Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería de Telecomunicación

Bachelor of Engineering in Telecommunication Technologies and Services

Final degree project

Aggregation and presentation of NCBI study metadata

Marta Arcones Rodríguez

June 2024

Bachelor of Engineering in Telecommunication Technologies and Services

Final degree project

Title: Aggregation and presentation of NCBI study metadata

Author: Marta Arcones Rodríguez

Advisor: Claudia Rodríguez López, M.D.

Academic Advisor: Miguel García Remesal, Ph.D.

Members of the board

President: Carmen Sánchez Ávila

Speaker: Valentín De La Rubia Hernández

Secretary: Juan José Vinagre Díaz

Deputy: Alberto Soria Marina

Madrid, June 2024

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería de Telecomunicación

Bachelor of Engineering in Telecommunication Technologies and Services

Final degree project

Aggregation and presentation of NCBI study metadata

Marta Arcones Rodríguez

June 2024

To my wife

Contents

1. Resumen

El propósito de este trabajo de fin de grado es automatizar un proceso de extracción de datos relativos a estudios biomédicos. Se pretende crear una aplicación fácil de utilizar para los equipos de investigación, capaz de extraer ciertos metadatos estadísticos de los experimentos pertenecientes a un conjunto de estudios previamente filtrados. Entre estos estadísticos se encuentran parámetros de gran interés como los que determinan la calidad de los experimentos.

Durante la primera fase del proyecto, se revisaron las aplicaciones ya existentes, sin embargo, ninguna de ellas realizaba el proceso requerido de manera integral. No obstante, estas herramientas se han incorporado en la solución final cubriendo algunos pasos intermedios, exprimiendo así su potencial y evitando redundancias. Además, también se han acordado requerimientos tanto funcionales como no funcionales, que han determinado la arquitectura del software desarrollado. Se trata de una arquitectura por capas donde el nivel de presentación se enfoca en la interacción con el usuario, el nivel de cómputo se encarga de obtener y transformar los datos generando reportes, y el nivel de observabilidad permite la monitorización detallada de los flujos de datos y la salud del sistema.

Para la construcción de la aplicación, se han utilizado tecnologías punteras, siendo una de las decisiones más determinantes hacer que el sistema fuese nativo en la nube utilizando Amazon Web Services. Esta elección ha facilitado enormemente los requisitos de ubicuidad y escalabilidad y, junto con el resto del stack tecnológico compuesto por herramientas como Terraform, Python3 o PostgreSQL, se ha obtenido un sistema reproducible que puede ser desplegado en la nube de cualquier institución.

Además, diversos patrones de diseño han influido notablemente en la implementación del sistema donde el principio de responsabilidad única y la orientación a eventos fueron claves para el diseño la capa de cómputo. Asimismo, la seguridad ha sido un requisito estricto desde el primer momento que ha mitigado cualquier riesgo de pérdida de datos o uso no autorizado. En el apartado de calidad, se han realizado pruebas en distintos niveles y se han automatizado las mismas para que cualquier despliegue sea verificado antes de ejecutarse. Como resultado del uso conjunto de las tecnologías mencionadas y la aplicación de estos principios de diseño, se ha desarrollado un sistema que dada una consulta de texto busca los estudios biomédicos relacionados, extrae los experimentos realizados en esos estudios y posteriormente los metadatos estadísticos de cada uno de esos experimentos. El sistema compila toda esta información en un archivo en formato CSV que contiene toda la trazabilidad y que los grupos de investigación puede utilizar para comparar los resultados de sus propios estudios y para explorar nuevas hipótesis antes de gastar recursos. En conclusión, este proyecto de fin de grado proporciona una herramienta muy valiosa y fácil de usar para los grupos de investigación biomédica, facilitando la búsqueda y selección de estudios de buena calidad relacionados con sus proyectos entre la ingente cantidad de datos publicados.

2. Summary

The purpose of this end-of-degree project was to address a need from biomedical researchers for an end-to-end, automatic, and easy-to-use application to extract sample statistics from studies fetched by a text query. The initial investigation revealed partial solutions, but none met the need holistically. Nonetheless, some of these already existing tools were integrated into the final solution to handle middle steps, thus leveraging their potential and avoiding redundancy.

