Università degli Studi di Padova

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

Bachelor Degree in Computer Science

Migration from a relational database to a NoSQL database for an Amazon Transportation Services internal platform

Bachelor Degree Thesis

Supervisor

Prof. Ombretta Gaggi

Bachelor Candidate

Marco Andrea Limongelli
ID: 1225415

Academic Year

2021-2022

Migration from a relational database to a NoSQL database
for an Amazon Transportation Services internal platform

Bachelor Degree Thesis

Acknowledgments

I wish to express my deepest gratitude to my supervisor, Professor Ombretta Gaggi, for all her support during the internship recognition process and during the writing of this thesis. Without her help, I wouldn't have had the chance to graduate in July.

Words cannot express my gratitude to my manager, Medha Pathak, and to my mentor Joao Patricio, for their guidance during this project and for the feedbacks which made me grow. I am also grateful to all the PerfectMile Tech team members, which made me feel part of the team since the day 1, for all the new concepts I have learnt from them.

I would be remiss in not mentioning my friend, Adnan Gazi Latif, for the healthy competition that made us aim higher and higher. You deserve so much from the future.

Special thanks to my parents for all their sacrifices to allow me to study. In particular, to my mom for making me insist on the highest standard and to my dad who taught me more things than he imagines and he is the person I'm inspired by.

Finally, I would like to thank my girlfriend, Anna Bresolini, for all the moral support in these last 2 years of study and during this internship for being close to me even though we were far away, in two different countries. I love you.

Padua, July 2022 *Marco Andrea Limongelli*

Abstract

This document describes the work carried out during the internship period for a total duration of about three-hundred hours by the student Marco Andrea Limongelli at *Amazon* in Luxembourg. The internship had as main topic the migration from a *relational database* to a *NoSQL database*.

This wasn't a side project for the team because this database is one of multiple databases that the team's core product uses to work. In fact, this project born from the needs to improve the performance of the existing product that it was no longer able to support the workload of the application. The identified solution was the migration from a *relational database* to a *NoSQL database* for the type of data I worked on.

The internship project had three main phases. The first one was to design the new database in such a way that it was able to store the same data in an efficient way. The second one was the migration of the data from the old database to the new one. The last one was to modify the existing code to rely on the new database.

Contents

Table of Figures

1 Introduction

Typographic conventions

The following typographical conventions have been adopted in this document:

- The acronyms, abbreviations and ambiguous or uncommon terms mentioned are defined in the glossary, located at the end of this document;
- The names of the technologies and other technical words are highlighted in *italics*

1.1 The proposed internship project

The company had some available internship projects with different topics. The project they proposed to me was about the migration from a *relational database* called *Amazon* Aurora to a *NoSQL database* called *DynamoDB*. The product for which I had to perform the migration uses different databases to store different type of data. My internship project focuses only on one of these databases.

This project was born from the need to improve the performance of the existing product that it was no longer able to support the workload of the application. For this reason, in order to meet the required performance to support the workload of the application, the identified solution was the migration from a *relational database* to a *NoSQL database* for the type of data I worked on.

The internship project had three main phases:

1. **New database design**: Design the new database in such a way that it was able to store the same data in an efficient way;
2. **Data migration**: Migrate the data from the old database to the new one;
3. **Modify the existing code**: Modify the existing code to rely on the new database.

1.2 Business Context

1.2.1 The Amazon Company

Figure 1.1: Amazon logo

Amazon (*figure 1.1*) is an American multinational technology company founded on July 5, 1994 which focuses on e-commerce (*amazon.com* website), cloud computing (*Amazon Web Services*), digital streaming (*Amazon Prime Video*) and artificial intelligence (*Amazon Alexa*). It is one of the Big Five companies – the most dominant and most prestigious companies in the information technology industry of the United States – alongside Google, Apple, Facebook and Microsoft.

Amazon is a very large company and in fact has 1'622'000 employees (March 2022) with hundreds of offices and operations centers around the world.

1.2.2 My team: PerfectMile Tech

Figure 1.2: PerfectMile team logo

The PerfectMile team (*figure 1.2*) is a worldwide *Business Intelligence* and *Data Integration* team composed of *Software Development Engineers*, *Business Intelligence Engineers* and *Data Engineers*. The team is based in Seattle (USA), Philadelphia (USA), London (UK), Paris (FR), Luxembourg (LU), Munich (DE), Hyderabad (IN) and Beijing (CN).

The PerfectMile team is responsible to develop the homonym product which is an end-to-end *Business Intelligence* solution encompassing datasets, *metrics*, web dashboards, pixel-perfect pdf reports and corresponding technologies enabling them. Over the past years, PerfectMile has evolved to become the authoritative source for quality, productivity, performance, and financial datasets and *metrics* for transportation businesses across Amazon worldwide.

The team is structured around two product offerings (owned by different sub-teams):

- PerfectMile Content owns collation of data from multiple transportation systems and provides Business Intelligence services to businesses;
- PerfectMile Tech provides self-serve Business Intelligence software as a service that enable content reporting (e.g. PDFs, web UI).

I worked in PerfectMile Tech as a *Software Development Engineer Intern* in the *Back-End* sub-team of PerfectMile Tech.

2 The internship project

2.1 Overview

The *Back-End* team of PerfectMile Tech, in addition to other services, develops a system called *LosPollosHermanos* (LPH) whose name is a reference to the fast food restaurant chain of the famous TV series *Breaking Bad*. This system serves the *metric* data relative to the Amazon deliveries through a serverless API to its consumers (*PerfectMile internal website*, export processes and other UIs). Currently there are ~20k *metrics* and every day on average LPH serves a total of ~8 million requests only for the *GetMetricData* endpoint which is used to get the value of a *metric*.

At present, the system is no longer able to guarantee adequate performance. The LPH API availability in Q4 2020 was 99.14%, which translates to a combined 19 hours during which the *PerfectMile website* was impacted by *metric* loading issues. The main cause of the problems was found in the current *MySQL database* (*Amazon Aurora*) which has now reached its performance limits.

Two types of data are stored in *Amazon Aurora*:

- The data related to the deliveries on which to calculate the values of the *metrics*;
- The *metadata* associated with each individual *metric*, such as the formula to be used to calculate it and other data.

The solution the team found to ensure the necessary performance is to save the deliveries data to a type of database called *Apache Druid* and to save the *metadata* associated with the *metrics* in a *NoSQL key-value* database called *Amazon DynamoDB*. In this way the team can completely dismiss *Amazon Aurora*.

2.2 The problem to solve

When I started my internship the team was working on the migration of the *metric* data to *Apache Druid*. The project they assigned to me was the migration of the *metadata* from *Amazon Aurora* to *Amazon DynamoDB*.

If the LPH *Aurora MySQL* cluster experienced an outage, then Druid read and write were impacted. That's because the *metric* information and their list of linked dimensions, were all stored in *Amazon Aurora*. For this reason we needed to decouple *Apache Druid data ingestion* and data read from *Aurora* completely. So the choice was to migrate and back-fill the *metrics metadata* to another database type. *DynamoDB* was chosen because it is designed to run high-performance applications at any scale. It stores data as an *hashmap* and then when you make a query for a specific *metric metadata*, you get a faster response than from *Aurora*.

The problem to solve was how the new *DynamoDB* database could be designed to store the *metrics metadata* in an efficient way ensuring the same LPH API functioning and how to perform the migration.

2.2.1 Constraints

I was asked to find a solution for this problem keeping in mind the following constraint:

- The new *DynamoDB* database had to store the same data without loss any information;
- The data had to be stored in the most efficient way to ensure the best possible performance to retrieve it;
- The migration didn't have to cause LPH downtime, so that meant it had to be performed while LPH was running without affecting its operations;
- No data loss during the migration phase;
- During the migration phase data had to be consistent.

2.3 Project requirements

2.3.1 Notation

The requirements will be referred to according to the following notations:

- **M** for the mandatory requirements, binding as the primary goals required by Amazon;
- **D** for the desirable requirements, non-binding or strictly necessary goals but with recognizable added value;
- **O** for the optional requirements, representing added value that is not strictly competitive.

The abbreviations indicated above will be followed by a sequential pair of numbers which identify the requirements.

2.3.2 Requirements fixed

The following objectives are expected to be carried out:

- Mandatory
  - ⇒ **M01**: Drafting of the *Design Doc* which contains the design of the new *DynamoDB* database and the migration plan;
  - ⇒ **M02**: Development of all the classes needed to perform the CRUD operations (*create*, *read*, *update* and *delete*) on the new *DynamoDB tables*;
  - ⇒ **M03**: Perform and complete the migration of all *metadata* following the migration plan in the test environment;
  - ⇒ **M04**: Drafting of the *Next Steps* document which contains the description of the remaining work to be done;
- Desirable
  - ⇒ **D01**: Modification of all the service classes which rely on *Amazon Aurora* to make them rely on the new *DynamoDB tables*;
  - ⇒ **D02**: Modification of all the existing endpoints which rely on *Amazon Aurora* to make them rely on the new *DynamoDB tables*;
- Optional
  - ⇒ No optional requirements have been identified.

3 Working methodology at Amazon

3.1 Business Processes

3.1.1 Agile methodology: SCRUM framework in Amazon

As I wrote before, the *PerfectMile* team has two sub-teams: *Tech* and *Content*. In *PefectMile Tech* they use the AGILE methodology, in particular the *Scrum framework* (*figure 3.1*), in order to always be available to the customers' needs and to continuously improve their products.

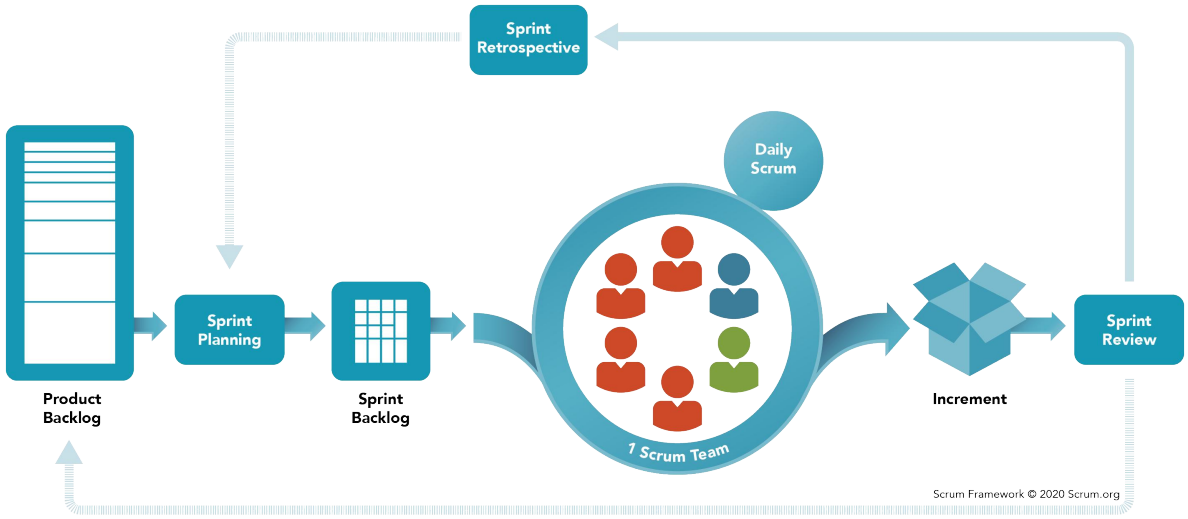

Figure 3.1: SCRUM framework

3.1.1.1 SCRUM Team

A *Scrum Team* is a collection of individuals (typically between five and nine members) working together to deliver the required product *Increments*.

