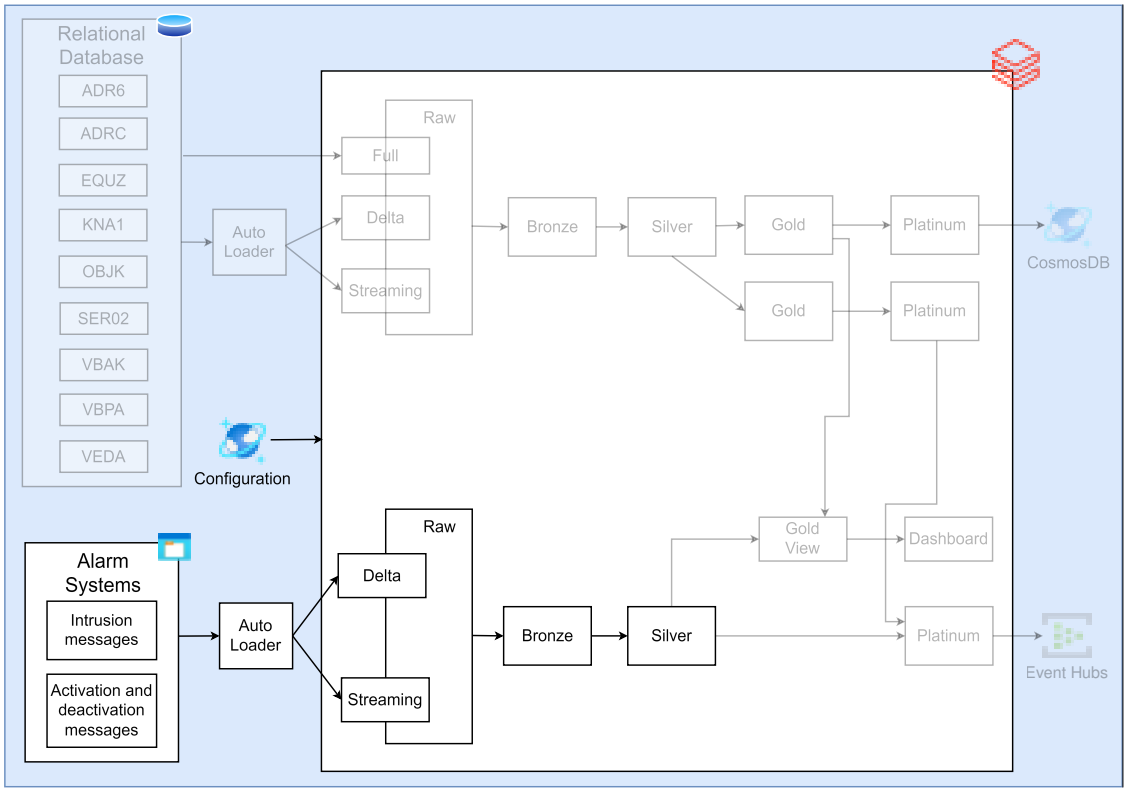

Figure 3.4: The Figure highlights the data platform section responsible for ingesting the messages from the IoT devices.

role, and the table's organization.

The Raw stage is common for all types of messages. It consists of taking the JSON files sent by the plants, transforming them into tabular data (one row for each message), and saving them in a table managed by the DLT framework and registered in the Metastore. The data transformation of this stage makes the required cast operations. In the end, the timestamp in which the message was loaded is added. The Raw stage works in append mode, so the new data are always appended to the table and do not impact the ones already written.

The Bronze stage makes the difference between the types of messages. For the prototype platform, it was decided to handle only two types of messages: the states of the systems (activation and deactivation messages) and the alarm fires. So, in the Bronze stage, the DLT framework creates two processes that read from the same raw table, filter the messages based on the type, select only the columns of interest in relation to the type of message, and save the

result data in a table managed by the DLT framework and registered in the Metastore. The Bronze stage operates in append mode.

The Silver stage is in charge of making data historicization. It reads the data from the corresponding bronze table, reorders them according to the production timestamp (not the ingestion timestamp added in the Raw stage), and performs SCD of type one or two, depending on which is required. Ultimately, the stage saves the data in a table managed by the DLT framework and registered in the Metastore. Clearly, the Silver stage can not work in append mode because it requires modifying records already registered in the silver table to make the historicization.

Figure 3.2 shows a separate directory (`Alarm_system_messages_DLT`) to contain the code that uses the DLT framework. The choice is not arbitrary, but it is the consequence of one of the limitations of the DLT framework explained in Section 2.3.7. This leads to a more chaotic code organization, but it is also the solution for building code that is as modular and maintainable as possible. So, the directory contains several Python files and Notebooks. The Python files define classes that are implemented and used in other classes or in the main notebooks. As in Section 4.1, one main notebook exists for each Medallion stage, and the different behavior between the message type streams is provided through the configuration.

The directory (`Alarm_system_messages_DLT`) contains Python files. It is possible to divide the files into two groups: files that contain common code and files that contain the class to declare the DLT's queries and tables of each stage. To the first category belong only one file: `CommonUtils`. It makes the common imports and defines two classes: one to load and make available the configuration, and a primitive class useful to easily implement the classes that declare the DLT's tables of each stage. It also provides common parameters and the class implementation to access the configuration.
Three files belong to the second category: one for each medallion stage. In each of them, a class is defined with two methods: one to wrap the DLT table declaration in order to allow parameterization, and one to declare several DLT tables at the same stage but with different parameters in accordance with the configuration. This second method calls the first method of the class as many times as there are tables to be declared in the stage. With the DLT framework, declaring a table also implies defining the data transformations to perform over the data.
The DLT directory also includes three Python Notebooks, each serving as the main notebook for a specific stage. These Notebooks import and implement

the corresponding stage class and execute the methods for creating the tables.

For a table built and managed by the DLT framework, it is possible to make it completely managed by Unity Catalog or make it an external table for which Unity Catalog only manages accesses made within Databricks. In the second case, a memory path must be specified. In the thesis project, it was chosen to build external tables to allow more control over the table's files.

The configuration is central in customizing the code behavior. The configuration is stored in a dedicated container of the CosmosDB database. At this point, a very strong limitation of the DLT framework emerged. It is impossible to directly read from Cosmos with the cluster that runs the DLT code (In Section 2.3.7 the limitation is explained). So, it was needed to develop and properly orchestrate a workaround. It is done at the level of the job that runs the DLT pipeline. It consists of the following: before running the DLT pipeline that runs the DLT main notebooks, the job runs a Python notebook that reads the configuration from the Cosmos DB database and writes them into a Delta Table, managed by Unity Catalog, overwriting it each time. This last notebook is run with a cluster that can interact with CosmosDB. So, the DLT code reads the configuration from the Delta Table, and the limitation is bypassed.
In the Raw stage, the configuration provides the data source to read the messages, the name of the raw table to be built, and the memory path to write the table files. The schema is not given, but it was enabled the possibility of automatically adapting the Delta Table schema in reaction to the new fields discovered in the incoming messages.
For each bronze table in the Bronze stage, the configuration provides: the table name, the row table from which read the data, the table schema, the filter condition, the column to select, and the memory path to writing the table files.
In the Silver stage, the configuration provides, for each silver table, the table name, the bronze table from which read the data, the type of SCD to perform, the primary keys, the column according to reorder the data in the SCD task, and the memory path to writing the table files. For the alarm fires data stream, the Bronze and Silver stages coincide because no historicization is needed, so, through the configuration, the Bronze data operations are done, and then the data are directly saved in the Silver table. This behavior is given simply through the configuration file without changing any code line. So, it also demonstrates the configuration power. Moreover, removing a stage speeds up the message processing.

The tables created and managed by the DLT framework are registered into

a single Schema (database) of the Metastore because of framework limitation (Section 2.3.7 explains it). On the other hand, the table's files stored in the storage account are divided into containers, one for each medallion stage, and each table corresponds to a directory in the specific container.

Ultimately, the message data are available in the silver tables, ready to satisfy business needs.

3.4 Business logic

This platform section is devoted to directly satisfying specific business needs. Due to the project prototype nature, it was decided to address only three business needs: provide documents with pre-computed aggregations ready to use, provide a real-time notification system for the messages received from the IoT devices, and provide a dashboard fully managed by Databricks.

In this platform part, the configuration changes its role and becomes a tool to make some operations more efficient but loses its ability to customize the code behavior. When possible, the code was built by exploiting modularity and class heredity. Moreover, some configuration parameters are provided through the jobs in charge of orchestrating the tasks.

3.4.1 Gold and Platinum primitive classes

Before explaining the details about the business tasks, it is required to explain the three primitive classes used to make the code modular and reusable in the business logic: the `GoldPrimitive` class, the `PlatinumAggregatePrimitive` class, and the `PlatinumMessagesPrimitive` class. All of them are defined in the notebook `PrimitiveClassesMedallion` in the Workspace folder `Common` (Section 3.2.1). Some features and behaviors of these classes may become clear after reading the sections in which they are used (Sections 3.4.3 and 3.4.4).

The `GoldPrimitive` class provides useful parameters, the implementation of the class `TableUtils` (Explained in Section 3.2.1) useful to interact with the Delta Tables, and methods to define logic in common to all the gold data processing. These methods are the following:

- `_read`: It reads the data of a silver table by retrieving only the new data added to the table since the last reading operation.
- `_checkpoints_manage`: It manages the checkpoints that are useful to read only the new data from the specific silver table.

- `_manage_full`: It manages the cases in which it is required to perform a full load of the silver table. It backs up both the gold table's files and the checkpoint's files and, after the backup operation, deletes the files and removes the table. These operations use methods provided in the utils section of the platform. When recalled in a case of non-full load, it does nothing.

- `_write`: It writes the data into a specific gold table by downgrading the SCD type from type two to type one. The data are written on an external Delta Table registered in the Metastore, so the file format used is Delta, and a path is specified for writing the file on the storage account; the path is managed by the write method to ensure the uniformity and compatibility with all the platform methods that require the access to the table's files.

The `PlatinumAggregatePrimitive` class provides useful parameters, implements three common utils classes, and defines methods to handle common work in the Platinum data processing aggregation tasks (Section 3.4.3). It provides one object that implements the `TableUtils` class and two objects that implement the `CosmosDbUtils` class. One of these objects is used for interacting with the Cosmos database that stores configurations, while the other interacts with the Cosmos database that holds the output data, as described in Section 3.4.3. Both `CosmosDbUtils` implementations require secure access to secrets for interacting with Cosmos, which are securely retrieved from the Key Vault. The primitive class also defines some methods:

- `_read`: It reads a table from the Gold stage.

- `_sub_documents_factory`: It builds aggregated documents by taking a Spark DataFrame, a list of key fields, and a boolean flag as input. It returns a DataFrame with one column for each key and an additional column containing the aggregated data. The aggregated column can either be of type structure or type list of structures, depending on the value of the boolean flag, which should be set appropriately for the specific case. In both scenarios, the aggregation process involves collapsing all non-key columns of each row of the input DataFrame into a single structure, where each field corresponds to an original column. The structures are combined into a list if multiple rows share the same key values; it is done in accordance with the input flag. Moreover, the developers need to know in advance if it is necessary to have the column of type structure

or list of structures; in case of doubts, it is suggested to set the flag to build a column of type list of structures.

- `_find_max_timestam`: It finds the most recent timestamp among all aggregated documents (Remember that all rows ingested from the relational database have a field that records the timestamp operation. Section 4.1). The method takes a DataFrame with columns of type structure, type list of structures, and standard data types. It finds the most recent timestamp operation for each row and adds it to the DataFrame as a new column.

- `_write`: The method writes the final aggregated documents to Cosmos, where each row of the DataFrame becomes a Cosmos document. It takes as input the DataFrame to be written to the NoSQL Cosmos database, the target database collection, the name of the DataFrame column representing the document's ID, and a time interval.
  In the case of a full load, all DataFrame rows are written to the database. Otherwise, only new or updated rows are written since the last write operation. To achieve this, the method reads the latest max timestamp operation from the collection's configuration document, subtracts the provided time interval from the retrieved timestamp, and filters the DataFrame to keep only rows with a max timestamp greater than this value. These rows are then written to Cosmos.
  The ingestion time interval is a safeguard, managed by the global orchestration mechanism (Section 3.5), to prevent any updates or new data from being lost due to platform delays that may occur during ingestion, which may be triggered at variable time intervals. Although this approach may result in some unnecessary overwriting of Cosmos documents, the performance impact is minimal due to the small number of redundant write operations.
  After completing the write operation, the method identifies the max timestamp in the input DataFrame and updates the collection's max timestamp in the Cosmos configuration document. The collection max timestamp operation is updated only if the write operation has been concluded successfully for all documents; in this way, in case of failure, no data are lost, but simply, in the next run, they will be written.

The `PlatinumMessagesPrimitive` class provides useful parameters, implements two common utils classes, and defines methods to handle common work in the Platinum stage that writes data on the Event Hubs (Section 3.4.4).