Both functional and non-functional requirements were gathered during the research phase, forming the basis for the software architecture. A layered approach was adopted, with tiers for presentation, computation, and observability: the presentation layer, which focuses on user interaction and result delivery; the compute layer, that processes text queries into reports using a data pipeline; and the observability layer, which allows fine-grained monitoring of data flows and system health.

To construct the system, cutting-edge technologies were used, with the pivotal decision being to make the system cloud-native using *AWS*. This choice, while entailing a trade-off with vendor lock-in, brings forth numerous benefits including ubiquity and scalability. Moreover, *AWS* offers a comprehensive suite of services catering to almost every requirement of a software project. Accordingly, *AWS* solutions were employed for serverless processing, integration queues, persistence, user management, secure secret storage, networking, observability, etc. *AWS* choice heavily influenced the selection of subsequent technologies and design patterns where certain tools played significant roles in the development phase, such as *Terraform* for managing infrastructure as code, *Python3* for coding the algorithms within serverless processors, and *PostgreSQL* as a reliable storage solution.

Besides the technology stack, design patterns played a crucial role shaping the system implementation. Single-responsibility principle and event-driven strategy were key on the design of the data pipeline. Furthermore, some libraries were developed to foster code reusability across different components. In terms of quality assurance, test-driven development was embraced to confidently deploy any new capability. Additionally, security-by-design approach mitigated the exploitation risks at system inception. Moreover, given the nature of the queries that the system must handle, including some that obtain massive quantities of statistics, fail-safe and fail-fast mechanisms were implemented. These mechanisms preserve the integrity of processed entities, and in combination with best-effort clause, ensure that flawed data does not compromise system functionality.

Once the design patterns and the technology stack were settled, the data pipeline took shape comprising nine sets of serverless processors, each one with a defined duty rigorously tested with diverse inputs to ensure reliability. To enhance fault tolerance, fallback techniques were integrated to address known errors and provide valuable insights. Having received as input the text query and the e-mail address of the researcher, the data pipeline extracts, transforms and loads outputs gradually, culminating the process with an e-mail delivery of a CSV report containing all the denormalized statistical metadata back to the scientist. The CSV format is renowned for versatility, making it compatible with spreadsheet editors or command-line processing tools. Finally, the data pipeline also stores some records that are not currently delivered to the scientist, which may be subject of further improvements of the system in the future.

Likewise, regarding the possible improvements that could be implemented on *SRA-Collector* service, numerous enhancements have been proposed to address the existing weaknesses in the product. These include performance optimizations aiming at accelerating data pipeline processing, cost efficiency strategies to cut down cloud infrastructure expenses, and solutions to tackle the system limitations such as serverless timeouts, query prioritization absence, and mail server constraints. Further refinements can be committed in the user experience too, like establishing a self-service for signup operation or providing a custom UI. On the security chapter, there is a need of enrichment in the authentication control to access the observability platform, streamlining its maintenance while increasing its capabilities.

In conclusion, this final degree project provides a very valuable and easy-to-use tool for biomedical scientists that work daily with NCBI databases, facilitating the search and selection of high quality studies among the huge number of already published sequencing data. However, to deploy the system with guarantees, an institution should assume its monetary costs. On this regard, the existence of the product can be publicized among researchers' forums to increase the chances of a sponsorship.

3. Overview and Goals

1. Introduction

Biomedical scientists use daily databases provided by the National Center of Biotechnology Information, abbreviated as NCBI, from United States government available online at [1]. One of the main goals of this website is to store molecular and genetic studies, allowing the scientific community to search for any published dataset on the subject of interest and to reanalyze from the raw data sequencing datasets.

Sequence-based data represent the majority of stored studies, and among them, RNA sequencing, commonly known as RNA-Seq experiments result of special interest to the researchers, as they allow to quantify RNA molecules in a biological sample, providing an overview of the general gene expression in such sample at a given time [2]. More in particular, the NCBI website offers the Gene Expression Omnibus DataSets repository, hereafter GEO, which given a text query, can list all the scientific studies with genomics data related to the query as shown in Figure 3.1.