The *Scrum Team* consists in:

- **One Product Owner**: The *Product Owner* is the Team member who knows what the customer wants and the relative business value of those wants. The *Product Owner* must know the business case for the product and what features the customers' wants and he is also responsible for managing the *Product Backlog*. In PerfectMile the *Product Owner* was a *Technical Product Manager*;
- **One Scrum Master**: The *Scrum Master* helps to keep the team accountable to their commitments to the business and also remove any roadblocks that might impede the team's productivity. They meet with the team on a regular basis to review work and deliverables. In my team every day because one *Development Team* member was also the *Scrum Master*;

- **The Development Team**: *Development Teams* are structured and empowered by the organization to organize and manage their own work. They are cross-functional, with all the skills as a team necessary to create a product *Increment*.

3.1.1.2 Sprint

A *Sprint* is a short, time-boxed period when a *Scrum Team* works to complete a set amount of work. *Sprint* lies at the core of the *Scrum agile methodology* and can be thought of as an *event* which wraps all other *Scrum Events* like *Daily Scrum*, *Sprint Review* and *Sprint Retrospective*. Like all of the *Scrum events*, *Sprint* also has a maximum duration. Usually, a *Sprint* lasts for one month or less. In my team we have used 3 weeks long *Sprints*.

3.1.1.3 Product Backlog

The *Product Backlog* is a continuously improved list, with the initial version listing only the most preliminary and well-known requirements (not necessary well understood). *Product Backlog* evolves based on changes in the product and development environment. The *Backlog* is dynamic and it often changes to identify what is necessary to make the product reasonable, competitive, and useful. The *Product Backlog* lists all the features, *use cases*, *user stories*, improvements, and *bug fixes* that will be made to future *releases*. The Product Backlog exists as long as the product exists.

Product Backlog Items (PBIs) are usually sorted by value, risk, priority, and necessity. It is a sequence of highest to lowest priority, with each entry having a unique order. Product to-do list entries at the top need to be developed immediately. The higher the ranking, the more urgent the product to-do list entry is, the more you need to think carefully and the more consistent your opinion on the value.

The *Product Backlog Items* that the development team will develop in *Sprint* are fine-grained and have been decomposed. The *Product Backlog Items* that the *Development Team* can complete in a *Sprint* are considered fulfilling the definition of "*ready*" and can be selected at the *Sprint planning meeting*.

3.1.1.4 Sprint Planning

Sprint Planning initiates the *Sprint* by laying out the work to be performed for the *Sprint*. This resulting plan is created by the collaborative work of the entire *Scrum Team*.

The *Product Owner* ensures that attendees are prepared to discuss the most important *Product Backlog Items* and how they map to the *Product Goal*. The *Scrum Team* may also invite other people to attend *Sprint Planning* to provide advice.

Sprint Planning addresses the following topics:

- Why is this *Sprint* valuable?
- What can be done in this *Sprint*?
- How will the chosen work get done?

3.1.1.5 Sprint Backlog

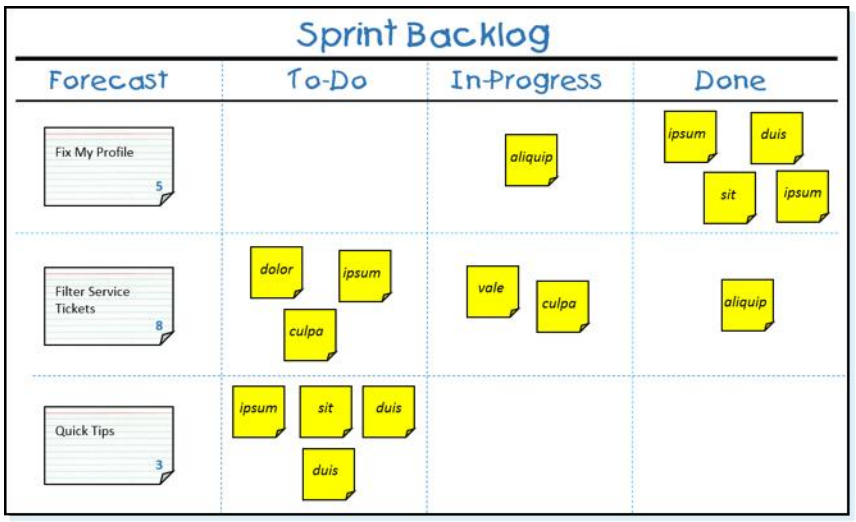

Figure 3.2: Sprint Backlog

The *Sprint Backlog* (*figure 3.2*) is composed of the *Sprint Goal* (*why*), the set of *Product Backlog Items* selected for the *Sprint* (*what*), as well as an actionable plan for delivering the *Increment* (*how*).

The *Sprint Backlog* is a plan by and for the Developers. It is a highly visible, real-time picture of the work that the Developers plan to accomplish during the *Sprint* in order to achieve the *Sprint Goal*. Consequently, the *Sprint Backlog* is updated throughout the *Sprint* as more is learned. It should have enough detail that they can inspect their progress in the *Daily Scrum*.

3.1.1.6 Increment

An *Increment* is a concrete stepping stone toward the *Product Goal*. Each *Increment* is additive to all prior *Increments* and thoroughly verified, ensuring that all *Increments* work together. In order to provide value, the *Increment* must be usable.

3.1.1.7 Daily SCRUM - Stand-Up meeting

The purpose of the *Stand-Up meeting* is to inspect progress toward the *Sprint Goal* and adapt the *Sprint Backlog* as necessary, adjusting the upcoming planned work. The *Stand-Up meeting* is a 15-minute event for the Developers of the *Scrum Team*. To reduce complexity, it is held at the same time and place every working day of the *Sprint*.

In my team the *Stand-Up meeting* was every day at 6.00 pm in order to allow other team members placed in a different time zone to attend the meeting. We used a program that randomly created a queue where all the *Development Team* members were present. Following the queue order, each team member had to talk about what he did during the working day.

3.1.1.8 Sprint Review and Sprint Retrospective

Usually, *Sprint Review* and *Sprint Retrospective* are two separated events. The purpose of the *Sprint Review* is to inspect the outcome of the *Sprint* and determine future adaptations. The purpose of the *Sprint Retrospective*, instead, is to plan ways to increase quality and effectiveness.

In my team we had a unique *Sprint Review-Retrospective* meeting that was a mix of *Sprint Review* and *Sprint Retrospective*. At the beginning of this meeting the *Scrum Master* shares to all the team members a link to the *Sprint Retrospective board* (*figure 3.3*) which contains 4 lists in which every team member can put an entry. Each list has its own topic:

- *Kudos*: In this list each team member can insert an entry to congratulate another team member for something he did during the *Sprint*;
- *Went well*: what went well during the *Sprint;*
- *To improve*: what can be improved for the next *Sprint;*
- *Action items*: which *action items* a team member or the entire team must perform.

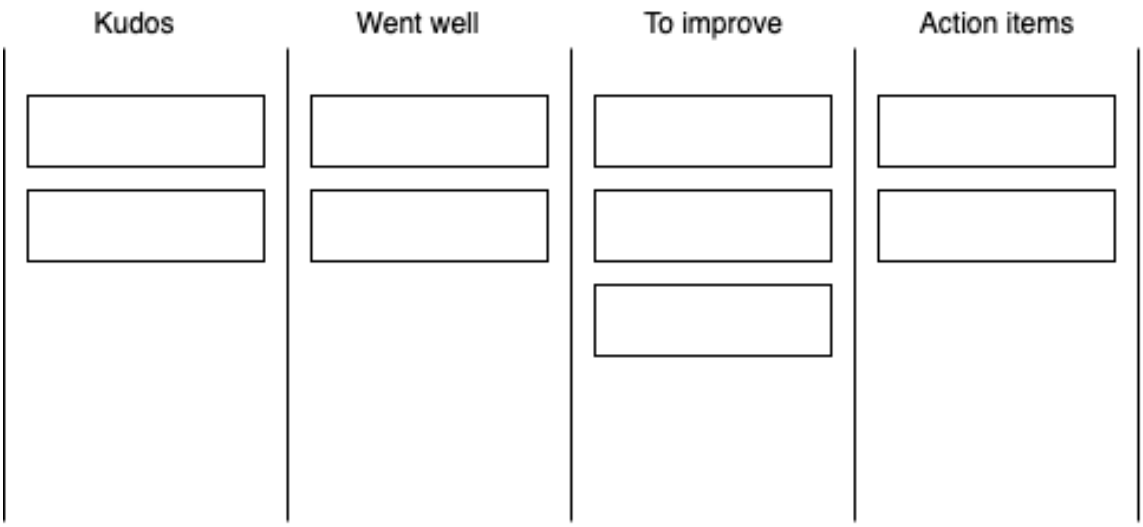

Figure 3.3: Sprint Retrospective board

After each team member has finished inserting his entries, the whole team discusses about all the entries inside the board. The discussion focuses on what was done during the past *Sprint* and how the team can improve.

3.2 Software Development in PerfectMile Tech

3.2.1 Software and Tools

In Amazon, at least in my team, each *Software Development Engineer* used to work an Apple MacBook Pro and the macOS Operative System. The typical software and tools used every day by the *SDEs* are listed below.

3.2.1.1 Amazon Chime

Figure 3.4: Amazon Chime logo

Amazon Chime (*figure 3.4*) is a communication service that the Amazon teams use to organize and conduct video meetings. It also permits to chat with other people inside the Amazon organization, but it is not used much for that.

3.2.1.2 Slack

Figure 3.5: Slack logo

Slack (*figure 3.5*) is a messaging program designed specifically for the workplace. It offers rooms organized by topics called *channels*, private groups and direct messaging. In *Slack* my team had a channel for each topic to discuss. Precisely for these functionalities it is used instead of *Amazon Chime* for direct messages and contact other team members.

3.2.1.3 Outlook

Figure 3.6: Outlook logo

Outlook (*figure 3.6*) is a *Personal Information Manager* software system. *Outlook* is primary an email client but it also includes other functions like the calendar which is very useful because it is automatically updated with all the meetings you have been invited to join or that you have created.