It provides an object that implements the `TableUtils` class and one that implements the `EventHubUtils` class. The last implementation requires secure access to secrets for interacting with the Event Hub, which are securely retrieved from the Key Vault. The primitive class also defines a method to read the data from a table registered in the Metastore Schema (a database in the Metastore) dedicated to the table managed by the DLT framework. The read operation is performed in streaming using the Spark SSP framework. Another method sets up a streaming data flow by taking the output object of the read method and configuring it with checkpoints and trigger settings. The other streaming operations are performed by the class that inherits this primitive class.

3.4.2 Data model

To understand the business logic tasks, it is important to know the data model in the company's relation database, from which a part of the data is ingested. As said before, the thesis project is a prototype work, so only some database tables are ingested. Figure 3.5 shows the Entity-relationship model of the interesting part. A brief explanation of the entities is provided below.

- Client: It represents the company's clients. They can be a physical person or a legal entity.
- Physical Address: It represents the regional address: country, city, street, location, phone, etc.
- Email Address: It represents the mail contact: mailbox, type, etc.
- Contract: It represents the contract and tracks its data.
- Service: It refers to a sub-contract for a specific service requested by the client. When a client utilizes multiple services of the same type, these services are grouped under a single contract. The contract manages and tracks information that applies to all associated sub-contracts.
- Located Performace: It represents the entity in which a service is provided. In other words, the location in which the service is offered. For example, in the case of a company with multiple locations, the offices in which some services are provided are the "Located performance", while the company is the client. Each "Located performance" has its own information that differs from the client's information.

- Validity: It represents the validity of a contract or a service. So, each contract is associated with a validity period that is applied for all services if no other validity is associated with a specific service. In any event, the validity of a service can not exceed the one of the contract with which it is associated.

- Plant: It represents an alarm system and tracks the information about it. Each burglar alarm is associated with a service. In this model, they can be the same entity, but service is a more general item; remember that for this project, only a small part of the company activities and data was considered, but the security company does not provide only alarm systems.

- Device: It represents the single devices that compose an alarm system.

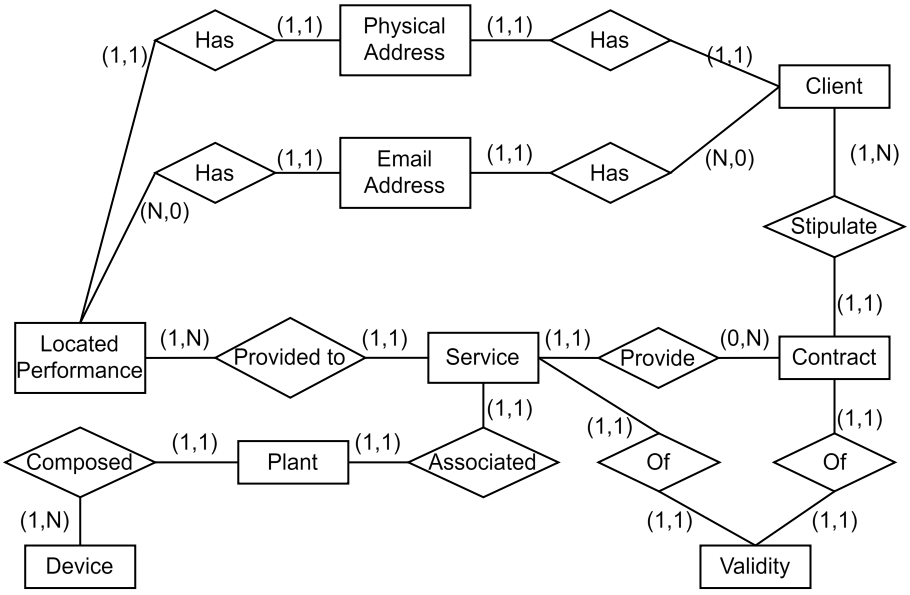

Figure 3.5: The Figure shows the Entity-relationship model of the data ingested by the data platform part explained in Section 4.1.

The company's relational database stores the records of the entities Client and Located Performance in the same table. The records for each entity can be separated using a column that acts as a flag.

3.4.3 Aggregated data

The company required already aggregated data saved on a NoSQL document database that could be accessed with REST APIs. Using a NoSQL database also allows the possibility of adding or removing some fields in the new documents without changing the ones already written. Moreover, it was also required to have the possibility to interact with the database using other APIs to satisfy future needs. So, NoSQL Cosmos DB was chosen.
The advantage of having already aggregated data consists of performing the data transformations, especially the join operations, only once and not every time the data are queried. Moreover, developing the APIs for data access is easy, as it only involves retrieving one or more documents from the appropriate collection.
Figure 3.6 shows the part of the platform that addresses the business need explained in this section. The medallion stages involved are Gold and Silver; below, each stage is explained in detail.

The Gold stage takes the data from the Silver stage. In the Gold stage, each data stream maintains the division of the Silver, so for each silver table that is required to satisfy the business need, a gold table exists. The business need explained in this section requires the use of data from all the silver tables available.
In the Gold stage, filtering and mapping operations are performed over the data. The filter operation consists of keeping only the current records of the Silver stage because, in this task, the historicization is not useful. In simple words, a downgrade of the SCD type, from two to one is performed. The names of the table's columns are also changed to provide more user-friendly and explanatory names for the field of the final JSON-like documents stored on Cosmos. A selection operation is also performed to keep only the desired fields of each table in the final Cosmos documents. At the end of the data transformations, the data of each gold data flow are written on an external Delta Table registered in the Metastore, so the file format used is Delta, and a path is specified for writing the table's files on the storage account.
Since each gold data stream corresponds to a specific silver table, updating the gold tables only requires reading the new data that has arrived since the last Silver stage update. Thus, the Gold stage is built using the Spark SSP framework, and the trigger option is set to read only new data and then stop the activity. The reason behind the type of trigger chosen will become clear at the end of the explanation of the Platinum stage.

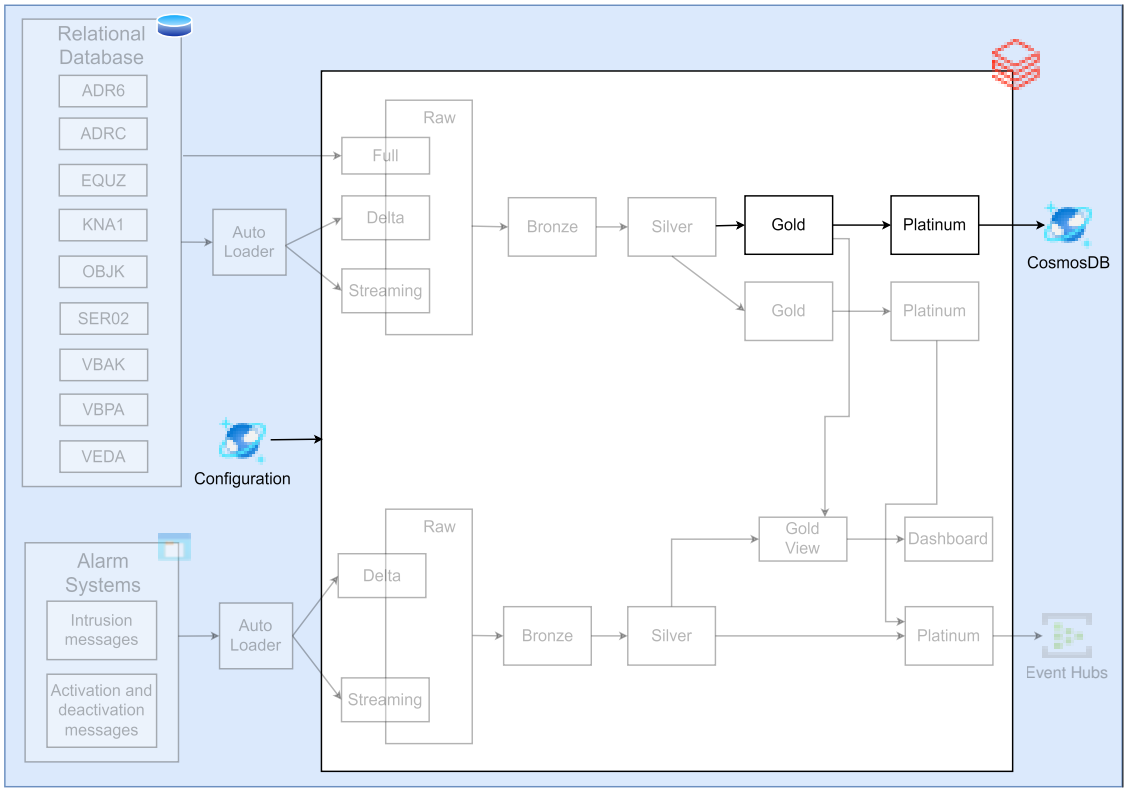

Figure 3.6: The Figure highlights the data platform section that is in charge of precomputed data aggregations to store on Cosmos DB and make them ready to use.

The code referring to the gold stage of the business need explained in this section is stored in the directory `Gold\to_Cosmos` of the Workspace (Figure 3.2). This directory contains a main notebook and the `table_classes` directory. It contains the notebooks that define one class for each gold table. These classes inherit from the `GoldPrimitive` class explained in Section 3.4.1, and define two methods. The first one is an internal method to perform the selection and mapping operations for the specific stage. It also recalls the writing method of the primitive class. The other method, `silver_to_gold`, builds the whole SSP activity that brings the data from the silver table, makes the transformations, and saves them in the gold table. The method first recalls the method to manage the full load cases defined in the primitive class, then recalls the read and checkpoints management methods always defined in the primitive class. As last operations, it uses the `foreachBatch` method of the SSP framework utilizing the internal class method defined before and starts

the process.
To manage the whole activities of the stage, a single main notebook was defined due to all activities are part of the process to satisfy a single business need. The main notebook retrieves the job parameters, performs the operations to import all the common libraries, and instantiates the table classes. For each class, it runs `silver_to_gold` method. The main notebook is run by a task of the job that orchestrates all the activities to address the business need, which is explained in this section.

The Platinum stage is responsible for building the aggregated documents to write on Cosmos. But before explaining how the process is technically carried out, it is necessary to define what is to be achieved in relation to the data model in Figure 3.5.
To satisfy the business requests, was decided to create four types of aggregated documents: Client, Contract, Located Performance, and Plant. The documents are stored in four Cosmos collections. The division of documents into the collections is done based on the document type. A dedicated Cosmos database stores the four collections. Figure 3.7 gives the document structure

```
{"id": str,
 "name": str,
 "physicalAddress": {
    "country": str,
    "city": str,
     ...
    },
 "emailAddresses": [
        {"mailbox": str, "type": str, ...}
    ],
  ...
}

```

Figure 3.7: It represents the JSON-like structure of the Client documents. Only the fields useful for understanding the document structure were reported.

of the entity Client. Only the fields useful for understanding the purpose and showing the aggregation performed were reported. Each document recorded

the data about a company client. The informations about the physical address are reported in a sub-document. The email addresses are reported as a list of documents because a client can have multiple mailboxes. The data about the contracts are not replicated in the client entity because they change a lot with respect to the data about a client, and the business does not need to perform frequent queries to retrieve all clients' contract data. Additionally, CosmosDB provides a by default indexing of all fields, so it is not a costly operation to query the contract collection to retrieve the client's contracts when necessary.

```
{"id": str,
 "clientID": str,
 "validity": {
    "startDate": date,
    "endDate": date,
     ...
    },
 "services": [
        {"serviceID": str,
         "locatedPerformanceID": str,
         "validity": {
            "startDate": date,
            "endDate": date,
             ...
            }
         ...
        }
    ],
  ...
}

```

Figure 3.8: It represents the JSON-like structure of the Contract documents. Only the fields useful for understanding the document structure were reported.

Figure 3.8 gives the document structure of the entity Contract. Again, only

the fields relevant to understanding the purpose and exhibiting the aggregation performed are shown. A document represents a single contract and includes the client ID to which the contract is associated. The validity data of the contract are stored in a sub-document. The informations about the related contract services are aggregated and reported in a list of documents. Each contract service sub-document records its validity, the Located Performance ID where the service is provided, and other data.

Figure 3.9 shows the structure of a document that aggregates the data about a Located Performance. It records its physical address and mailbox, respectively, as a sub-document and a list of documents. The Located performance document also replicates the information about the contracts and the services in which it is involved. So, the contracts are recorded as a list of documents. Each contract contains a list of documents that record the respective services related to the specific Located Performance. So, the contracts' data are not simply replications of the contract collection document but are customized documents containing only the contract data related to the specific Located Performance. The client's data are not recorded, but the client's ID is recorded, so, if necessary, they can be easily retrieved from the client collection by exploiting the automatic Cosmos indexing.

The last entity document structure is given by Figure 3.10. It aggregates the data about a single alarm system. The data about the devices that compose the plant are recorded in a list of documents: one for each device. The plant document also collects information about the physical address in which the plant is located and the validity data of the related service. Moreover, the contract and service IDs are also recorded if it is necessary to retrieve more information.