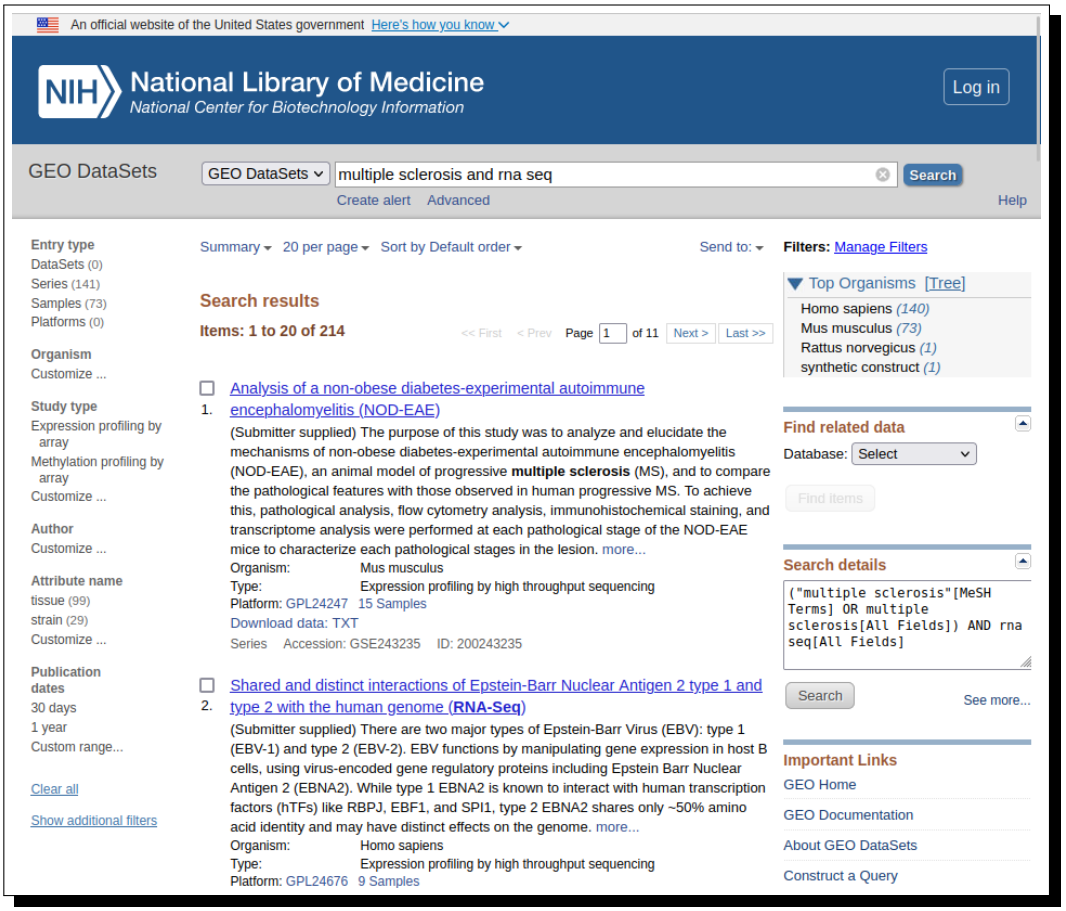

Figure 3.1: GEO Query Result

The studies found in GEO repository are linked to Sequence Read Archive repository, SRA for short, where each of the experiments comprising a given study can be accessed individually in order to check their statistical metadata or download its raw data, as shown in Figure 3.2.

In GEO, each study has an unique NCBI number identifier and a GSE number that correlates with the SRP code in SRA database. Each experimental sample from a given SRP is identified by an SRR code, and for each SRR a number of statistical metadata is given, from which six parameters are of special interest, as detailed below. This sequence is schematically depicted in Figure 3.3.