3.2.1.4 Amazon Meetings

Figure 3.7: Amazon Meetings logo

Amazon Meetings (*figure 3.7*) is an Amazon internal application used to schedule meetings with other people within the Amazon organization. Every time you schedule a meeting, it automatically sends an email to all the invited people and creates an event on the *Outlook* calendar.

3.2.1.5 Quip

Figure 3.8: Quip logo

Quip (*figure 3.8*) is a collaborative *Productivity Software* which allows groups of people to create and edit documents and spreadsheets as a group. It is very useful also because it allows to add comments wherever you want within the document.

3.2.1.6 IntelliJ IDEA

Figure 3.9: IntelliJ IDEA logo

During my internship I always wrote *Java* code and for this reason I used *Intellij* (*figure 3.9*) which is one of the most popular *Integrated Development Environment* - a software application used for software development which contains at least a *source code editor*, *build automation tools* and a *debugger* - for *Java*. It also supports other *Programming Languages*.

3.2.1.7 DataGrip

Figure 3.10: DataGrip logo

DataGrip (*figure 3.10*) is a database *Integrated Development Environment* that is tailored to suit the specific needs of professional *SQL* developers. I used *DataGrip* to have access to the relational database I had to work on and to do some queries.

3.2.1.8 AWS Management Console

Figure 3.11: AWS logo

AWS Management Console (*figure 3.11*) is a web application that comprises and refers to a broad collection of service consoles for managing AWS resources. It is used to manage and monitor users, service usage, health and monthly billing costs of AWS resources.

3.2.1.9 Git

Figure 3.12: GIT logo

Git (*figure 3.12*) is a free and open-source distributed *Version Control System*. It is used for tracking changes in any set of files inside a project and it is mainly used for coordinating work among programmers. Its goals include speed, data integrity and support distributed non-linear workflows (thousands of parallel branches running on different systems). The branch that most developers work against is called *mainline* and it is also the branch used to deploy in production.

Git saves different versions of the files in the .git/ folder known as *GIT repository*. This repository tracks all changes made to files inside the project folder, building a history over time. This means that if you delete the .git/ folder, you delete your project's history.

3.2.2 Deployment to production process

At Amazon, the *Continuous Integration* (CI) software development practice is used. This means that every *Software Development Engineer* regularly merges his code changes into a central *GIT* repository after run automated builds and tests. The key goals of CI are to find and address bugs quicker through frequent testing, improve software quality and reduce the time to validate and release new software updates.

The process to ship code changes in production is described in *figure 3.13*.

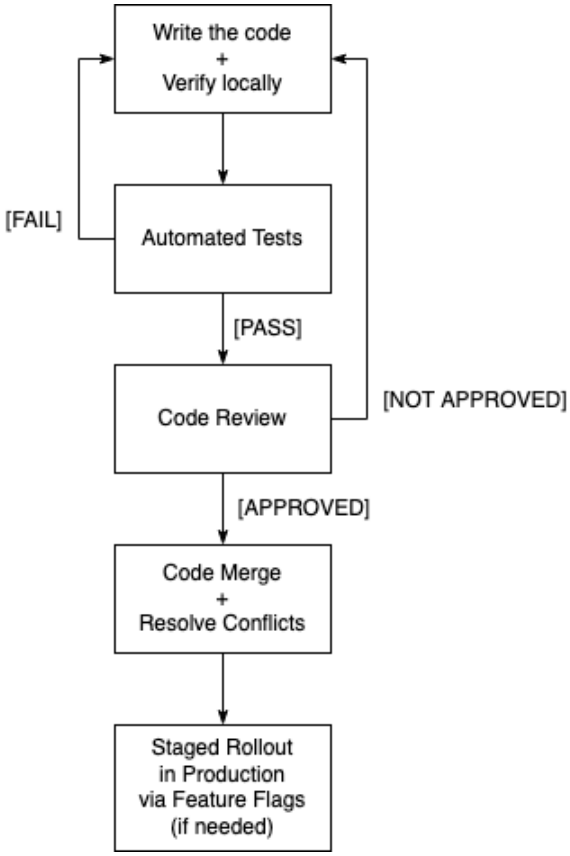

Figure 3.13: Code deployment to production process

When the *Software Development Engineers* have to push some code in the repository, first of all they must verify the new code through *unit tests*. If the *unit tests* for the functionalities they have implemented don't exist, they must create them. After passing all the *tests* locally and verifying that the entire program builds correctly without any errors, they must request a C*ode Review* to at least two team members. Before publishing the *Code Review* and sending the notification to the requested reviewers, the *Code Review Portal* automatically runs some other *tests*. If one of these *tests* fails, it means that the code has some errors or the *test coverage* - the percentage of lines covered by tests - decreased and some new *test cases* must be created. Once verified that all the *tests* passed, it is possible to publish the *Code Review* and wait the feedbacks from the team mates. Sometimes the reviewers publish some comments and the author of the code needs to address these comments and change something on the code and restart the entire process to deploy to production. Instead, if all

the reviewers approved the code changes, it is possible to push the new code in the *mainline* and eventually resolve the conflicts if needed. After that the code is pushed in the *mainline*, the changes are ready to be deployed to production. Sometimes when the code changes go to production, it's necessary to perform a *staged rollout* by using *feature flags* which are software development processes used to enable or disable functionalities remotely without deploying new code. The deployment to production of the software product where I made code changes during my internship was automatic (*Continuous Deployment*), so every time I made a code change that was approved, it went to production. In some other software products, the deployment to production required a manual approval (*Continuous Delivery*).

4 Overview of the AWS technologies

During my internship I used various *AWS technologies* to solve the problem and meet the constraints imposed on the project, so an overview of these technologies is mandatory to understand the solution design.

4.1 Amazon DynamoDB

Figure 4.1: Amazon DynamoDB logo

Amazon DynamoDB (*figure 4.1*) is a fully managed, *serverless*, key-value *NoSQL* database designed to run high-performance applications at any scale. Developers can use *DynamoDB* to build modern serverless applications that can start small and scale globally to support petabytes of data and tens of millions of read and write requests per second.

4.1.1 Features

These listed below are the main features of *DynamoDB*.

4.1.1.1 Performance at scale

DynamoDB supports *tables* of virtually any size with horizontal scaling. This enables *DynamoDB* to scale to more than 10 trillion requests per day with peaks greater than 20 million requests per second, over petabytes of storage.

4.1.1.2 Serverless

When you work with *DynamoDB* there are no servers to provision, patch or manage, and no software to install or maintain. *DynamoDB* automatically scales *tables* to adjust for capacity and maintains performance with zero administration. Availability and fault tolerance are built in, eliminating the need to architect your applications for these capabilities.

4.1.1.3 Enterprise ready

DynamoDB is built for mission-critical workloads, including support for atomicity, consistency, isolation and durability (ACID) transactions for a broad set of applications that require complex business logic. *DynamoDB* helps secure data with encryption and continuously backs up data for protection.

4.1.2 Core components

The following are the basic *DynamoDB* components:

- **Tables**: As other database systems, *DynamoDB* stores data in *tables* which are a collection of data;
- **Items**: Each *table* contains zero or more *items*. An *item* is a group of *attributes* that is uniquely identifiable among all of the other *items*. *Items* in *DynamoDB* are similar to *rows*, *records* or *tuples* in other database systems. In *DynamoDB* there isn't a limit to the number of *items* you can store in a *table*;
- **Attributes**: Each *item* is composed of one or more *attributes*. An *attribute* is an atomic element, something that doesn't need to be broken any further. *Attributes* in *DynamoDB* are similar in many ways to fields or columns in other database systems.

An example of a *DynamoDB table* which contains some *items*:

Music

Figure 4.2: DynamoDB sample table

Note the following about the *Music table* (*figure 4.2*):

- The primary key for *Music* consists of two *attributes* (*Artist* and *SongTitle*). Each *item* in the *table* must have these two *attributes*. The combination of *Artist* and *SongTitle* distinguishes each *item* in the *table* from all of the others;
- Other than the *Primary Key*, the *Music table* is *schemaless*, which means that neither the *attributes* nor their data types need to be defined beforehand. Each *item* can have its own distinct *attributes*;
- One of the *items* has a *nested attribute* (*PromotionInfo*), which contains other *nested attributes*. *DynamoDB* supports *nested attributes* up to 32 levels deep.

4.1.2.1 Primary Keys

When you create a new *table* in *DynamoDB* in addition to the *table name*, you must specify the *Primary Key* of the *table*. As in other database systems, the *Primary Key* uniquely identifies each *item* in the *table*, so that no two *items* can have the same *key*.

DynamoDB supports two types of *Primary Keys*.

Partition Key

It is a simple *Primary Key* composed of one *attribute* known as *Partition Key*.

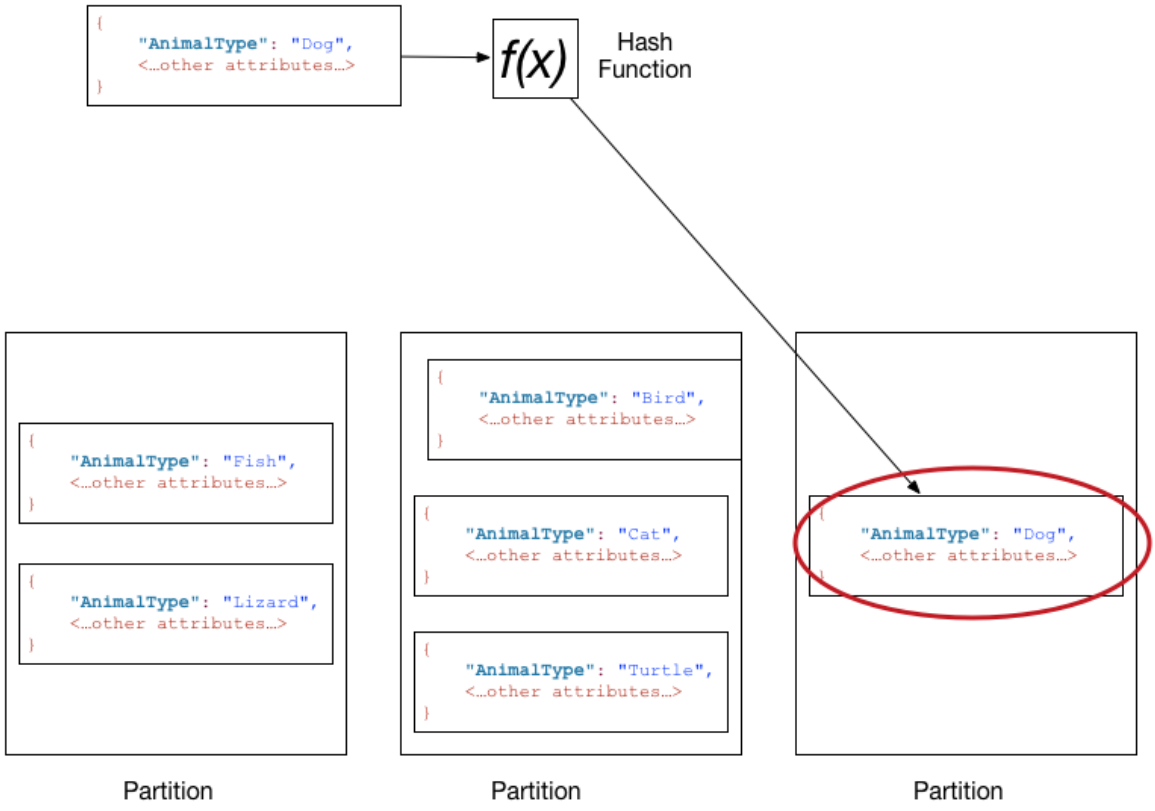

Figure 4.3: Functioning of the DynamoDB hash function for Partition Key

DynamoDB uses the *Partition Key*'s value as input to an internal *hash function* (*figure 4.3*). The output of the *hash function* determines the partition (physical storage internal to *DynamoDB*) in which the *item* will be stored. In a *table* that has only a *Partition Key* as *Primary Key* two *items* can't have the same *Partition Key* value.