According to Cosmos theory (Section 2.2.3), each document must include some mandatory fields. While some fields are automatically managed and added by the system, the `id` field must be handled by the developers. Every document contains an `id` field, which serves as the document's unique identifier within the collection. The uniqueness of the IDs was ensured by utilizing the primary keys of the company's relational database. The ID field is also useful for updating documents already stored in the collection. When a new document with an existing ID is sent to the collection, the new data fully overwrites the old document. In other words, updating a single field within a document is impossible; the entire document is replaced. (when this work was being written, the Python API to use the Cosmos' feature to update a document partially was in preview release, so it was decided not to use it). The other field that cannot be absent in the documentsis the one assigned the

```
{"id": str,
 "clientID": str,
 "physicalAddress": {
    "country": str,
    "city": str,
     ...
    },
 "emailAddresses": [
        {"mailbox": str, "type": str, ...}
    ],
 "contracts":[
        {"contractID": str,
         "validity": {
            "startDate": date,
            "endDate": date,
             ...
            },
         "services": [
                {"serviceID": str,
                 "validity": {
                    "startDate": date,
                    "endDate": date,
                     ...
                    },
                 ...
                }
            ],
          ...
        }
    ],
  ...
}

```

Figure 3.9: It represents the JSON-like structure of the Located Performance documents. Only the fields useful to understanding the document structure were reported.

```
{"id": str,
 "alarmCode": str,
 "devices": [
        {"deviceID": str, "type": str, ...}
    ],
 "contractService": {
        "serviceID": str,
        "contractID" : str,
        "validity": {"startDate": date, "endDate": date, ...}
    }
  "physicalAddress": {
    "country": str,
    "city": str,
     ...
    },
  ...
}

```

Figure 3.10: It represents the JSON-like structure of the Plant documents. Only the fields useful to understanding the document structure were reported.

role of partition key for the collection. The document `id` was chosen in the Client collection because it is the only field that ensures enough value variety. Moreover, no field, when used as a partition key, improves query performance for the queries relevant to business needs. In the other three collections was chosen the `clientId` because it ensures enough variety in the values and increases the query's performance to retrieve all the collection's documents referred to a specific client; it is particularly true when the collection becomes very large.

The Platinum stage reads the data from the gold tables, performs the data aggregations, and writes the aggregated documents in a Cosmos NoSQL database. It is the only stage that does not write data in any Delta Table registered in the Unity Catalog Metastore.
Building an aggregated document requires reading multiple gold tables and performing aggregations and joins, so it is not possible to read only the new

data, as in the preceding stages, but it is necessary to read always all data from the gold tables required to build the document. Additionally, the new data or updates can arrive from all tables used to create the aggregation and build or update an aggregate document required to join new and past data. So, it is not possible to use the Spark Structured Streaming Programming, but Spark Batch processing is required using the Spark SQL component because the data to manage are structured.
The code referring to the Platinum stage of the business need explained in this section is stored in the directory `Platinum\to_Cosmos` of the Workspace (Figure 3.2). This directory contains a main notebook and the `entity_classes` directory. It contains the notebooks that define one class for each Platinum entity. These classes inherit from the `PlatinumAggregatePrimitive` class explained in Section 3.4.1, and define two methods: one to perform the aggregation operations and the other to write the result into the appropriate Cosmos collection. The last method wraps the `_write` method defined in the primitive class. Another method is defined. It is more complex and is in charge of building the DataFrame to write on Cosmos; it properly uses the `_read`, `_sub_documents_factory`, and `_find_max_timestam` methods defined by the primitive class. For all entities, the process consists of the following operations:

- Reading the gold tables' data required to build the aggregations. For example, in the case of the Client entity, it implies reading the Client, Physical Address, and Mail Address tables.

- For each sub-document (In the cases of lists of documents, the operations are the same) that we want to create, it is required to join the DataFrames that contain data to put in the sub-document and perform the necessary selection and filtering operations. Then, it is crucial to choose the fields to use as keys and do the aggregations using the `_sub_documents_factory` method. As the last operation, the aggregated DataFrames are joined with the entity's DataFarme, and the DataFrame to write on Cosmos is obtained. For example, to build the `physicalAddress` sub-document of the Plant entity, it is necessary to join data of the Service, Located Performance, and Physical Address, select only the fields to put in the sub-document, plus the plant ID, and use the `_sub_documents_factory` method with key Plant ID. As the last operation, it is required to join what is obtained with the Plant DataFrame using the Plant ID as a key. Aggregations can be more or less complex depending on the case. Moreover, it is important to use the type of join best suited to the specific situation

and, if necessary, use Spark's cache method for some DataFrame to optimize the performances.

- Write on Cosmos the DataFrame with all the required aggregations using the dedicated class's method. The primitive class provides all the features to perform and optimize the write operation as is explained in Section 3.4.1.

A single main notebook was created to manage the Platinum activities for the same reasons as the Silver stage. The notebook retrieves the job parameters, performs the operations to import all the common libraries, instantiates the table classes, and, for each class, runs the method for building the aggregated DataFrame and the one to write on Cosmos.

The orchestration of the activities of this section is in charge of a job with two tasks. The job defines general parameters such as the containers to store checkpoints and backups, the flag to discriminate between a full or a delta operation, and the parameters to define, in case of full, if it involves both stages or only the Platinum stage.
The job should be triggered periodically to ensure the data on Cosmos are updated. Determining the appropriate time window to activate the job requires considering both the need for up-to-date data and the associated economic costs. The global orchestration mechanism allows for this time window to be set, and all the operations are managed to ensure data integrity. For example, suppose a full load was performed over a silver table used to build the aggregated documents. In that case, the global orchestration mechanism ensures it is also carried out for the Gold and Platinum stages.

3.4.4 Alarm systems notification

The company was required to have a notification system for the messages that arrived from the alarm plants. As said before, the thesis work managed the messages indicating the firing of an alarm and the activation and deactivation state. Such messages contain the minimal amount of information possible; this speeds up the transmission and the processing but needs to be integrated before providing the messages to the notification system. For example, when a sensor detects an intrusion and the alarm fires, the notification system must provide the address to send the vigilante, the phone contact of the owner, etc. All this information is not contained in the message itself, but is available in the silver tables of the platform explained in the Section 4.1. Figure 3.11 shows the platform part in charge of efficiently providing the notification

system with all the required information for each message received. It is easy to see that this platform section uses data from both the Silver stage, which contains the data ingested from the relational database, and the Silver stage, which contains the messages received from the sensors.

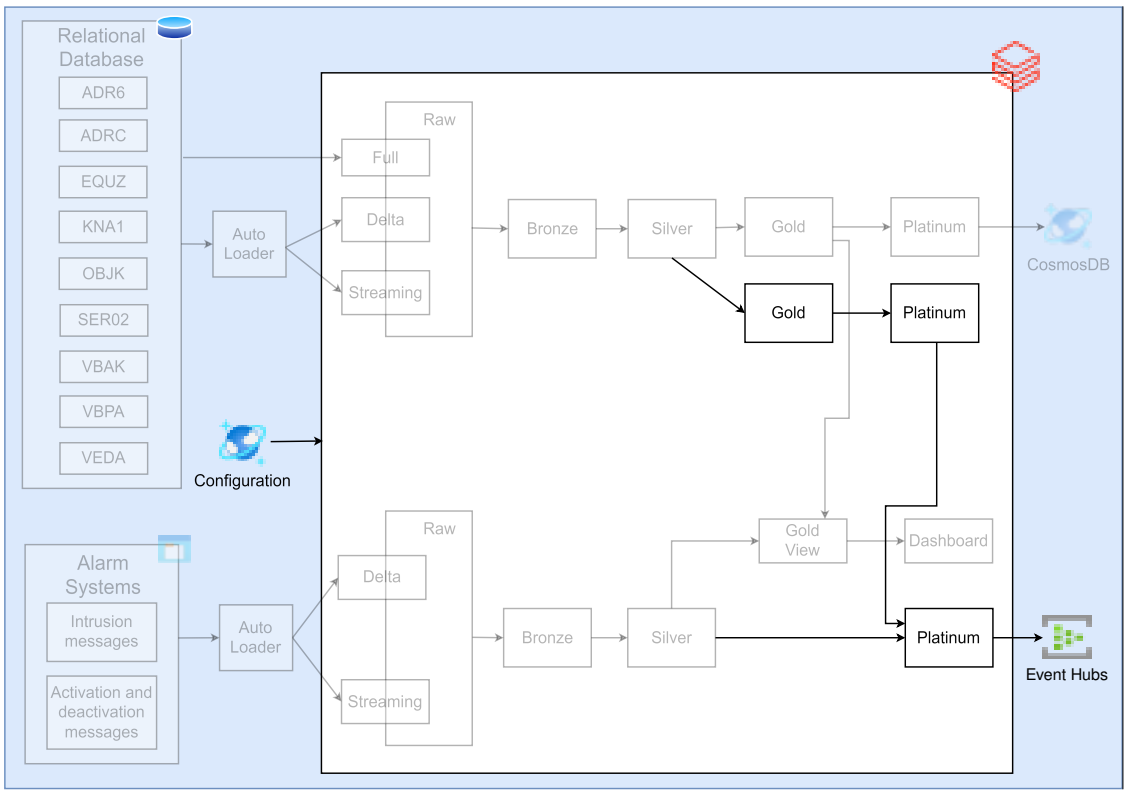

Figure 3.11: The figure highlights the data platform section responsible for delivering a real-time external notification service for messages received from the alarm plants.

Figure 3.12 shows the platform part in charge of providing the aggregated data ready to be joined with the messages received from the IoT sensors network. It works like the platform part explained in Section 3.4.3. Only three modifications are done.

- Into the Gold stage, the data about the contract are filtered to keep only the unexpired contracts.
- Into the Platinum stage, `_sub_documents_factory` method is not used, and the data are joined classically.

- At the end of the Platinum stage, the data are not written on CosmosDB but are persisted in a Platinum table ready to be used.

The activities of this platform part are orchestrated with a job that is triggered periodically. The considerations about the trigger interval and the job structure are the same as those that orchestrate the platform's activities explained in Section 3.4.3.

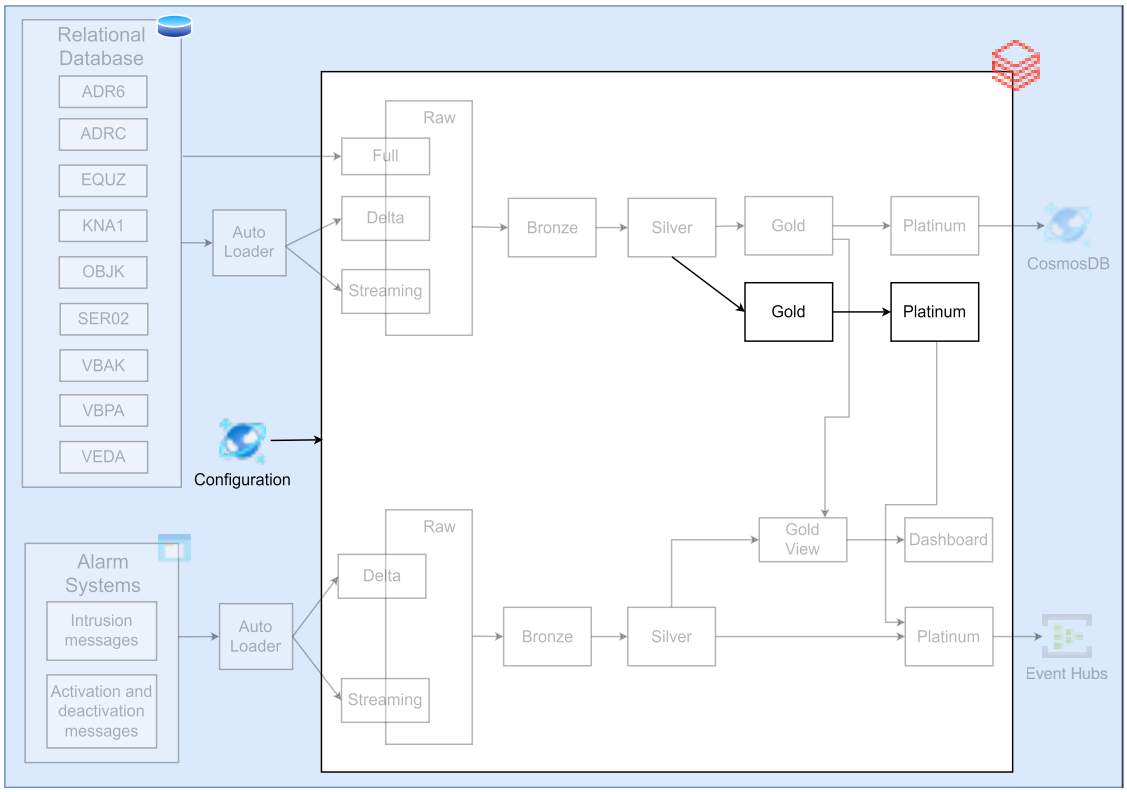

Figure 3.12: The figure highlights the data platform section responsible for providing aggregated information ready to be joined with the messages received from the sensors.