Partition Key and Sort Key

Referred to as a *Composite Primary Key*, this type of key is composed of two *attributes*. The first *attribute* is the *Partition Key* and the second one is the *Sort Key*.

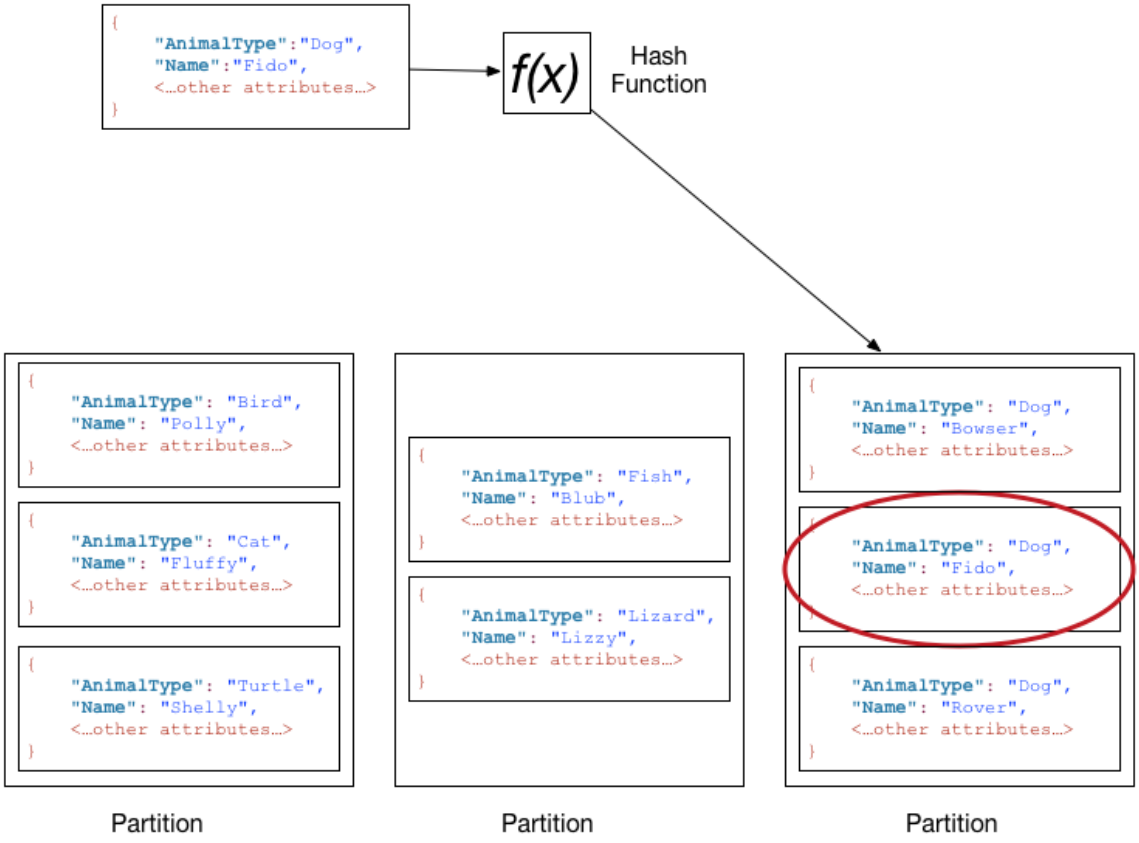

Figure 4.4: Functioning of the DynamoDB hash function for Composite Primary Key

All the *items* with the same *Partition Key* are stored in the same partition in sorted order by *Sort Key* value (*figure 4.4*). If a *table* has a *Composite Primary Key*, two *items* that have the same *Partition Key*, they must have a different *Sort Key*.

In *DynamoDB* each *Primary Key attribute* must be a scalar, so the only data types allowed for *Primary Key attributes* are *String*, *Number* or *Binary*. There are no such restrictions for other *non-key attributes*.

4.1.2.2 Data Types

DynamoDB supports many different data types for *attributes* within a *table*. They can be categorized as follows:

- **Scalar Types**: A scalar type can represent exactly one value. The scalar types are *Number*, *String*, *Binary*, *Boolean* and *Null*;
- **Document Types**: A document type can represent a complex structure with *nested attributes*, such as you would find in a JSON document. The document types are *List* and *Map*;
- **Set Types**: A set type can represent multiple scalar values. The set types are *String Set*, *Number Set* and *Binary Set*.

4.1.3 Operations for reading data from a table

Amazon DynamoDB provides different operations for reading data.

4.1.3.1 GetItem operation

Retrieves a single *item* from a *table*. This is the most efficient way to read a single *item* because it provides direct access to the physical location of the *item*. To perform this operation (*figure 4.5*), you need to provide the *primary key* (*simple* or *composite*) of the *item* you want to get.

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    }
}
```

Figure 4.5: Example of a GetItem operation

4.1.3.2 Query operation

This operation (*figure 4.6*) retrieves all of the *items* that have a specific *partition key*. Whitin those *items*, it is possible to apply a condition to the *sort key* and retrieve only a subset of the data.

```
{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    ExpressionAttributeValues: {
        ":a": "No One You Know"
    }
}
```

Figure 4.6: Example of a Query operation to get all the songs of an artist

4.1.3.3 Scan operation

This operation (*figure 4.7*) retrieves all the *items* in the specified *table*. It is possible to apply a condition to retrieve only a subset of the data.

```
{
    TableName:  "Music"
}
```

Figure 4.7: Example of a Scan operation

4.1.4 Differences from Relational databases

Obviously there are some differences between *DynamoDB* which is a *NoSQL* database and relational databases.

4.1.4.1 Schema

DynamoDB is a *NoSQL* database and is *schemaless*. This means that, other than the *Primary Key attributes*, you don't have to define any *attributes* or data types when you create *tables*. By comparison, relational databases require you to define the names and data types of each *column* when you create a *table*.

4.1.4.2 Relationships

In the *relational databases*, the *relationships* are at the foundation of how the entire database works and the design focuses on the *relationships* between *entities*. When you want to query for some data it is common to join different *tables* to retrieve the desired data.

Relationships don't exist in *DynamoDB* and it isn't allowed to join *tables* when making a query. Therefore, in *DynamoDB* you need to design the database in a different way respect in a *relational database*. That's why in *DynamoDB* is allowed to store also *Document* and *Set* types.

4.1.4.3 Client's interaction

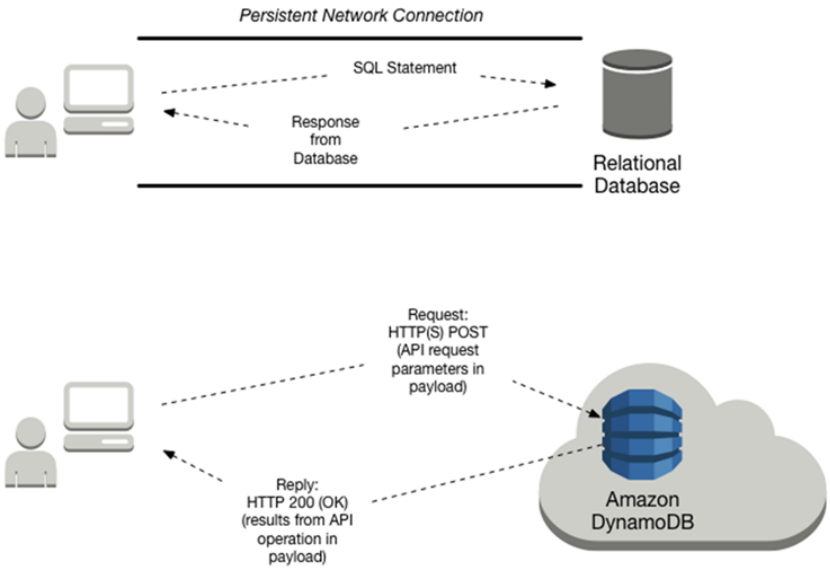

Figure 4.8: Differences between DynamoDB and the relational databases on Client's interaction

On a *relational database* you need a persistent connection (*figure 4.8*) which ends when the application is finished. Instead, in *DynamoDB* you don't have a persistent connection: interaction with *DynamoDB* is stateless and it occurs using HTTP(S) requests and responses.

4.2 AWS Lambda

Figure 4.9: AWS Lambda logo

AWS Lambda (*figure 4.9*) is a *serverless*, *event-driven* compute service that lets you run code for virtually any type of application or *Back-End* service without provisioning or managing servers.

4.2.1 Features

These listed below are the main features of *AWS Lambda*.

4.2.1.1 Build custom Back-End services

It's possible to use *AWS Lambda* to create new Back-End application services triggered on demand using the *Lambda application programming interface* (API). *AWS Lambda* processes custom *events* instead of servicing these on the client, helping you avoid client platform variations, reduce battery drain, and enable easier updates.

4.2.1.2 Completely automated administration

AWS Lambda manages all the infrastructure to run code on highly available, fault tolerant infrastructure, freeing you to focus on building differentiated backend services. With *Lambda*, you never have to update the underlying *operating system* (OS) when a patch is released, or worry about resizing or adding new servers as your usage grows.

4.2.1.3 Automatic scaling

AWS Lambda invokes your code only when needed, and automatically scales to support the rate of incoming requests without any manual configuration. There is no limit to the number of requests your code can handle. *AWS Lambda* typically starts running your code within milliseconds of an event. Since *Lambda* scales automatically, the performance remains consistently high as the event frequency increases. Since your code is *stateless*, *Lambda* can start as many instances as needed without lengthy deployment and configuration delays.

4.2.1.4 Orchestrate multiple functions

Build *AWS Step Functions* (*chapter 4.3*) workflows to coordinate multiple *AWS Lambda functions* for complex or long-running tasks. *Step Functions* lets you define workflows that trigger a collection of *Lambda functions* using sequential, parallel, branching, and error-handling steps. With *Step Functions* and *Lambda*, you can build stateful, long-running processes for applications and backends.

4.2.2 Lambda concepts

4.2.2.1 Function

A *function* is a resource that you can invoke to run your code in *Lambda*. It has code to process the *Events* passed into the function;

4.2.2.2 Trigger

A *Trigger* is a resource or configuration that invokes a *Lambda function*. *Triggers* include *AWS services* that you can configure to invoke a function and *event source mappings*. An *event source mapping* is a resource in *Lambda* that reads *items* from a stream or queue and invokes a function.

4.2.2.3 Event

An *Event* is a JSON-formatted (*figure 4.10*) document that contains data for a *Lambda function* to process. The runtime converts the event to an *object* and passes it to your function code. When you invoke a function, you determine the structure and contents of the *Event*.