Figure 3.13 shows the data platform part that merges the messages with the information provided by the previous section explained and wrote the final product over the Event Hubs to provide the notification system.
The data transformations of this Platinum stage are performed in streaming using the SSP framework. Because the system manages two types of messages, two streaming data flows were built. These two flows act very similarly, and the code structure is the same, so only one is explained.
The logic was defined in a notebook in which is defined a class that inherited

from the primitive class `PlatinumMessagesPrimitive` explained in Section 3.4.4. This class defines all the parameters useful for interacting with a specific Event Hub. The name and key of the Event Hub access policy were not written into the code as this is critical information, but the names of the secrets stored in the Key Vault containing these values were defined. The class also defines a method to build the streaming and a method to use in the `foreachBatch` SSP framework method.
The method to build the streaming recalls two primitive methods. The first is the one to read the new messages from the silver table. The second is the one to make the other setup of the streaming parameter, such as triggers and checkpoints. As the last step, the class method uses `foreachBatch` SSP method to join the message with the aggregated information made available by the platform section explained before and shown in Figure 3.12. The `foreachBatch` calls the primitive method to write the data over a specific Event Hub instance.
A main notebook reads the working parameters of the job, instantiates a class for each message type, and runs the class's method to build and run the streaming activities. The main notebook also defines the trigger interval. It is set to a very small amount of time because a real-time notification system was required.
A job was defined to orchestrate the activities of this part. In this case, the job has only one task that runs the main notebook. Obliviously, the global orchestration mechanism manages the activities in combination with the other platform parts.

3.4.5 Dashboard

The last business need chosen to be addressed consists of providing a dashboard to visualize some data. In the specific case was chosen to visualize the trend over the years and the total number of alarm fires to particular clients. The clients visualized are the ones with the highest number of alarms fired over the selected years. The number of clients and the year interval can be chosen dynamically by interacting with the dashboard. Figure 3.14 shows the resulting dashboard with the text boxes to configure the parameters. The dashboard uses default parameters defined in the back end if no values are entered in the text boxes.

Figure 3.15 shows the data platform part in charge of providing the dashboard service. In this case, the data sources are the silver table, which contains the received alarm messages, and some Gold tables, which contain

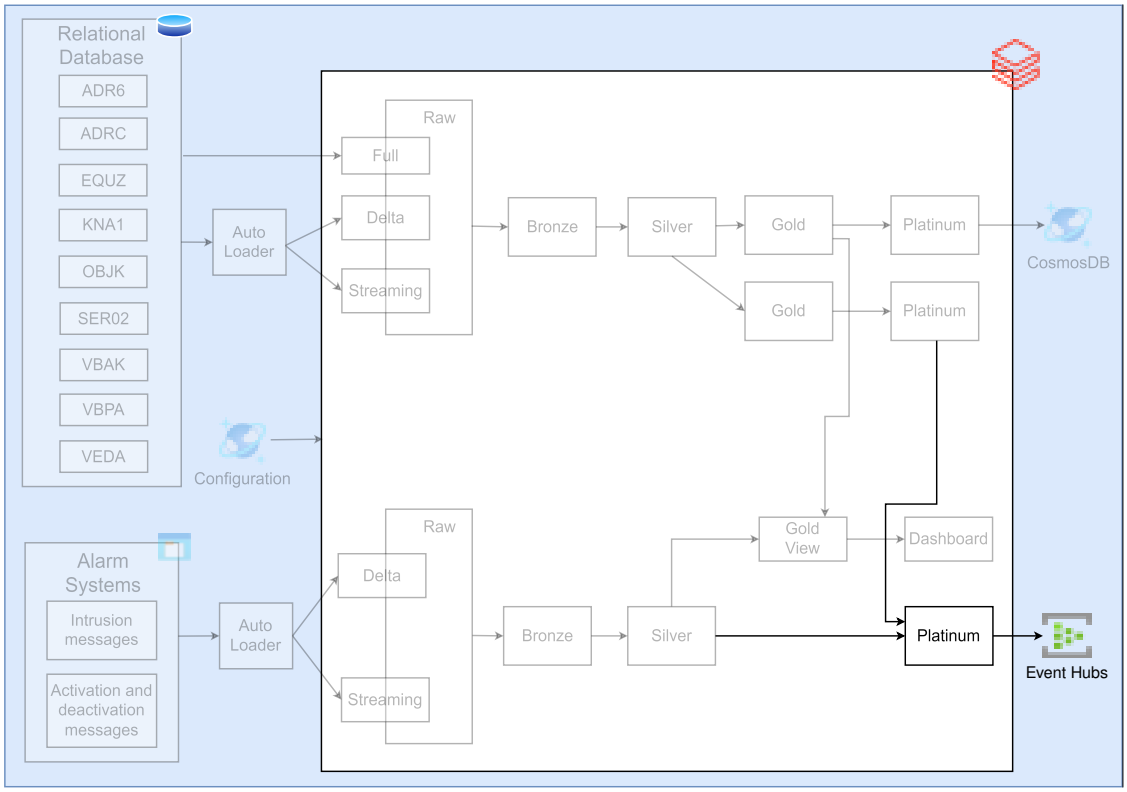

Figure 3.13: The figure highlights the data platform section responsible for taking the messages, joining them with the aggregated information provided by the platform part of Figure 3.12, and writing to the Event Hub.

the data to be merged in order to associate the alarm with the client. As shown in Figure 3.15, are used the Gold tables of the data stream explained in Section 3.4.3. This led to an optimization of performance, which was possible because the visualization needs less up-to-date data than is required for Cosmos documents. Another reason is that the necessary tables with the required fields are already available in the Gold stage.

SQL code was used to provide the visualization. In particular, a Gold view was registered in the Metastore. It defines the data aggregations to have the number of alarms fired for each client and year. These data are not stored but are dynamically retrieved at runtime when the view is queried. The choice of using a view instead of a table resides in the fact that these data are required a few times in a year, and it is not possible to know a priori when scheduling the updates, so establishing a process that periodically updates the data can lead to a waste of the computational resources.

Figure 3.14: The figure shows the dashboard that presents the trend and the total number of alarms fired over the years by single clients. The clients visualized have the highest number of alarms fired over the selected years. The dashboard provides text boxes in which it is possible to set parameters to customize the visualization dynamically. In particular, it is possible to choose the year interval and the number of clients to visualize. The figure shows the graphical interface with which the users can interact with the visualization.

The dashboard was built with the Databricks dedicated tool. It has a graphical interface through which the developer can define what is visualized in the dashboard: in simple words, the graphical interface allows the definition of what is shown by Figure 3.14: type of visualization, title, axis, colors, legend, and the text box into insert the parameter values. The dashboard tool also has

a part in which it is possible to define SQL queries with which to retrieve the data to visualize. In the thesis work, only a query was defined to retrieve and filter the data from the Gold view. The filter operations are performed on the basis of the parameters that can be set into the dashboard interface shown in Figure 3.14. Ultimately, the dashboard is published and can be accessed through a URL. The access can be set to public or restricted to specific users. Of course, the second access permissions was chosen. Orchestrating the

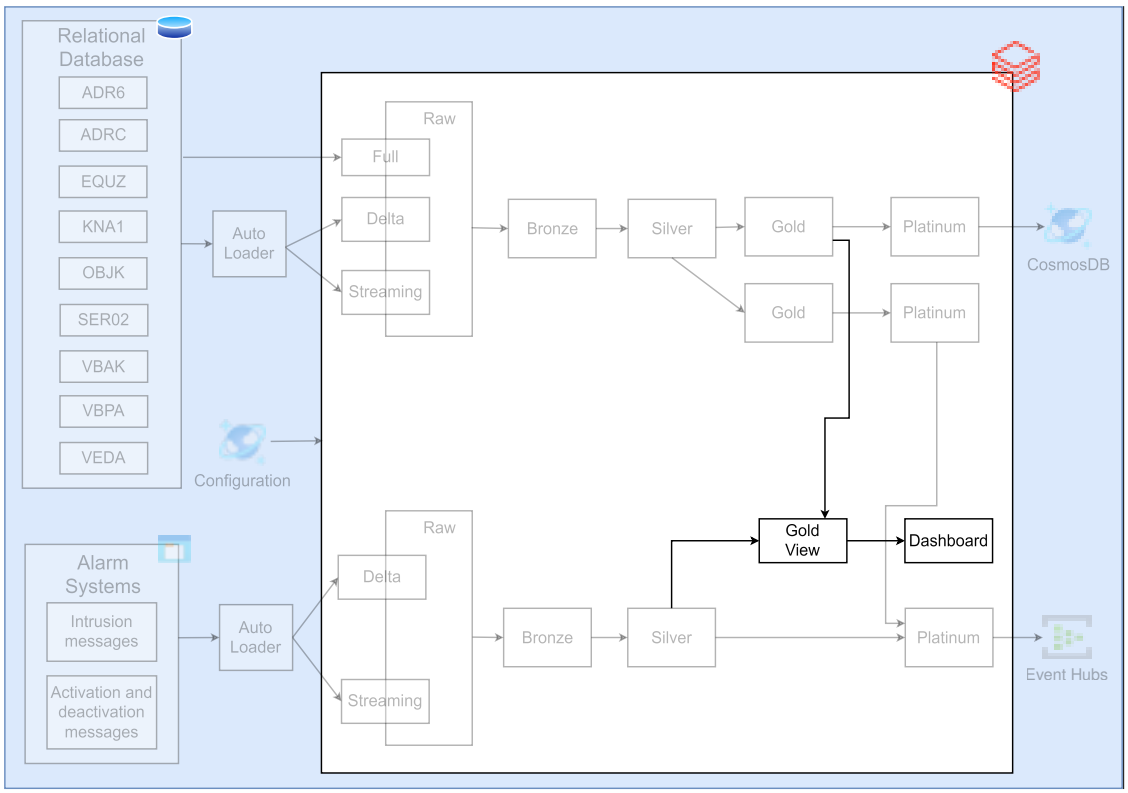

Figure 3.15: The figure highlights the data platform section that is in charge of providing a visualization fully managed by Databricks.

activities explained in this section does not require a job; the data and the visualization are updated when a user requires them through the user web interface, and Databricks fully manages everything.

As said before, the dashboard is interactive; Figure 3.16 shows the result of a redefinition of the year's interval and a selection done over the bar plot about which clients are shown. All these interactions can be done directly by the authorized users without changing the back end.

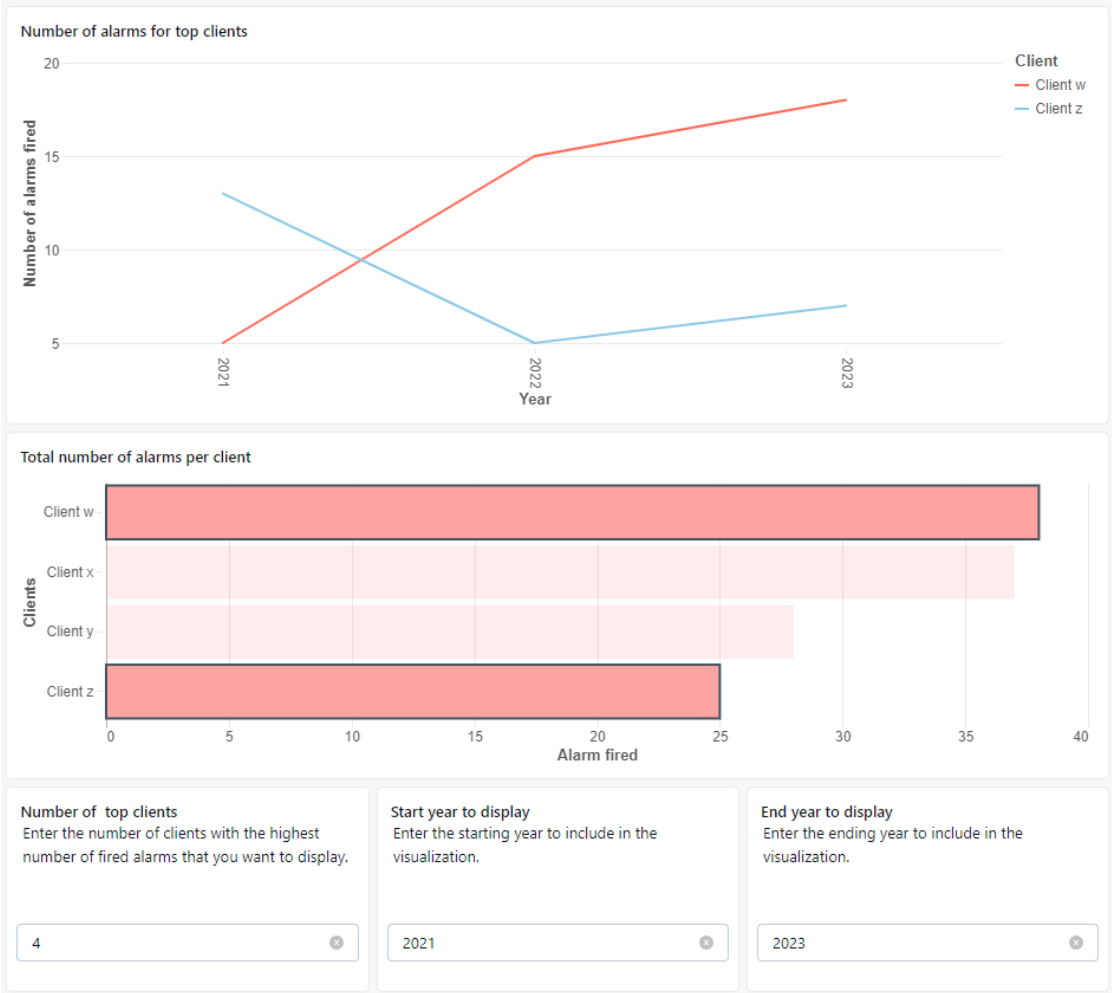

Figure 3.16: The figure shows the dashboard of Figure 3.14 in which were personalize the parameters and selected the first and the last of the top four clients referred to the years interval 2021 - 2023.