```
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

Figure 4.10: JSON-formatted document

4.2.2.4 Concurrency

Concurrency is the number of requests that a *function* is serving at any given time. When the *function* is invoked, *Lambda* provisions an instance of it to process the event. When the *function* code finishes running, it can handle another request. If the *function* is invoked again while a request is still being processed, another instance is provisioned, increasing the *function's concurrency*.

4.3 AWS Step Functions

Figure 4.11: AWS Step Functions logo

AWS Step Functions (*figure 4.11*) is a low-code, visual workflow service that developers use to build distributed applications, automate IT and business processes, and build data pipelines using *AWS services*. Workflows manage failures, retries, parallelization, service integrations, and observability so developers can focus on higher-value business logic.

4.3.1 Features

AWS Step Functions provides *serverless* orchestration for modern applications. Orchestration centrally manages a workflow by breaking it into multiple *steps*, adding flow logic, and tracking the inputs and outputs between the *steps*. As your applications execute, *Step Functions* maintains application state, tracking exactly which workflow step your application is in, and stores an event log of data that is passed between application components. That means that if networks fail or components hang, your application can pick up right where it left off.

4.3.1.1 Built-in service primitives

AWS Step Functions provides ready-made *steps* for your workflow called *states* that implement basic service primitives for you, which means you can remove that logic from your application. *States* can pass data to other states and microservices, handle exceptions, add timeouts, make decisions, execute multiple paths in parallel, and more.

4.3.1.2 AWS service integration

Using AWS Step Functions Service Integrations, you can configure your Step Functions workflow to call over 200 AWS services which includes *AWS Lambda* and *Amazon DynamoDB*.

4.3.1.3 Workflow configuration

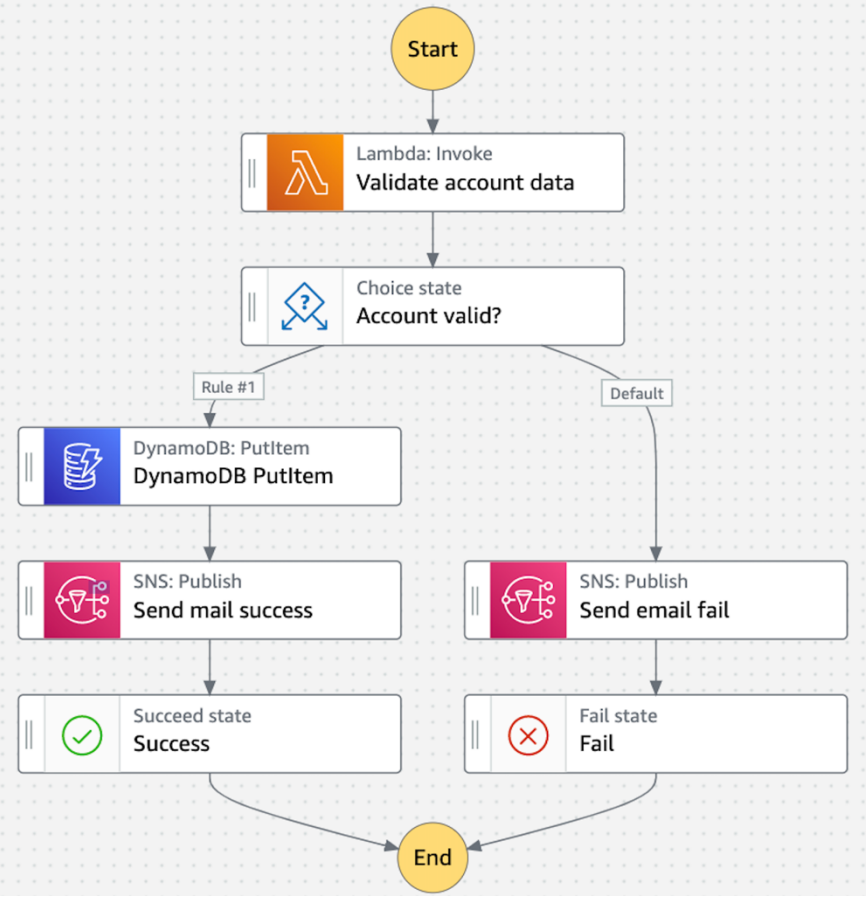

Figure 4.12: AWS Step Function workflow example

Using *AWS Step Functions*, you define your workflows as *state machines* (*figure 4.12*), which transform complex code into easy to understand statements and diagrams.

4.3.1.4 Workflow abstraction

AWS Step Functions keeps the logic of your application strictly separated from the implementation of your application. It is possible to add, move, swap, and reorder *steps* without having to make changes to the business logic. Through this separation of concerns, the workflows gain modularity, simplified maintenance, scalability, and code reuse.

4.3.1.5 Automatic scaling

AWS Step Functions automatically scales the operations and underlying compute to run the *steps* of your application for you in response to changing workloads. *Step Functions* scales automatically to help ensure the performance of your application workflow remains consistent as the frequency of requests increases.

4.3.2 States

Individual *states* can make decisions based on their input, perform actions, and pass output to other *states*. In *AWS Step Functions* you define your workflows in the *Amazon States Language.*

The following (*figure 4.13*) is an example *state* named *HelloWorld* that performs an *AWS Lambda function*.

```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
  "Next": "AfterHelloWorldState",
  "Comment": "Run the HelloWorld Lambda function"
}
```

Figure 4.13: AWS Lambda function task state example

States share some common fields:

- Each *state* must have a *Type* field indicating what type of *state* it is.
- Each *state* can have an optional *Comment* field to hold a human-readable comment about, or description of, the *state*.
- Each *state* (except a *Succeed* or *Fail* state) requires a *Next* field or, alternatively, can become a terminal state by specifying an *End* field.

4.3.2.1 Task

A *Task state* (“*Type*” : “*Task*”) represents a single unit of work performed by a state machine.

All work in a *state machine* is done by *Tasks* (*figure 4.10*). A *Task* performs work by using an activity (a process not hosted by *AWS*) or an *AWS Lambda function*, or by passing parameters to the API actions of other services. *AWS Step Functions* can invoke *Lambda functions* directly from a task state.

4.3.2.2 Pass

A *Pass state* (“*Type*” : “*Pass*”) passes its input to its output, without performing work. *Pass states* (*figure 4.14*) are useful when constructing and debugging state machines.

```
"No-op": {
  "Type": "Pass",
  "Result": {
    "x-datum": 0.381018,
    "y-datum": 622.2269926397355
  },
  "ResultPath": "$.coords",
  "Next": "End"
}
```

Figure 4.14: Pass state example

4.3.2.3 Choice

A *Choice state* (“*Type*” : “*Choice*”) adds branching logic to a state machine.

In addition to the common fields, the *Choice state* (*figure 4.15*) introduces these additional fields:

- **Choices** (required): An array of *Choice rules* that determines which *state* will be the next;
- **Default** (optional, recommended): The name of the *state* to transition to if none of the transitions in *Choices* is taken.

```
"ChoiceStateX": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.value",
      "NumericEquals": 0,
      "Next": "ValueIsZero"
    }
  ],
  "Default": "DefaultState"
},

"ValueIsZero": {
  "Type" : "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Zero",
  "Next": "NextState"
},

"DefaultState": {
  "Type": "Fail",
  "Cause": "No Matches!"
}
```

Figure 4.15: Choice state example

4.3.2.4 Wait

A *Wait state* (“*Type*” : “*Wait*”) delays the *state machine* from continuing for a specified time. It is possible to choose either a *relative time*, specified in seconds from when the state begins, or an *absolute end time*, specified as a timestamp.

In addition to the common fields, the *Wait state* (*figure 4.16*) has one of these fields:

- **Seconds**: A time in seconds to wait before beginning the *state* specified in the *Next* field. It must be an integer positive value;
- **Timestamp**: An absolute time to wait until beginning the *state* specified in the *Next* field.

```
"wait_until" : {
  "Type": "Wait",
  "Timestamp": "2022-07-22T10:59:00Z",
  "Next": "NextState"
}
```

Figure 4.16: Wait state example

4.3.2.5 Succeed

A *Succeed state* (“*Type*” : “*Succeed*”) stops an execution successfully.

Because *Succeed states* (*figure 4.17*) are *terminal states*, they haven’t the *Next* field and don’t need an *End* field.

```
"SuccessState": {
  "Type": "Succeed"
}
```

Figure 4.17: Succeed state example

4.3.2.6 Fail

A *Fail state* (“*Type*” : “*Fail*”) stops the execution of the *state machine* and marks it as a failure.

The *Fail state* (*figure 4.18*) only allows the use of *Type* and *Comment* fields from the set of common state fields. In addition, it allows the use of the following fields:

- **Cause** (optional): Provides a custom failure string that can be used for operational or diagnostic purposes;
- **Error** (optional): Provides an error name that can be used for operational or diagnostic purposes.

Because *Fail states* always exit the *state machine*, they haven’t a *Next* field and they don’t require an *End* field.

```
"FailState": {
  "Type": "Fail",
  "Cause": "Invalid response.",
  "Error": "ErrorA"
}
```

Figure 4.18: Fail state example

4.3.3 Error handling

Any state can encounter runtime errors. Errors can happen for various reasons:

- *State machine* definition issues (for example, no matching rule in a *Choice* state);
- *Task* failures (for example, an exception in a *Lambda function*);
- Transient issues (for example, network partition events).

By default, when a state reports an error, *AWS Step Functions* causes the execution to fail entirely.

4.3.3.1 Retrying after an error

Task states can have a field called *Retry*, whose value must be an array of objects known as *Retriers*. An individual *Retrier* (*figure 4.19*) represents a certain number of retries, usually at increasing time intervals.

A *Retrier* contains the following fields:

- **ErrorEquals** (required): A non-empty array of strings that match error names;
- **IntervalSeconds** (optional): An integer that represents the number of seconds before the first retry attempt;
- **MaxAttempts** (optional): A positive integer that represents the maximum number of retry attempts;
- **BackoffRate** (optional): The multiplier by which the retry interval increases during each attempt (2 by default).

```
"Retry": [ {
   "ErrorEquals": [ "States.Timeout" ],
   "IntervalSeconds": 3,
   "MaxAttempts": 2,
   "BackoffRate": 1.5
} ]
```

Figure 4.19: Step Function Retrier example

4.3.3.2 Fallback states

Task states can have a field named *Catch*. This field's value must be an array of objects, known as *Catchers*.