3.5 Orchestration mechanism

The global orchestrator mechanism coordinates all platform activities to ensure data integrity by configuring, executing, and properly stopping the tasks. The orchestrator consists of a Python notebook that makes REST API calls to the platform jobs. The calls are done to the Databricks REST API [34]. An API call can configure, run, or stop a job depending on the needs. In this notebook, also the global parameters are defined. They are the storage account containers that store the checkpoints, the backups, and the containers that

memorize the table files. These parameters are used to configure the jobs.
The orchestrator notebook was divided into several sections, each of which addressed a specific situation that could happen. Of course, not all possible situations were addressed due to the prototype nature of the work, but only a subset of them to show the feasibility of having a single orchestrator that can be used by people who do not know the platform structure deeply but need to have an easy tool to manage the current operations without broke the system or the data integrity.
The orchestrator use also supporting tables registered into the Metastore to track what was run.

For example, a notebook section allows to perform the full load of the tables ingested by the relational database (platform part explained in Section 4.1). It consists of a configuration part that defines: the relational tables involved, from which medallion stage performs the operation, and the type of full to be done in the Silver stage. Then, a loop iterates over the table and performs the following operations for each table. Access the supporting table and find if a job run exists for the table and which operation it has been done. The supporting table tracks the job `run_id` and the load type of the last job run for each table; when the type operation is full, it is also recorded from which stage it was carried out. Through an API call with the `run_id`, it finds the state of the job (running or finished). When the preceding job run is finished, an API call does the job configuration and runs it. The job `run_id`, the load type, and the medallion stage from which the operation is performed are recorded in the supporting table. On the other hand, two situations can happen if the preceding job run is not concluded. If the preceding run was done in streaming mode, an API call stops the run, and then a new run is configured and started, and the run's information is recorded in the support table. In the case in which the preceding run was a full load, two other situations can happen: if the preceding full was done from the same medallion stage as the new full or from a subsequent stage, is doing exactly the same operations as the case in which the preceding run is nonfull. In the case in which the previous full run was done from a preceding medallion stage with respect to the new full, nothing is done to ensure the data integrity, and a message is printed to inform the user.
Another section of the orchestrator allows running the jobs to ingest the relational tables in streaming mode. In this case, for each table, it is checked if a running job already exists; the type of it is not important. In case a running job exists, nothing is done, and a message is printed. In the other case, the job is properly configured and run, and its data are recorded in the

supporting table.
Constantly referring to the relational table ingestion activities, a section also exists to stop the jobs running in streaming mode. The jobs running in full mode can be interrupted only by a new full, done from the same or a preceding medallion stage. This decision ensures the data integrity without requiring to building a more complicated orchestrator.
Analog mechanisms there exist to manage the other platform sections.

A section also exists to check which jobs are running and to find the state of a specific job. For example, it is possible to insert the name of a relational table and check if a job is running and the type of operation it performs.

Due to the prototype nature of the thesis project, it was decided to guarantee data integrity between the ingestion part and the business logic in a hard way, and a part to be left to the user. All business logic activities are stopped before making a full load in the ingestion platform part. Then, it is the user's responsibility to make the full operations in the business logic part after the conclusion of the full ingestion activities. Of course, all the operations can be done using the orchestrator notebook.

Chapter 4

Performances

This chapter analyzes platform performances by exploiting both vertical and horizontal cluster scalability. Different test types were built to test the different platform sections.

Table 4.1 gives the configuration of the three clusters used. The cluster having a single node with 14 GB of memory and four cores was chosen as the baseline. The other two clusters were chosen to exploit horizontal and vertical scalability. Both reach the same amount of memory and cores, but in the first configuration, the resources are spread between four nodes, while in the second, they are concentrated in a single node. Both clusters possess a total resource allocation exactly four times that of the baseline cluster. The DBU/h column of the table indicates the metric used by Databricks to measure the amount of processing power consumed per hour by a cluster. DBU/h is used to compute the hourly cost of using the cluster. The relation between DBU/h and economic costs is linear. All the tests were independently repeated five

Cluster type	**Node number driver + workers**	**Memory [GB] per node**	**Cores per node**	**DBU/h cluster**
DS3_v2_1	1	14	4	0,75
DS3_v2_4	4	14	4	3,00
DS5_v2_1	1	56	16	3,00

Table 4.1: The characteristics of the clusters used to make the platform tests. The column *DBU/h cluster* reports the metric used by Databricks to compute the hourly cost of the cluster. The metric has a linear relation with the final economic cost.

times. All the values reported in the tables of the following sections were computed using the Microsoft Excel functions.

- AVERAGE function: It returns the arithmetic mean of the arguments [35]. It was used to compute the mean values obtained from the five experiments.

- STDEVA function: It estimates standard deviation based on the sample. The standard deviation measures how widely values are dispersed from the average value. It uses the formula $\sqrt{\frac{\sum(x-\overline{x})^2}{n-1}}$ where n is the sample size, and $\overline{x}$ is the sample mean [36]. It was used to estimate the standard deviation based on the sample.

- CONFIDENCE.T function: It returns the confidence interval for a population mean, using a Student's t distribution. [37]. It was used to compute the confidence interval of the sample mean based on the estimated standard deviation. In all the tests, the confidence interval was computed with the 90% confidence level, and the sample always had a size equal to 5.

4.1 Ingestion Relational Tables

This section explains the types of tests over the platform part in charge of ingesting the tables from the company's relational database (Section 4.1) and presents the results.

The test was designed to compare the performance of different cluster configurations that exploit horizontal and vertical scalability. It involved ingesting 1 million records in full mode from a single table. Due to the architecture used in this platform part, the test consists of ingesting all the rows in a single microbatch. The metrics data were taken directly from the streaming performance metrics shown by Spark SSP at the end of the batch processing, and they consist of the following.

- *Trigger Execution*: It is the time that it takes to plan and execute the microbatch [38].

- *Add Batch*: It is the time taken to execute the microbatch, excluding the time used by Spark to plane it [38].

- *Processed Rows Per Second*: It is the aggregate rate at which Spark processes the data [38]. It is useful to understand the data flow that is possible to process.

The test was performed for each stage in order to find what was the most expensive. The procedure was the following: upload 100 thousand CSV files containing 1 million records (10 thousand records for each file) in the storage account. Then, the full load was executed in the Raw stage, then in the Bronze stage, and, in the end, in the Silver stage. So, the execution metrics data were recorded, and the test was repeated five times for each cluster configuration. The metrics data take into account only the time to ingest the records. The time required to set up the streaming activity and delete the eventual data already present in the table's stages was not considered. In other words, the overhead time needed to set up the platform for executing the activity was not considered. The test did not consider ingesting multiple tables in parallel because it was tricky to synchronize the different activities with the chosen platform design. So, it was decided to perform strict tests about the ingestion of a single table and avoid tests that can result in misleading results because of bad synchronization.
Two tests were performed on the Silver stage. The first was the same as the other stages; the second focused on measuring performances when the new data is an update of existing records. In this case, 100 thousand updates were processed when the silver table contained 1 million records. SCD type two had been enabled for the silver table used. Over the other stages, this second test was not done because the Raw and Bronze stages always work in append mode, so no update occurs.

Table 4.2 and Figure 4.1 show the results of the tests performed over the Raw stage. The first parameter evaluated is the *Trigger Execution* that gives the total time to execute the activity. As expected, the cluster *DS3_v2_1* is the one that required more time to perform the activity. The time required by cluster DS3ž_1 is slightly more than four times longer than the time required by cluster DS3ž_4. Even from the point of view of the stability of the performances, the baseline cluster is the one that performs worse; in fact, it is associated with the widest confidence interval.
The clusters DS3_v2_4 and DS3_v5_1 have almost comparable performances, but the one that exploits the horizontal scalability is the best respect both the time required and the performance stability ensured. The cluster DS3_v2_4 also wins from the point of view of the economic costs because it has the same hourly costs as the cluster DS3_v5_1 and has a quarter of the cost respect the baseline. In the end, the cluster DS3_v2_4 wins from all the points of view in

executing the Raw activity tested.
The last consideration touched the time required for planning the activity. It can be computed as the difference between the *Trigger Execution* and *Add Batch* time. In this case, the cluster `DS3_v5_1` required less time, followed by `DS3_v2_4`, and `DS3_v2_1`. In order to assess whether this difference could affect the processing of very small microbatches, it would be necessary to carry out targeted tests. These have not been done as the difference in planning times between clusters `DS3_v5_1` and `DS3_v2_4` is very small; it is around 100 milliseconds.
The metric *Processed Rows/Second* gives the throughput that each cluster configuration can handle. Because it is a function of the total number of rows and the time required to perform the activity, the cluster `DS3_v2_4` has the highest throughput.

Cluster type	**Trigger Execution [ms]**	**Add Batch [ms]**	**Processed Rows/Second**
DS3_v2_1	94755 ± 7473 $\sigma = 7838$	92171 ± 6730 $\sigma = 7059$	10613 ± 848 $\sigma = 889$
DS3_v2_4	21147 ± 692 $\sigma = 726$	18756 ± 512 $\sigma = 537$	47338 ± 1530 $\sigma = 1605$
DS5_v2_1	26034 ± 2654 $\sigma = 2784$	23750 ± 1879 $\sigma = 1971$	38732 ± 3526 $\sigma = 3698$

Table 4.2: Performances to ingest 1 million records into the Raw stage with different cluster configurations. The test was done with the table containing information about the Client and the Located Performances. All the values were computed over five independent experiments with a 90% confidence interval.

Table 4.3 and Figure 4.2 shows the results of the tests performed over the Bronze stage. The *Trigger Execution* is the most important metric. As in the Raw stage, the baseline cluster was the one that performed worst both from the point of view of the time required and the performance's stability. On the other hand, the *DS3_v2_4* was the one that required less time amount. It also ensured good performance stability, but from this point of view, *DS5_v2_1* was the best. It is important to note that the cluster *DS3_v2_4* took slightly less than three times as long as the *DS3_v2_1* cluster to perform the stage activities. So, if we take into account only the economic costs, the baseline wins over all the others.

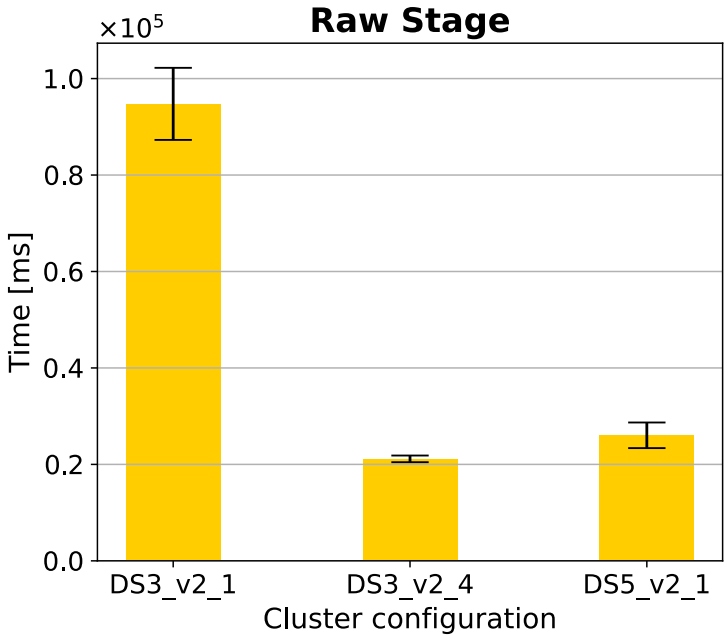

Figure 4.1: It shows the performance of ingesting 1 million records into the Raw stage with different cluster configurations. The metric shown in the visualization is *Trigger Execution*. The test was done with the table containing information about the Client and the Located Performances. All the values were computed over five independent experiments with a 90% confidence interval. The confidence was reported over the bars.

As in the Raw stage, the difference between the clusters about the time required to plane the batch was very small, and the ones that performed better under this point of view was *DS5_v2_1*.
Of course, the cluster with the highest throughput was *DS3_v2_4*.

Table 4.4 and Figure 4.3 show the results of the tests performed over the Silver stage. Also in this case the most important metric is *Trigger Execution*. As in the preceding stages the baseline cluster is the one that required more time to execute the activity. But in this case the gap with the other clusters is much wider. Compared to *DS5_v2_1* it took over four times longer and compared to *DS3_v2_4* it took, even, seventeen times longer. So *DS5_v2_1* is not only the faster, but also the cheaper cluster in which performing the activities. From the the point of view o the performances stability and economic costs the cluster *DS3_v2_4* performed much better than the others with a large gap in all respects.