A *Catcher* (*figure 4.20*) contains the following fields:

- **ErrorEquals** (required): A non-empty array of strings that match error names;
- **Next** (required): A string that must exactly match one of the state machine's state names;
- **ResultPath** (optional): A path that determines what input is sent to the state specified in the Next field.

```
"Catch": [ {
   "ErrorEquals": [ "java.lang.Exception" ],
   "ResultPath": "$.error-info",
   "Next": "RecoveryState"
}, {
   "ErrorEquals": [ "States.ALL" ],
   "Next": "EndState"
} ]
```

Figure 4.20: Step Function Catcher example

Note: States.ALL is a wildcard that matches any known error name.

5 Solution Design

The purpose of this section is to explain how the *Amazon Aurora* database was previously designed, how it should work and finally explain the adopted solution for the design of the new *DynamoDB* database and the migration plan.

5.1 Amazon Aurora database schema

The *Amazon Aurora* relational database (*figure 5.1*) was designed to store the *metrics metadata*. The *metadata* are information about the *metrics* used to calculate their value.

For each *metric* the following common information are stored:

- **Name**: The *metric* name;
- **Value Type**: The value type can be *scalar*, *weighted average*, *hybrid* or *percentile*;
- **Aggregation Style**: The type of *data aggregation* can be *sum*, *last day of period* or *percentile*;
- **Chained metric and Chain cutoff date**: They are optional fields. These fields are used when someone asks for the *metric* data and the date for which he is requesting the data is before the *chain cutoff date*. In this case the system has to take the data from the *chained metric*;
- **Creation date**: The date when the *metric* was created;
- **Creator**: The *metric* creator username;
- **Deprecation date**: It is used only when the *metric* is deprecated.

Metrics have two main categories: *Primitives* which use some *dimensions* to calculate their value and *Hybrids* (*Value Type = hybrid*) which are calculated using *Primitive metrics*. For this reason, *Primitives* and *Hybrids*, in addition to the common information, have other specific information.

A *hybrid metric* also has this information:

- **Hybrid formula**: It is used only by the *Hybrid metrics* which *value type* is *hybrid*. The *hybrid formula* field contains the formula used to calculate the *metric* value;
- **Consumed metrics**: The *metrics* used by the *hybrid metric* to calculate its own value.

Instead, a *primitive metric* has this information:

- **Related dimensions**: The dimensions used by the *primitive metric* to calculate its own value.

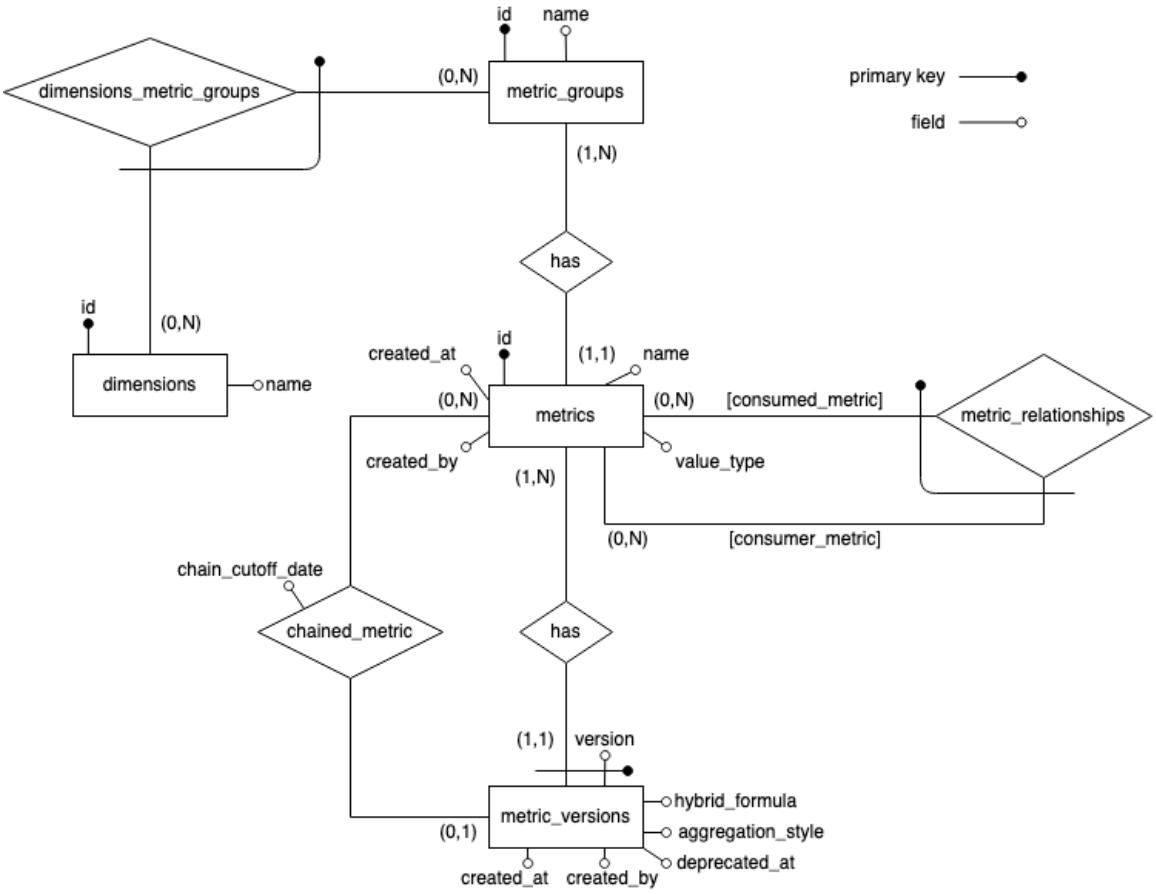

Figure 5.1: Amazon Aurora database schema

Notes about the schema:

- In *dimensions_metric_groups* that notation means that its *primary key* is composed by the foreign keys to *dimension* and *metric_groups*;
- In *metric_relationships* the *primary key* is composed by the *foreign keys* to *metrics* (*consumed_metric* and *consumer_metric* fields);
- In *metric_version* the *primary key* is composed by the *foreign key* to *metrics* and the *version* field.

5.1.1 Comments about the Amazon Aurora database schema

5.1.1.1 Metric groups and related dimensions

When the database was designed the idea was to have more *metric groups* which had to use the same *dimensions*. But what happened was the following:

- A *primitive metric* belongs to exactly one *metric group* and a *metric group* contains exactly one *metric*. The only exception is the *hybrid metrics* group;
- The *group name* is equals to the *primitive metric name*, because when someone wants to create a new *primitive metric* the *website* first does a call to the *Back-End* to create a new metric group with some *related dimensions* and after that it does a call to create the new *metric* which belongs to the new *metric group*;
- A *hybrid metric* belongs to the *hybrid metrics* group which doesn't have *related dimensions*, so the *hybrid metrics* group doesn't use the *dimensions_metric_groups table*;
- The *dimensions_metric_groups table* is used only by the *Primitive metrics*;
- The *metric_groups table* is used only to connect a single *primitive metric* to all the *related dimensions* to that *metric* since a *metric group* contains exactly one *primitive metric*.

5.1.1.2 Metric relationship

The *metric_relationships table* is used only by the *Hybrid metrics* since the *Primitive metrics* don't consume other *metrics* to calculate their value.

5.1.2 Redesign of the Amazon Aurora database schema

Before designing the new *DynamoDB* database, it is useful to redesign the *Amazon Aurora* database schema to reflect the correct behavior of LPH.

Keeping in mind the comments above, I arrived at the conclusion that it is useless to have the *metric_groups table* because it is only used to connect the *metric group* with its *related dimensions*. Since a *metric group* has a 1:1 relationship with a *metric*, it is possible to remove the *metric_groups table* and directly connect the *related dimension* to the *metrics table* with a new relationship.

It is safe to remove the *metric_groups table* because it is possible to distinguish the *Hybrid metrics* using the *value type*: when the *value type* of a *metric* is *hybrid*, it means that it is a *hybrid metric*.

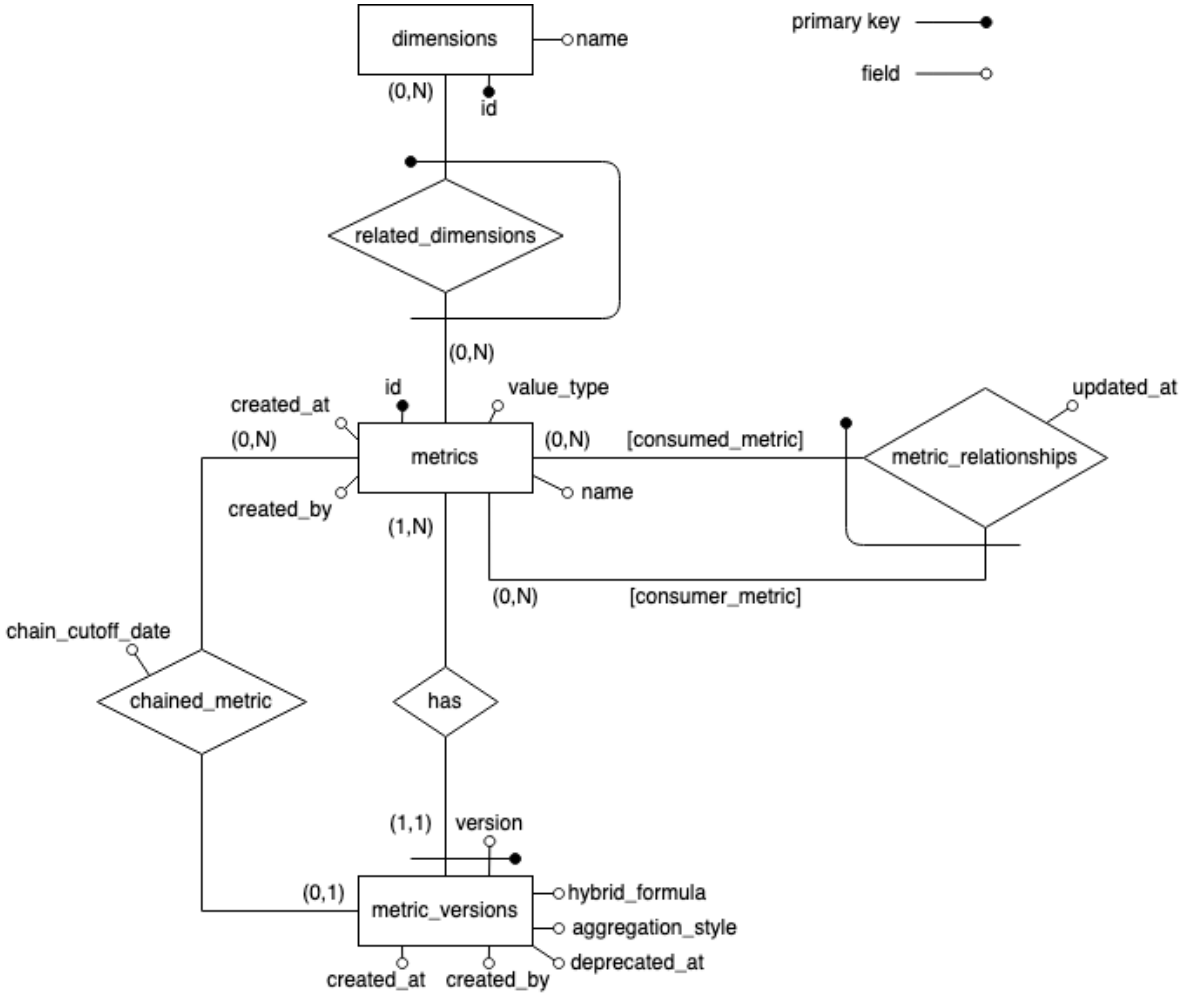

Figure 5.2: Amazon Aurora database schema redesigned

Notes about the diagram (*figure 5.2*):

- The *dimensions_metric_groups* relationship and the *metric_groups table* were removed because they were redundant;
- The *dimensions_metric_groups* relationship is substituted by the *related_dimensions* relationship between *metrics* and *dimensions*, so basically in the new *related_dimensions table* there are the same data of the *dimensions_metric_groups table* with the difference that instead of the *metric_group_id* there is directly the *metric_id*.

5.2 Designing the Amazon DynamoDB database

The design choices about the database schema made in this section were done keeping in mind the *data access pattern*, since *NoSQL* databases offers less query flexibility than their *SQL* equivalent and keeping in mind that the *DynamoDB* costs depend on the number of queries - *read* and *write* - performed. It was also important to note that using multiple *DynamoDB tables* costs because *"joins"* aren't possible and therefore multiple *reads* are required.

In *DynamoDB* the *single table design* is a best practice, so I decided to store the data in a single *table* by merging the data from the *metrics* and *metric_versions tables* and using the *version control design pattern*.

By storing the data in this way, we can get all the information about a *metric* in a single query.

5.2.1 Version control design pattern implementation

The new *DynamoDB metrics table* (*figure 5.3*) has the following characteristics:

- It has a *composite primary key* (*name*, *version*) because we want to store all the versions of a *metric* and for this reason there will be more *items* with the same *name* in this *table*;
- The *consumed metrics* and the *related dimensions* are stored as a *Set<String>*;
- The *attributes name*, *version*, *updatedAt*, *UpdatedBy*, *aggregationStyle* and *valueType* must have a value and they can't be *null*.

Field name	Type	Notes	
name	String	Partition key	Primary key
version	String	Sort key	
createdAt	Timestamp	Version 1 creation date and time - always equals and only in v0 item	
createdBy	String	Version 1 creator - always equals and only in v0 item	
updatedAt	String		
updatedBy	String		
deprecatedAt	Timestamp		
deprecatedBy	String		
deprecatedReason	String		
aggregationStyle	String	enum ['SUM', 'LAST_DAY_OF_PERIOD', 'PERCENTILE']	
chainedMetric	String	Reference to a metric name	
chainCutoffDate	Timestamp		
hybridFormula	String	Used only by *hybrid metrics*	
aggregationParam	Number		
valueType	ENUM	["SCALAR", "WEIGHTED_AVERAGE", "HYBRID", "PERCENTILE"]	
consumedMetrics	Set<String>	Used only by *hybrid metrics*. References to metrics names	
relatedDimensions	Set<String>	Used only by *primitive metrics*. References to dimensions names	

Figure 5.3: DynamoDB metrics table schema

The *version control design pattern* (*figure 5.4*) is implemented in the new *metrics table* which means that for each *metric* a *version zero item* is stored which is a copy of the *latest version item*. Storing the *version zero item* is very useful to use the *GetItem operation* because when a query for a specific *metric* is done, it's difficult to know the latest version number. So to retrieve the *latest version data*, it is only necessary to do a *GetItem operation* for the *version zero* of the *metric*, then for the *primary key* {*name* : *metricName, version* : *0*}.

Primary key		Attributes						
Partition key	Sort key							
name	version	updatedAt	updatedBy	aggregationStyle	more attributes	latestVersion	createdAt	createdBy
sample_metric	0	2022-05-17 9:35:15.0	mrcandre	SUM	[...]	5	2018-02-15 18:23:45.0	mrcandre
	1	2018-02-15 18:23:45.0	mrcandre	PERCENTILE	[...]			
	[...]	[...]	[...]	[...]	[...]			
	5	2022-05-17 9:35:15.0	mrcandre	SUM	[...]			

Figure 5.4: Items stored with the version control design pattern

Notes:

- The *version zero item* has an additional *attribute latestVersion* which contains the latest version number of the *metric*;
- The field *createdAt* and *createdBy* always refer to the creation date-time and creator of the *version 1*.

5.2.2 Type of items stored in the new table

Two types of *items* will be stored in the new *table*: one for the *Primitive metrics* and one for the *Hybrid metrics*.

5.2.2.1 Primitive metric item

The *Primitive metrics items* (*figure 5.5*) must have the *relatedDimensions attribute* (*not null*) and not the *consumedMetrics attribute*.

Primary key		Attributes				
Partition key	Sort key					
name	version	updatedAt	updatedBy	aggregationStyle	valueType	relatedDimenions
primitive_metric	1	2022-05-17 9:35:15.0	mrcandre	SUM	SCALAR	{dimension1, dimension2}

Figure 5.5: Primitive metric item example

5.2.2.2 Hybrid metric item

The *Hybrid metrics items* (*figure 5.6*) must have the *consumedMetrics* (*not null*) and the *hybridFormula* (not *null*) *attributes*. They don't have the *relatedDimensions attribute*. The *Hybrid metric items* must also have the *attribute valueType* set to *HYBRID*.

Primary key		Attributes					
Partition key	Sort key						
name	version	updatedAt	updatedBy	aggregationStyle	valueType	hybridFormula	consumedMetrics
primitive_metric	1	2022-05-17 9:35:15.0	mrcandre	SUM	HYBRID	metric1/metric2	{metric1, metric2}

Figure 5.6: Hybrid metric item example

5.3 Migration plan

Before making code changes to the LPH endpoints to rely on the new *DynamoDB table*, it is necessary migrate all the data from *Aurora* to *DynamoDB*. The problem to solve is how to migrate data without downtime and be sure that these data remain consistent during and after the migration without losing information.

Before performing the migration it is needed ensuring that all the new *metrics* added during the migration phase, will be inserted both in *Amazon Aurora* and *DynamoDB*. In this way LPH will continue to work as always, the new data will be already in *DynamoDB* and when the migration will be performed, it will not be necessary to migrate in *DynamoDB* the new data added during the migration phase.

To perform the migration is used a *Step Function* which can be run when LPH is working ensuring no LPH downtime during the migration.

5.3.1 Changes to the system before the migration phase

Before passing to the migration phase, some changes were required.

5.3.1.1 Amazon Aurora metrics table

To keep track of which *metrics* have been migrated to *DynamoDB* and their migration state, the *migratedToDynamoDB* field (*figure 5.7*) was added in the *Amazon Aurora metrics table*. The intermediate state *IN_MIGRATION* is useful for keeping track of which *metrics* are currently performing the migration.

Field name	Type	Notes
migratedToDynamoDB	enum	['NOT_MIGRATED', 'IN_MIGRATION', 'MIGRATED']

Figure 5.7: MigratedToDynamoDB field definition

5.3.1.2 API endpoint changes

As described before, two API calls are made when someone wants to create a *primitive metric*: the first one to add a new *metric group* with the *related dimensions* and the second one to create a new *metric* with all the other fields.

It is not necessary to modify the endpoint used to create new *metric groups*. Instead, a change is required for the endpoint used to create new *metrics*.

```
Validate metric body
Add metric in Amazon Aurora

# Additional behavior

Get request body data and put it in a DynamoDB item

if is a primitive metric:
    -> Get the related dimensions from Aurora
    -> Add to the item the related dimensions

if is an hybrid metric:
    -> Get the consumed metrics names from Aurora
    -> Add to the item the consumed metrics

Add the new item in DynamoDB (implementing the version control design pattern)
Set migratedToDynamoDB to 'MIGRATED'
```

Figure 5.8: Behavior of the AddMetric endpoint during the migration phase

The change to the *AddMetric* endpoint (*figure 5.8*) is not the only change that needs to be done because:

- *What if a metric which is already present in DynamoDB must be updated?*
- *What if a metric which is performing the migration must be updated?*

In order to cover these scenarios a change to the *UpdateMetric* endpoint is required.

The new endpoint behavior (*figure 5.9*) uses a *feature flag* to maintain the data consistency: it blocks updates of a *metric* when its migration status is *IN_MIGRATION*. Otherwise, it updates the *metric* in *Amazon Aurora* and if the *metric* is also stored in *DynamoDB* it updates the *metric* also there.

```
if the metric is in migration:
    -> Block the update

Update metric in Aurora

if the metric is present in DynamoDB:
    -> Update metric in DynamoDB
```

Figure 5.9: Behavior of the UpdateMetric endpoint during the migration phase

5.3.2 Step Function for the migration

To perform the migration of a single *metric*, I decided to use an *AWS Step Function*. The *Step Function* workflow (*figure 5.10*) involves the use of a main *AWS Lambda* which try to perform the migration of all the *metric versions* and the use of another *Lambda* used as a *Fallback State* of a *Catcher* in case an error occurred in the main one.