Cluster type	Trigger Execution [ms]	Add Batch [ms]	Processed Rows/Second
DS3_v2_1	44780 ± 9124 $\sigma = 9570$	42752 ± 9068 $\sigma = 9512$	22987 ± 3645 $\sigma = 3823$
DS3_v2_4	16469 ± 1679 $\sigma = 1761$	14291 ± 1687 $\sigma = 1770$	61255 ± 5916 $\sigma = 6206$
DS5_v2_1	21965 ± 600 $\sigma = 630$	20028 ± 642 $\sigma = 674$	45561 ± 1204 $\sigma = 1263$

Table 4.3: Performances to ingest 1 million records into the Bronze stage with different cluster configurations. The test was done with the table containing information about the Client and the Located performances. All the values were computed over five independent experiments with a 90% confidence interval.

Cluster type	Trigger Execution [ms]	Add Batch [ms]	Processed Rows/Second
DS3_v2_1	251947 ± 25596 $\sigma = 26847$	249226 ± 25695 $\sigma = 26952$	4009 ± 446 $\sigma = 468$
DS3_v2_4	14769 ± 1184 $\sigma = 1242$	40550 ± 1149 $\sigma = 1205$	23961 ± 679 $\sigma = 712$
DS5_v2_1	59302 ± 4028 $\sigma = 4225$	57596 ± 4027 $\sigma = 4224$	16929 ± 1071 $\sigma = 1124$

Table 4.4: Performances to ingest 1 million records into the Silver stage with different cluster configurations. The DSC type selected was two. The test was done with the table containing information about the Client and the Located Performances. All the values were computed over five independent experiments with a 90% confidence interval.

Table 4.5 and Figure 4.4 report the results of the tests about updating records of a silver table. In particular, the tests consisted of updating 100 thousand records in a table containing 1 million. The data about Clients and Located Performances was used. Even in this case, the baseline and *DS3_v2_4* cluster represent the extremes in action duration, with cluster *DS3_v2_1* requiring the most time and cluster *DS3_v2_4* the least. Of course, the same was true for the throughput.
In this activity, all three clusters had wide confidence intervals; in particular,

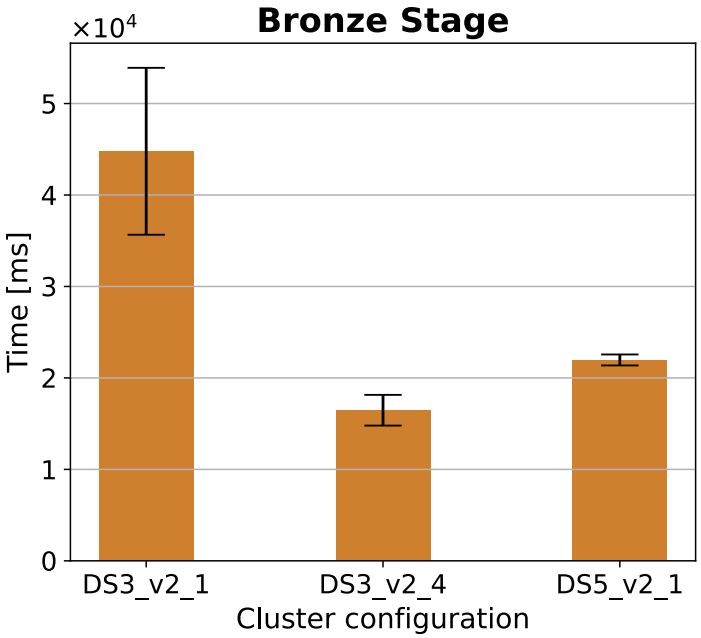

Figure 4.2: It shows the performance of ingesting 1 million records into the Bronze stage with different cluster configurations. The metric shown in the visualization is *Trigger Execution*. The test was done with the table containing information about the Client and the Located Performances. All the values were computed over five independent experiments with a 90% confidence interval. The confidence was reported over the bars.

DS3_v2_4 was the one with the widest, followed by *DS3_v2_1* and *DS5_v2_1*. In general, this activity ensured the lowest performance stability on average. Considering the economic cost, as in the Bronze stage, the smallest cluster was the most convenient because it took a little more than twice the time required by *DS3_v2_4*, but it has an economic cost that is a quarter respect the cluster *DS3_v2_4*.
The baseline cluster was also the cluster that required less time to plan the batch, but, also in this case, the differences between the three clusters were minimal.

As mentioned, three different cluster configurations were used to make the above tests. In particular, one cluster was taken as a baseline, and the two others were configured to exploit, respectively, the horizontal and vertical scalability in order to have the same amount of resources in total. Table 4.1

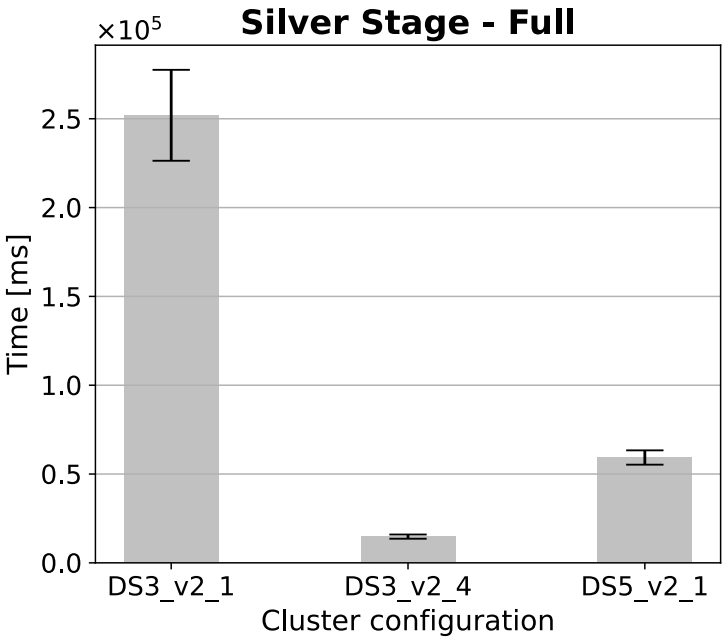

Figure 4.3: It shows the performance of ingesting 1 million records into the Silver stage with different cluster configurations. The metric shown in the visualization is *Trigger Execution*. The DSC type selected was two. The test was done with the table containing information about the Client and the Located Performances. All the values were computed over five independent experiments with a 90% confidence interval. The confidence was reported over the bars.

reports the cluster configurations. With the test result performed in this Section, it is possible to conclude that horizontal scalability is more advantageous in terms of computational time. It can be explained by the fact that all the activities performed in the Raw, Bronze, and Silver stages required access a lot to the secondary memory. So, the resources of a single node play an important role, but it is also important to divide the effort over the secondary memory.
Regarding the stability of the performances across multiple runs, it is not possible to find a configuration that always had significantly better performances compared to the others. In general, the basic configuration gave the worst results, but between horizontal and vertical scalability, it was different for each treated case.
From the economic cost point of view, it is not possible to find a configuration

Cluster type	Trigger Execution [ms]	Add Batch [ms]	Processed Rows/Second
DS3_v2_1	139529 ± 13955 $\sigma = 14638$	138867 ± 13893 $\sigma = 14572$	722 ± 64 $\sigma = 68$
DS3_v2_4	61250 ± 18066 $\sigma = 18949$	60567 ± 17911 $\sigma = 18786$	1727 ± 363 $\sigma = 381$
DS5_v2_1	71196 ± 13783 $\sigma = 14457$	70530 ± 13779 $\sigma = 14453$	1450 ± 268 $\sigma = 282$

Table 4.5: Performances to update 100 thousand records into a Silver table in which 1 million records were already stored. The SCD type selected was two. The test was done with the table containing information about the Clients and the Located Performances. The test was done with different cluster configurations, and all the values were computed over five independent experiments with a 90% confidence interval.

that always works better or worse. Also, in this case, it depends on which activity is considered.

The two most important metrics to consider when choosing cluster configurations are computational time and economic costs. In their base, it is possible always to exclude the choice of a cluster that scales vertically for the tasks analyzed in this Section. The reason is that the cluster *DS5_v2_1* shares the same hourly cost as *DS3_v2_4*, but it requires more time for all activities. The choice between the other two depends on the specific task and by which metrics it was decided is the most important. It is not a technical decision.

From the above tests, it can also be concluded that the Silver stage is the most expensive. It performs the most intensive computations. Indeed, it is in charge of historicizing the data. In particular, the tests were conducted with the most complex historicization type available in the system: SCD of type two. The complexity of the operation regards especially the cases in which it is required to perform record updates. Table 4.5 shows the time to update only 100 thousand rows, but the time required by each cluster configuration was very much greater than the time required to perform all the other actions that involved 1 million of new records.

The less expensive stage is Bronze. This can be explained by the fact that the Bronze stage applies only schema to the data. Moreover, it reads that data from a limited number of CSV files due to the source files being already read by the preceding stage and automatically compacted by the framework

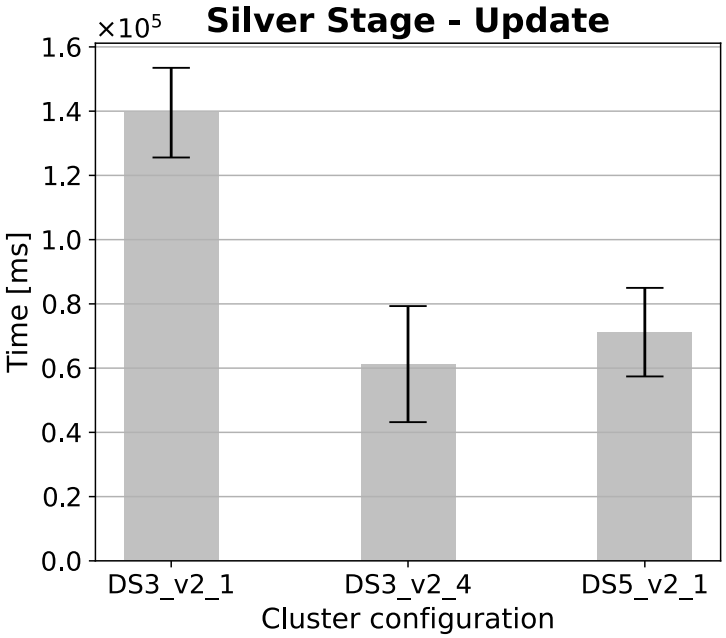

Figure 4.4: It shows the performance of updating 100 thousand records into a Silver table in which 1 million records were already stored. The SCD type selected was two. The test was done with the table containing information about the Clients and the Located Performances. The test was done with different cluster configurations, and all the values were computed over five independent experiments with a 90% confidence interval. It was reported over the bars. The metric shown in the visualization is *Trigger Execution*.

into fewer files.

4.2 Aggregated data

This section deals with the tests carried out on the platform part explained in Section 3.4.3. In this case, only a type of test was performed. It measured the time required to compute the aggregated documents to write on CosmosDB, starting from the data stored in the Silver layer. In particular, it was chosen to measure the time required to compute 1 million aggregated documents of type Client (Figure 3.7 shows the document structure). Each client was associated with one physical address and two email addresses. So, before starting the tests, 1 million client records, 1 million physical address records, and 2 million email address records were loaded into the platform up to the

Silver layer. The test considers the time required by the Gold and Platinum stages. It did not consider the time to write the documents on Cosmos because this time depends only on the Cosmos configuration and not on the cluster configuration. The test was repeated five times for each cluster configuration shown in Table 4.1. The data were reported with the 90% confidence interval. As explained in Section 3.4.3, the aggregation activities are performed by running the Gold and the Platinum stages in sequence. So, the time to make the operation was measured in the following manner by modifying the source code of both stages a little. When a record enters the Gold stage, the system attaches the actual timestamp to it. Another timestamp was attached when the aggregated document was ready to be sent to Cosmos. When the activities ended, the smallest entered timestamp, and the largest exit timestamp were found, and was computed the difference between them; this is the time considered to perform the actions. Of course with this approach it was also included the time required by the platinum main notebook to set up the activity. Obviously, it is possible to compute the activity time also in other manners, but the one chosen represents the good balance between not taking into account all the Job and Spark overhead and not establishing a complex test system.

Cluster type	**Gold and Platinum [ms]**	**Standard deviation [ms]**
DS3_v2_1	76545 ± 2463	2583
DS3_v2_4	34161 ± 2392	2509
DS5_v2_1	34435 ± 2257	2367

Table 4.6: Performances to create 1 million aggregated documents of type Client to write on CosmosDB. Figure 3.7 shows the document structure. It was decided to assign to each client two email addresses. The part of the platform involved is explained in Section 3.4.3. The test was done with three different cluster configurations, and all the values were computed over five independent experiments with a 90% confidence interval. The test did not take into account the time needed to write the documents on CosmosDB.