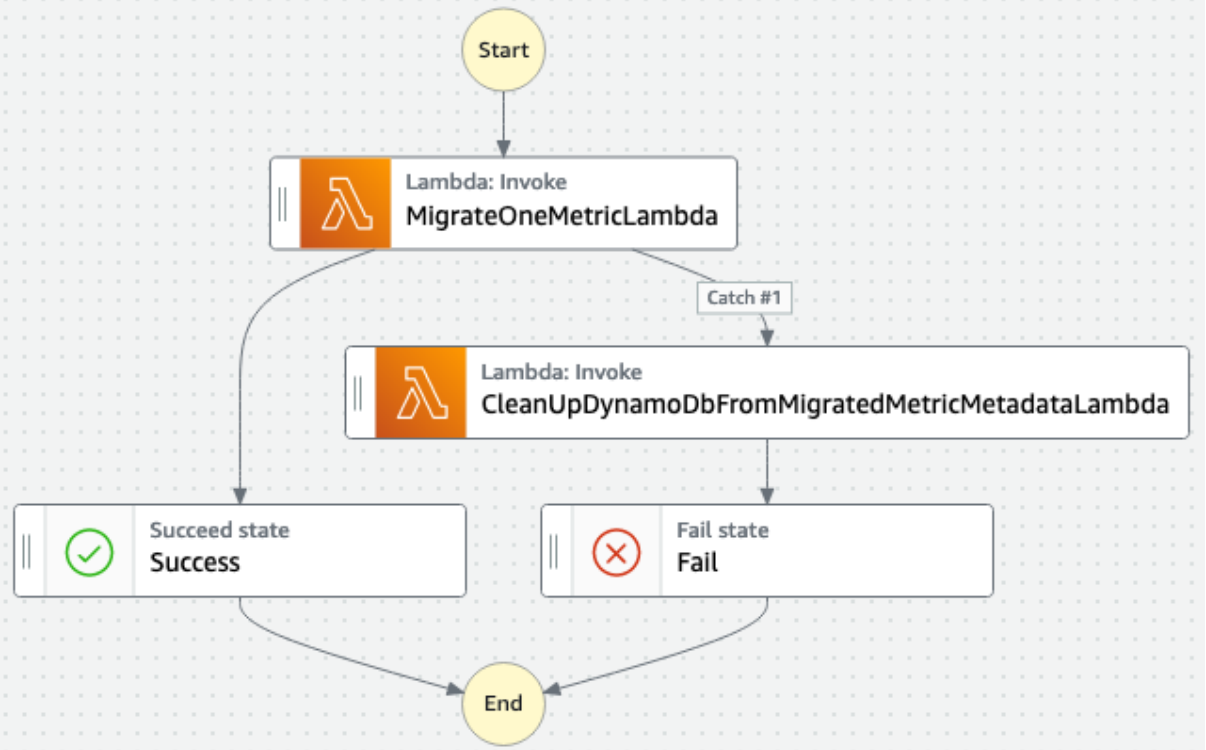

Figure 5.10: Workflow of the Step Function for the migration

5.3.2.1 Metric metadata migration Step Function definition

The following code snippet (*figure 5.11*) shows the Step Function definition.

```
{
   "Comment":"Metric metadata migration state machine",
   "StartAt":"MigrateOneMetric",
   "States":{
      "MigrateOneMetric":{
         "Type":"Task",
         "Resource":"${MigrateOneMetricLambdaArn}",
         "Next":"Success",
         "Retry":[ // Retry running this Lambda only when TooManyRequestsException occurs
            {
               "ErrorEquals":[
                  "Lambda.TooManyRequestsException" // When Lambda's concurrent execution limit is exceeded
               ],
               "IntervalSeconds":5, // Wait 5 seconds before retrying
               "MaxAttempts":10, // Retry for up to 10 times
               "BackoffRate":2.0 // Multiplier by which the retry interval increases during each attempt
            }
         ],
         "Catch":[ // Go to CleanUpDynamoDbFromMigratedMetricMetadata state when other errors occur
            {
               "ErrorEquals":[
                  "States.ALL"
               ],
               "ResultPath":null, // This makes the input of the next state equal to the input of the current one
               "Next":"CleanUpDynamoDbFromMigratedMetricMetadata"
            }
         ]
      },
      "CleanUpDynamoDbFromMigratedMetricMetadata":{
         "Type":"Task",
         "Resource":"${CleanUpDynamoDbFromMigratedMetricMetadataLambdaArn}",
         "Next":"Fail",
         "Retry":[ // Retry running this Lambda when TooManyRequestsException occurs
            {
               "ErrorEquals":[
                  "Lambda.TooManyRequestsException" // When Lambda's concurrent execution limit is exceeded
               ],
               "IntervalSeconds":5, // Wait 5 seconds before retrying
               "MaxAttempts":10, // Retry for up to 10 times
               "BackoffRate":2.0 // Multiplier by which the retry interval increases during each attempt
            }
         ]
      },
      "Success":{
         "Type":"Succeed"
      },
      "Fail":{
         "Type":"Fail"
      }
   }
}
```

Figure 5.11: Metric metadata migration Step Function definition

5.3.2.2 MigrateOneMetricLambda – Activity Diagram

The following diagram (*figure 5.12*) shows the *MigrateOneMetricLambda* workflow.

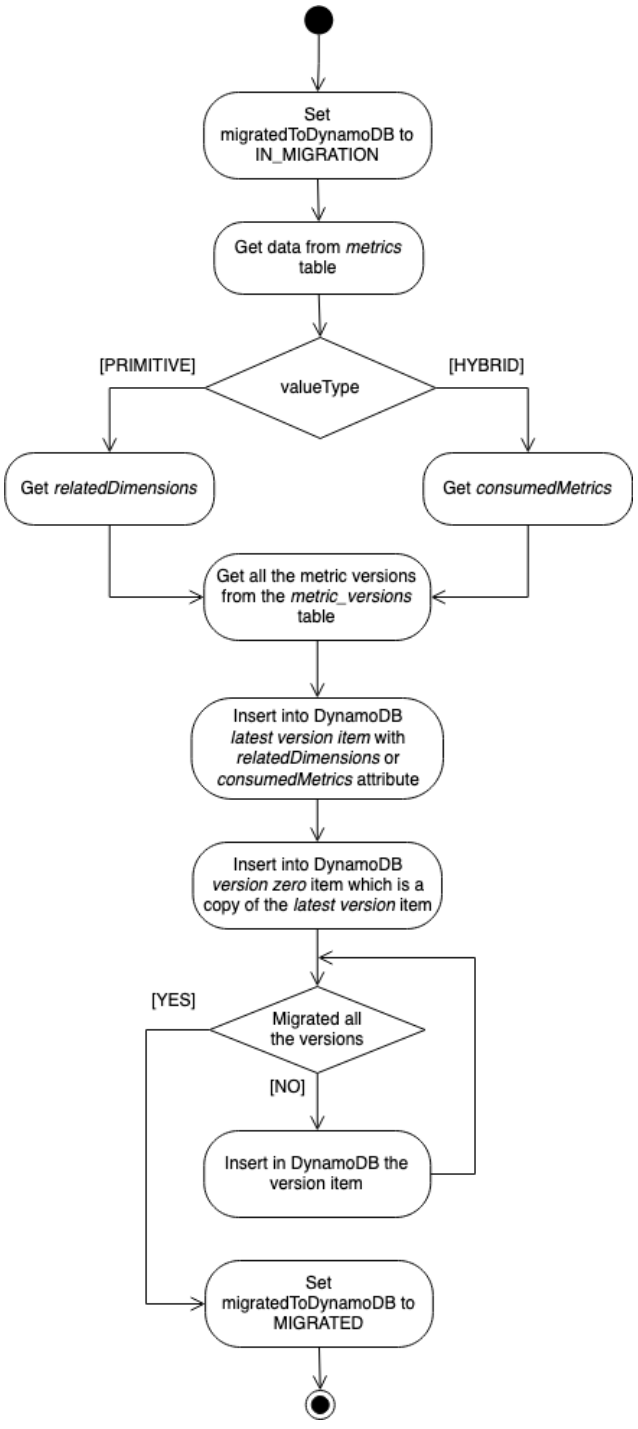

Figure 5.12: MigrateOneMetricLambda activity diagram

Note: In *Amazon Aurora* are stored only the *consumedMetrics* and *relatedDimensions* of the latest version of the *metrics*. This implies that in *DynamoDB* only the *latest version* and *version zero items* will have the *attribute relatedDimensions* or *consumedMetrics*.

5.3.2.3 CleanUpDynamoDbFromMigratedMetricMetadataLambda – Activity Diagram

The following diagram (*figure 5.13*) shows the *CleanUpDynamoDbFromMigratedMetric MetadataLambda* workflow.

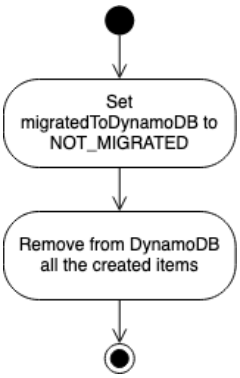

Figure 5.13: CleanUpDynamoDbFromMigratedMetricMetadataLambda activity diagram

This *Lambda* is used as *Fallback state* of the main *Lambda Catcher*. The task it has to do is clean-up *DynamoDB* from any *items* created by the main *Lambda*.

The clean-up is necessary because otherwise if a subsequent execution of *MigrateOneMetricLambda* tries to create an *item* with a *primary key* that already exists, *DynamoDB* generates an error. Therefore, if a previous execution of the migration *Step Function* failed and *DynamoDB* wasn't cleaned-up, the *Step Function* will continue to fail for this error with the *primary keys*.

5.3.2.4 Step Function invoker

In order to automate the migration process, it was used an *AWS Lambda* as an *invoker* of *metric metadata* migration *Step Function executions*. The task of the *invoker* (*figure 5.14*) is to query *Amazon Aurora* for 500 *metrics* which aren't already migrated to *DynamoDB* and for each *metric* to start a *Step Function execution*.

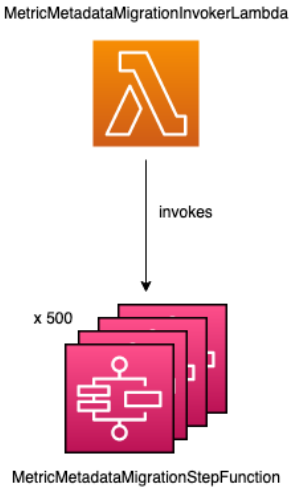

Figure 5.14: Metric metadata Step Function invoker

6 Glossary

Action item: A documented event, task, activity, or action that needs to take place. Action items are discrete units that can be handled by a single person.

API: *Application Programming Interface*, it is a type of software interface offering a service to other pieces of software.

Back-End: It refers to the separation of concerns between the presentation layer (*front-end*) and the data access layer (*back-end*) of a piece of software.

Business Intelligence: The strategies and technologies used by enterprises for the data analysis and management of business information.

CRUD: The four basic operations (create, read, update, and delete) of persistent storage.

Data aggregation: The compiling of information from databases with intent to prepare combined datasets for data processing.

Data ingestion: The process of obtaining and importing data for immediate use or storage in a database.

Data integration: Combining data residing in different sources and providing users with a unified view of them.

MySQL: An open-source relational database management system.

NoSQL: Database that provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.

Serverless: A cloud computing execution model in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers.

Stateless: A communication protocol in which the receiver must not retain session state from previous requests.

User story: An informal, natural language description of features of a software system.

7 Sources

[1] Wikipedia – Amazon: https://en.wikipedia.org/wiki/Amazon_(company)

[2] Wikipedia – Big Tech: https://en.wikipedia.org/wiki/Big_Tech

[3] Scrum.org: https://www.scrum.org/

[4] AWS – Continuos Integration: https://aws.amazon.com/devops/continuous-integration/

[5] AWS – DynamoDB: https://aws.amazon.com/dynamodb/

[6] AWS – Lambda: https://aws.amazon.com/lambda/

[7] AWS – Step Function: https://aws.amazon.com/step-functions/