Table 4.6 and Figure 4.5 show the results of the tests. As expected, also in this activity, the baseline cluster was the one that performed worst both from the point of view of the time required and performance stability. The cluster configuration that exploits vertical scalability had the best performance in terms of both velocity and stability. In any case, the performances difference between cluster *DS5_v2_1* and *DS3_v2_4* is very small. The fact that cluster

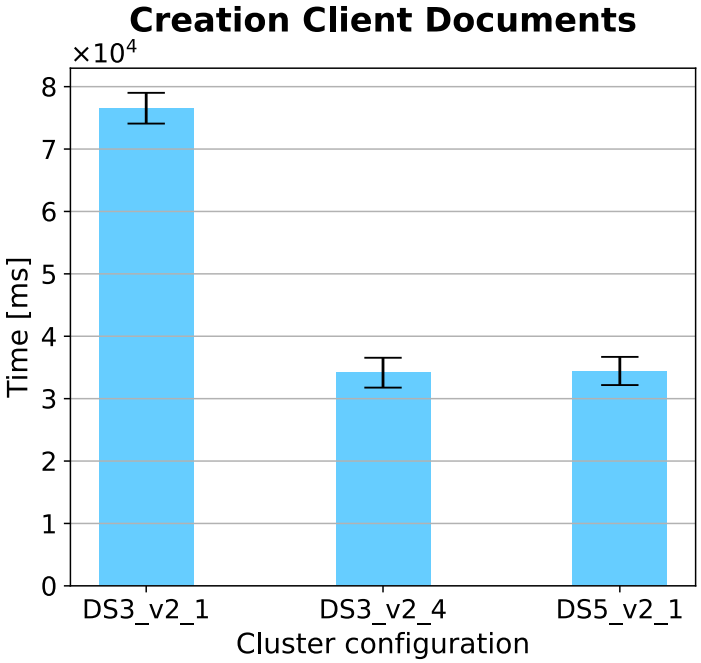

Figure 4.5: It shows the performance of creating 1 million aggregated documents of type Client to write on CosmosDB. Figure 3.7 shows the document structure. It was decided to assign to each client two email addresses. Section 3.4.3 explains the part of the platform involved. The test was done with three different cluster configurations, and all the values were computed over five independent experiments with a 90% confidence interval. It was reported over the bars. The test did not take into account the time needed to write the documents on CosmosDB.

DS5_v2_1 performed a little better than cluster *DS3_v2_4* can be explained by the fact that this activity required a lot of joint operations in the Platinum stage, and this type of operations is difficult to parallelize and required to send many data between the cluster nodes. So, if the cluster is composed of a single node, all the data are already in it.
From the point of view of the economic cost, the baseline cluster was the cheapest. Its hourly cost is a quarter of the cluster *DS5_v2_1*, but it required a little more than twice the time respect *DS5_v2_1*.

As said before, the above tests did not take into account the time to write the documents on CosmosDB because it does not depend on the cluster configuration but on the configuration of the CosmosDB's container being written to. As said in Chapter 2, the CosmosDB containers can scale up

and down manually or automatically in response to the requests; obviously, the scale leads to an increase or decrease in economic costs. For the thesis project, it was decided to take a fixed amount of resources and not exploit the Cosmo's scalability to keep the costs under control. Moreover, the scale up or down of a container has a linear relation with the increase or decrease of the performances because the parameter on which it is possible to take action is a measure of the container throughput. It is the Request Units per second, or in the short term, the RU/s.
A container with $RU = 5000$ was used to perform the tests explained in this Section. With this configuration, writing 1 million Client documents required $3780 \pm 14[s]$. The data was obtained by writing the documents in an empty container in five independent experiments. Even in this case, the data was reported with the 90% confidence interval and it is associated with a standard deviation $\sigma = 14[s]$.

4.3 Notification system

The last platform parts to test are the ones explained in Sections 3.4.4 and 3.3.2.

Unlike the other platform parts, the test, in this case, took into account the whole data flow: from the ingestion to the final data destination. This choice was made because the business need that was chosen to be satisfied with the data received from the alarm pants is a pseudo real-time notification system. So, it was fundamental to evaluate the whole data processing.

The test was designed to evaluate the time required by the system to ingest, process, and deliver the notification for a small group of messages received in the same time instant. In particular, it was chosen to process twenty messages of type activation and deactivation. No modifications to the source code were necessary to perform the test.
The test was conducted in the following manner. First, the streaming activities were run: the DLT pipeline of Section 3.3.2, and the main notebook of Section 3.4.4. The DLT pipeline was run with trigger continuous, and the SSP activity was run with a trigger interval equal to $1[s]$. The two activities were run over two cluster instances of the same type. Then, the twenty messages had been moved into the storage account container from which the streaming activity took the data. Thus, the messages were found from the Azure web interface when they arrived at the destination Event Hub. To calculate the total processing time, the ingestion timestamp assigned to each message by the

Raw stage of the DLT pipeline was utilized, along with the message's arrival timestamp in the Event Hub. Of course, the smallest ingestion timestamp and the largest arrived timestamp were used to compute the time required.
The experiment was repeated five times for each cluster configuration shown in Table 4.1.
A Python script was used to move the messages from another container of the same Azure storage account to ensure that all twenty messages arrive simultaneously on the storage account container.
It was decided to use only twenty messages because this system does not need to elaborate a high throughput, but it was very important the time in which the whole elaboration is carried out.
The same test was conducted with twenty messages of fired alarms to assess the impact of the historicization stage performed over the activation and deactivation messages. As said in Section 3.3.2 for the fired alarm messages, no historicization is performed. This last test was conducted only with a type of cluster configuration because the goal was to find the difference between making or not historicization. Of course also, in this case, the test was repeated five times.

The notification delivery also comprises the stages shown in Figure 3.12 that prepare the data to join with the messages. These stages were not tested because their work does not have an impact on the time to deliver the notification. On the other hand, they operate similarly to the one tested in Section 4.2 but are less complex due to the data are simply joined and are not used in the `_sub_documents_factory` method.

Cluster type	**Notification time [ms]**	**Standard deviation [ms]**
DS3_v2_1	22119 ± 3347	3510
DS3_v2_4	16623 ± 800	839
DS5_v2_1	16457 ± 415	436

Table 4.7: Performances to process twenty messages received from the alarm plants simultaneously. The process involved the platform parts explained in Sections 3.3.2 and 3.4.4. All the messages referred to the activations and deactivations of the plants. The test measured the time taken to notify the Event Hub since the system received the messages. The test was repeated with three different cluster configurations. All the values were computed over five independent experiments with a 90% confidence interval.

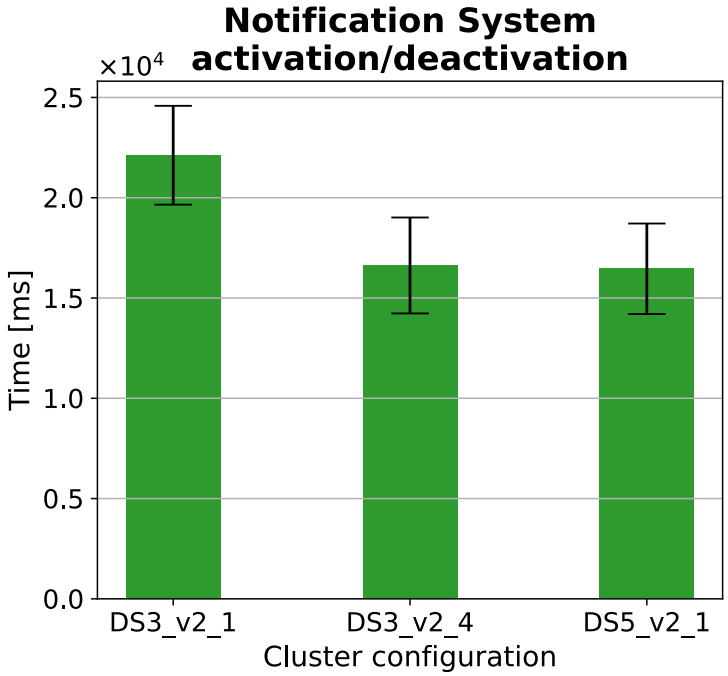

Figure 4.6: It shows the performances of processing twenty messages received from the alarm plants simultaneously. The process involved the platform parts explained in Sections 3.3.2 and 3.4.4. All the messages referred to the activations and deactivations of the plants. The test measured the time taken to notify the Event Hub since the system received the messages. The test was repeated with three different cluster configurations. All the values were computed over five independent experiments with a 90% confidence interval. It was reported over the bars.

Table 4.7 and Figure 4.7 show the time to deliver the notifications from the messages received resulting from the experiments. With respect to the results of Sections 4.1 and 4.2, the three cluster configurations have not very different performances both in time required and in performance stability. This can be explained by the fact that the number of messages to process was limited, so the scalability did not lead to significant improvement in the performance. In any case, the cluster configuration that ensured faster elaboration was *DS5_v2_1*, and the one that ensured the worst was *DS3_v2_1*. This is not surprising. Cluster *DS3_v2_4* performed worst respect cluster *DS5_v2_1*, but this difference is so small that it is impossible to find a specific reason. Only more accurate and varied tests may be found if this difference is only a case linked to the runs variability or if a specific reason exists.

From the point of view of the costs, the baseline configuration won because it employed a few more times than the other configurations.
In conclusion, the choice of cluster configuration should balance the need for faster data processing with the cost the company is willing to incur.

Message type	Notification time [ms]	Standard deviation [ms]
Activation Deactivation	22119 ± 3347	3510
Alarm	6741 ± 688	721

Table 4.8: Performances to process twenty messages received from the alarm plants simultaneously. The process involved the platform parts explained in Sections 3.3.2 and 3.4.4. The test compared the time to process twenty messages of type *Activation/Deactivation* and twenty of type *Alarm*. The test was done with the cluster configuration *DS3_v2_1*. The test measured the time taken to notify the Event Hub since the system received the messages. All values were computed over five independent experiments with a 90% confidence interval.

Table 4.8 and Figure 4.7 show the results of the tests to compare the elaboration of twenty messages of type activation/deactivation with respect to the elaboration of twenty messages indicating the firing of an alarm. The tests were conducted with the cluster configuration *DS3_v2_1*.
It is easy to see that the fired alarm messages' elaboration required more than three times less than the elaboration of the other messages. Due to the two types of messages carrying the same amount of information, the reason behind the difference in time elaboration can be explained by the historicization stage that does not exist for the fired alarm messages as explained in Section 4.1.

4.4 Final considerations

In the thesis work, only the cluster configurations shown in Table 4.1 were taken into account, but Databricks gives many more available to address different needs. The work demonstrated that scale-up led always to better performances. Which type of scale-up is better to use depends on the single situation. However, it can be extrapolated as a general rule for which it is better to scale horizontally when access to the second memory significantly

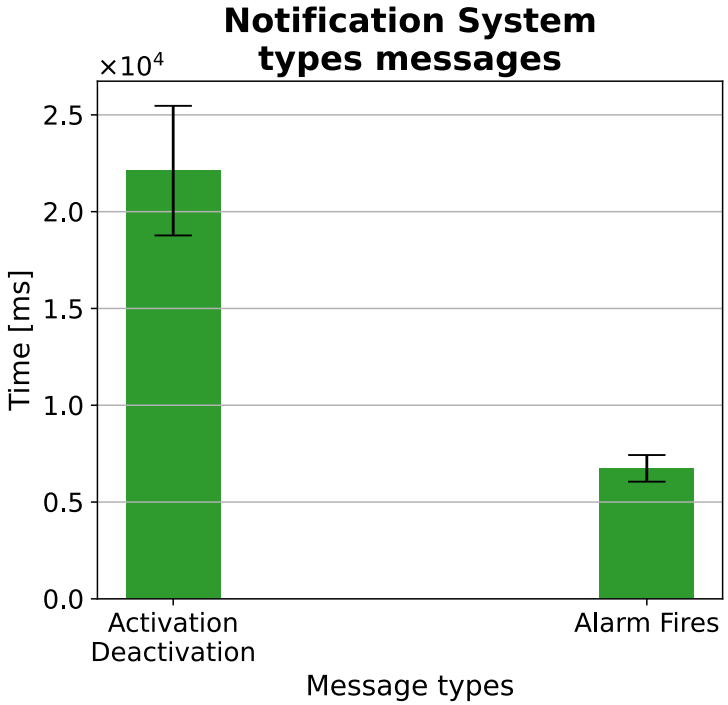

Figure 4.7: It shows the performances of processing twenty messages received from the alarm plants simultaneously. The process involved the platform parts explained in Sections 3.3.2 and 3.4.4. The test compared the time to process twenty messages of type *Activation/Deactivation* and twenty of type *Alarm*. The test was done with the cluster configuration *DS3_v2_1*. The test measured the time taken to notify the Event Hub since the system received the messages. All values were computed over five independent experiments with a 90% confidence interval. It was reported over the bars.

impacts the overall performance. In the other cases, as expected, the vertical scale-up gives better performances. Of course, vertical scalability has structural limitations because it is impossible to concentrate up to a certain amount of resources into a single machine. On the other hand, the horizontal scalability is virtually unbounded. It also demonstrated the scale-up and down with Databricks. It is very easy and fast and does not require the modification of any code line.

Of course, in the final production system, it will be necessary to fine-tune which cluster configuration is the best for each task. It depends on the amount of data the system must elaborate and the time the activity can take. Obviously, the economic costs that the company wants to incur will also be considered.

Due to Databricks' versatility, it will also be possible to choose different cluster configurations depending on the load type. For example, a big and costly cluster could be used in cases of full load, and a small and economical configuration could be used for daily activities.

Chapter 5

Future developments

As explained in the previous chapters, the platform described in this thesis work is a prototype designed to demonstrate the feasibility of creating a platform capable of ingesting data from various sources, processing it to add value, and making it available to meet different business needs. The prototype also demonstrates the possibility of achieving this in a configurable and modular way to ensure high maintainability and versatility. This final chapter proposes some possible developments within and beyond the platform, addressing data ingestion activities and new business requirements.

5.1 Expand the number of tables ingested

The first natural step in evolving the platform from a prototype to a fully operational platform for ingesting and processing the company's data requires handling the data of all tables of the company's relational database. This modification concerns the platform part explained in Section 4.1. Given the system's modular nature, it is sufficient to create the respective configuration documents in the dedicated Cosmos database and adjust the global orchestrator (Section 3.5). The change to the global orchestrator involves allowing the launch of new instances of the job that orchestrate the ingestion activities: one new instance for each new table from the relational database; the remaining changes in the behavior are handled automatically through the configuration, without requiring changes to the source code.
The next phase of this extension will involve creating new aggregates to be written to Cosmos; in other words, it means extending the platform explained in Section 3.4.3. In this case, modifications to the source code will be necessary to define the fields of interest and how to organize the aggregates.

However, it will not be required to modify the primitive classes used. In the end, new jobs should be created and integrated into the global orchestrator. The integration in the orchestrator ensures data integrity and defines appropriate triggers for the activities.

5.2 Handle more messages

A second natural evolution and extension of the platform involves increasing the types of messages managed. Currently, only activation and deactivation messages for systems and intrusion alerts are managed. In reality, alarm systems send many more messages, such as malfunctions, battery replacement requirements, battery depletion of a specific sensor, lack of power supply for wired plants, and other message types. Additionally, the maintenance technicians report the kind of intervention, its status, and other data to manage and track their work. The security company also offers a private surveillance service. For alarm systems with a surveillance contract, a guard is dispatched in response to an intrusion alert; after the intervention, the guard reports the type of intervention and whether the intrusion was real or false. Currently, these data are not processed but could be in the future using the platform components described in Section 3.3.2. Here as well, source code changes would be minimal, with most of the work handled at the configuration level. Based on the speed requirements for ingesting these data, a decision would need to be made on whether to use the same Delta Live Tables pipeline described in Section 3.3.2 or to use other DLT pipelines. In the first case, this choice would mean processing all messages in real-time since the DLT framework does not allow different triggers within the same pipeline. Alternatively, opting for the second strategy would require defining one or more new DLT pipelines to assign to specific message groups, allowing specific triggers to be set according to the message type. For example, final maintenance intervention reports do not require real-time processing and can be processed at intervals defined according to the business needs. Naturally, these new activities would also need to be incorporated into the platform's global orchestration mechanism.

5.3 Make predictions over false alarms

Once a substantial data history has been collected from the alarm systems and surveillance feedback indicating whether alarms were true or false, it

becomes possible to develop a predictive system for assessing the veracity of individual alarms received. For each received alarm, the system would generate a prediction on the likelihood that an intrusion has truly occurred, aiming to optimize the deployment of guards in situations where many systems report intrusions but it is not feasible to immediately send a guard to each location. After completing the check, the guard records whether the alarm was true or false on the platform. The system then uses these data to improve itself periodically.

The predictive system can be developed directly within Databricks using the machine learning tools available, or it can be developed on external platforms and integrated with the alarm notification system. In the latter case, Databricks would be responsible for providing the necessary data in the appropriate format. In both cases, completely new code would need to be developed, but the process uses the data of the Silver stages, so the ingestion processes would remain untouched, and modifications would only apply to subsequent stages.

Chapter 6

Conclusions

The thesis consisted of building a cloud-based data platform for a security company. The primary objectives were to make the platform configurable, modular, easily maintainable, scalable, secure, resilient, fault-tolerant, and versatile, allowing an easy management system accessible even to users with only a high-level understanding of data flows.

The configurability was obtained by developing code that can be customized runtime. Simply put, the code contains many parameters that change hit behavior depending on their values. The code parameters can be customized in two ways: through the parameters of the Databricks jobs and through JSON-like documents read runtime. In the second case, the parameter values are stored in JSON-like documents memorized in a NoSQL database. During the execution and depending on the activity, a procedure reds a specific document, extracts the values, and personalizes the code behavior.

Both class inheritance and code configurability were exploited to ensure modularity and easy maintainability. Class inheritance enabled the reuse of methods across different platform parts. Code configurability allowed the same source code to handle different processes of the same type but over different data. Simply put, the configuration tailors the code behavior to the specific data. In addition, the configuration allows different types of data ingestion to be performed over the same data stream.

Using a managed cloud service together with Databricks allows the easy scalability of the platform to address the change in the data volume and computational time required. The tests conducted over the platform showed the impact of the horizontal and vertical scalability over the time required to execute the data transformations and the economic costs. The best configuration

depends on the activities to execute and the balance between computation time and costs that were decided to achieve.

Integrating Azure services, such as the Key Vault and the Managed Identity, guaranteed security. Moreover, both Azure and Databricks allow different permission levels to be assigned to the different actors that use and work on the platform. Ultimately, all the data stored in Azure are encrypted by default.

Reliability and fault tolerance were achieved by implementing the Medallion design pattern. At each stage of the process, data are stored, resulting in data storage at varying levels of quality. For example, the raw stage stores the raw data as they have been read from the sources. This approach allows data transformation to be recomputed anytime and from any stage.

The Medallion approach also ensured the possibility of having a valuable data quality layer from which to start the development of the logic to satisfy specific business needs. In this way, an ingestion process can provide data to multiple business needs, and the developers can remain focused on satisfying the specific need without putting their attention on the ingestion phases.

The versatility of the developed platform can be found in the capability to ingest data from multiple and different types of sources. In this specific case, two sources of different types were chosen: a relation database and a network of IoT sensors. The first is a centralized data source, while the second is a spread one. Even the types of data are different. The first source gives transactional data, while the second gives JSON-like documents. The ingestion of the data from these sources involves different challenges, but from the point of view of the people who develop the task to satisfy specific business needs, the data of both sources appear in the same manner and can be used together easily. Another versatility that should be highlighted consists in the fact that the same ingestion process up to the Silver stage of the Medallion can serve multiple business needs, present and future.

Due to the prototype nature of the project, only three business needs were selected to be handled. Each covers a different area, from creating pre-aggregated data ready for use to providing real-time data processing and creating an interactive dashboard for data visualization.

To conclude, this thesis showed the potential of scalable cloud-based data platforms to address complex data management needs. By combining state-of-the-art technologies with a solid architectural structure, this project achieved its goals and gave the foundation for future developments in the data management of the security company for which it was developed. The insights and methodologies developed here can be a valuable resource not only for the company for which the system was built but also for similar

projects in the field of data science and data engineering.

Bibliography

[1] Microsoft. *Azure documentation*. https://learn.microsoft.com/en-us/azure/?product=popular. [Online] (cit. on p. 3).

[2] Microsoft. *What is Azure Virtual Network?* https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview. [Online]. June 2024 (cit. on p. 5).

[3] Microsoft. *Storage account overview*. https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview. [Online]. Aug. 2024 (cit. on p. 5).

[4] Microsoft. *Understanding block blobs, append blobs, and page blobs*. https://learn.microsoft.com/en-us/rest/api/storageservices/understanding-block-blobs--append-blobs--and-page-blobs. [Online]. June 2023 (cit. on p. 6).

[5] Microsoft. *Access tiers for blob data*. https://learn.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview. [Online]. Sept. 2024 (cit. on p. 6).

[6] Microsoft. *Azure Storage encryption for data at rest*. https://learn.microsoft.com/en-us/azure/storage/common/storage-service-encryption. [Online]. Feb. 2023 (cit. on p. 7).

[7] Microsoft. *Azure Cosmos DB documentation*. https://learn.microsoft.com/en-us/azure/cosmos-db/. [Online] (cit. on p. 7).

[8] Microsoft. *Azure Cosmos DB - Database for the AI Era*. https://learn.microsoft.com/en-us/azure/cosmos-db/introduction. [Online]. Aug. 2024 (cit. on p. 7).

[9] Microsoft. *Documents*. https://learn.microsoft.com/en-us/rest/api/cosmos-db/documents. [Online]. Nov. 2021 (cit. on p. 8).

[10] Microsoft. *Partitioning and horizontal scaling in Azure Cosmos DB*. https://learn.microsoft.com/en-us/azure/cosmos-db/partitioning-overview. [Online]. Sept. 2024 (cit. on p. 8).

[11] Microsoft. *Overview of indexing in Azure Cosmos DB*. https://learn.microsoft.com/en-us/azure/cosmos-db/index-overview. [Online]. Aug. 2024 (cit. on p. 9).

[12] Microsoft. *Choose an API in Azure Cosmos DB*. https://learn.microsoft.com/en-us/azure/cosmos-db/choose-api. [Online]. Sept. 2024 (cit. on p. 9).

[13] Microsoft. *Features and terminology in Azure Event Hubs*. https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-features. [Online]. Oct. 2024 (cit. on p. 10).

[14] Microsoft. *Azure Event Hubs: A real-time data streaming platform with native Apache Kafka support*. https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about. [Online]. Nov. 2024 (cit. on p. 10).

[15] Microsoft. *What are managed identities for Azure resources?* https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview. [Online]. Oct. 2023 (cit. on p. 11).

[16] Microsoft. *Azure Key Vault basic concepts*. https://learn.microsoft.com/en-us/azure/key-vault/general/basic-concepts. [Online]. July 2024 (cit. on p. 12).

[17] Databricks. *What is Azure Databricks?* https://learn.microsoft.com/en-us/azure/databricks/introduction/. [Online]. Sept. 2024 (cit. on p. 12).

[18] Databricks. *Azure Databricks concepts*. https://learn.microsoft.com/en-us/azure/databricks/getting-started/concepts. [Online]. Oct. 2024 (cit. on pp. 12, 14).

[19] Apache Spark. *Unified engine for large-scale data analytics*. https://spark.apache.org/. [Online] (cit. on p. 13).

[20] Apache Spark. *Spark SQL API*. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html. [Online] (cit. on p. 13).

[21] Apache Spark. *Structured Streaming Programming Guide*. https://spark.apache.org/docs/3.5.2/structured-streaming-programming-guide.html. [Online] (cit. on p. 13).

[22] Databricks. *What is Unity Catalog?* https://learn.microsoft.com/en-us/azure/databricks/data-governance/unity-catalog/. [Online]. Oct. 2024 (cit. on p. 14).

[23] Databricks. *Unity Catalog: Open, Multimodal Catalog for Data AI*. https://github.com/unitycatalog/unitycatalog/blob/main/README.md. [Online] (cit. on p. 14).

[24] Databricks. *What are catalogs in Azure Databricks?* https://learn.microsoft.com/en-us/azure/databricks/catalogs/. [Online]. June 2024 (cit. on p. 14).

[25] Databricks. *What are schemas in Azure Databricks?* https://learn.microsoft.com/en-us/azure/databricks/schemas/. [Online]. June 2024 (cit. on p. 14).

[26] Delta Lake. *Build Lakehouses with Delta Lake*. https://delta.io/. [Online] (cit. on p. 15).

[27] Databricks. *What is Delta Lake?* https://docs.databricks.com/en/delta/index.html. [Online]. Oct. 2024 (cit. on p. 15).

[28] Databricks. *What is Delta Live Tables?* https://learn.microsoft.com/en-us/azure/databricks/delta-live-tables/. [Online]. Oct. 2024 (cit. on p. 16).

[29] Databricks. *Delta Live Tables*. https://www.databricks.com/product/delta-live-tables. [Online] (cit. on p. 16).

[30] Microsoft. *What is Auto Loader?* https://learn.microsoft.com/en-us/azure/databricks/ingestion/cloud-object-storage/auto-loader/. [Online]. Aug. 2024 (cit. on p. 19).

[31] oracle. *Slowly Changing Dimensions*. https://www.oracle.com/webfolder/technetwork/tutorials/obe/db/10g/r2/owb/owb10gr2_gs/owb/lesson3/slowlychangingdimensions.htm. [Online] (cit. on p. 20).

[32] *Slowly changing dimension*. https://en.wikipedia.org/wiki/Slowly_changing_dimension. [Online] (cit. on p. 20).

[33] Databricks. *Medallion Architecture*. https://www.databricks.com/glossary/medallion-architecture. [Online] (cit. on pp. 22, 23).

[34] Databricks. *Azure Databricks REST API reference*. https://docs.databricks.com/api/azure/workspace/introduction. [Online] (cit. on p. 56).

[35] Microsoft. *Microsoft Excel AVERAGE function*. https://support.microsoft.com/en-us/office/average-function-047bac88-d466-426c-a32b-8f33eb960cf6. [Online] (cit. on p. 60).

[36] Microsoft. *Microsoft Excel STDEVA function*. https://support.microsoft.com/en-us/office/stdeva-function-5ff38888-7ea5-48de-9a6d-11ed73b29e9d. [Online] (cit. on p. 60).

[37] Microsoft. *Microsoft Excel CONFIDENCE.T function*. https://support.microsoft.com/en-us/office/confidence-t-function-e8eca395-6c3a-4ba9-9003-79ccc61d3c53. [Online] (cit. on p. 60).

[38] Databricks. *Monitoring Structured Streaming queries on Databricks*. https://docs.databricks.com/en/structured-streaming/stream-monitoring.html. [Online] (cit. on pp. 60, 61).