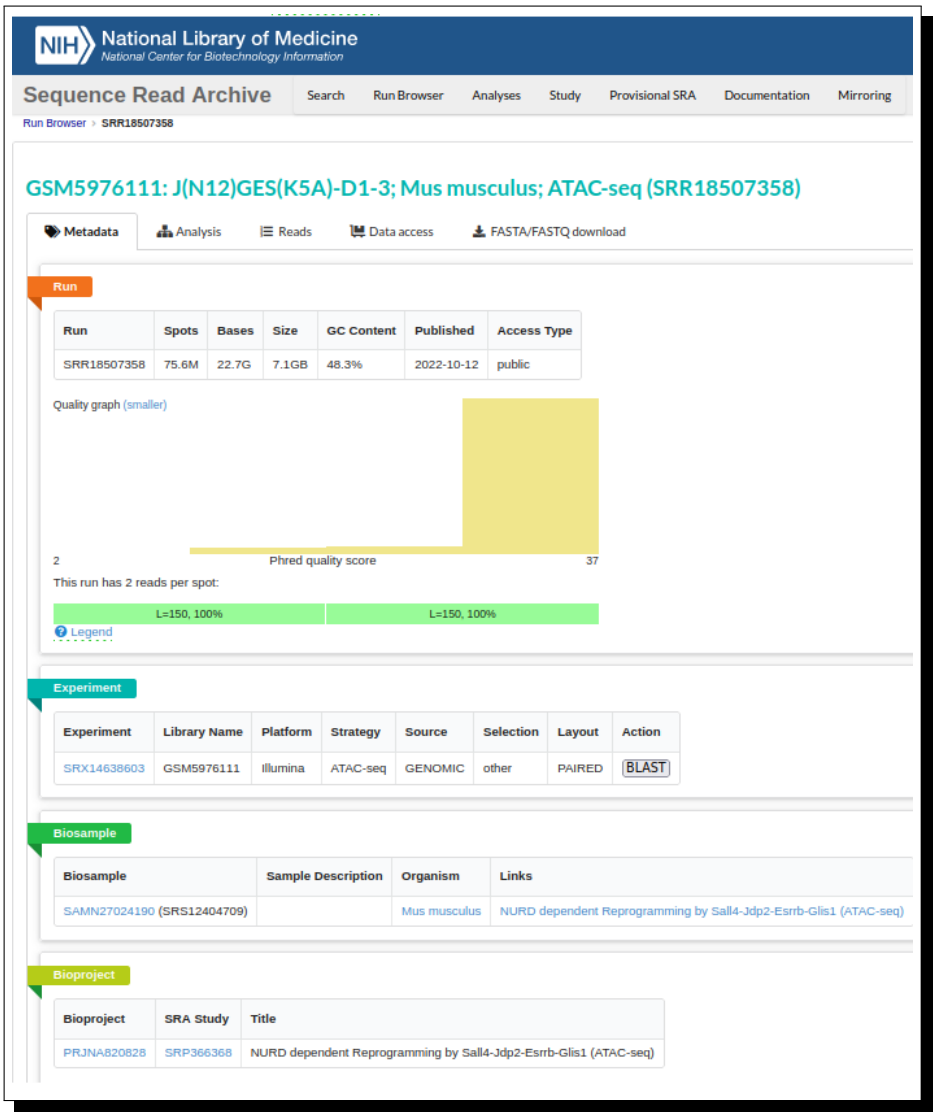

Figure 3.2: SRA Experiment Metadata

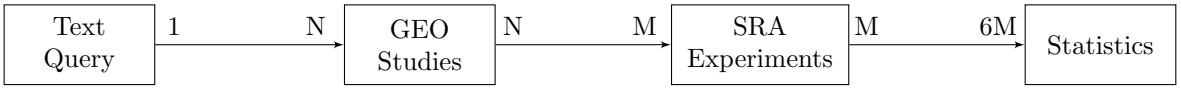

Figure 3.3: Sequential flow from GEO Database to SRA Database

Among this statistical metadata, the following fields are particularly useful to identify high quality experiments that may be worthy to be downloaded and reanalyzed by the researcher, as well as to find among the parameters those needed in a subsequent bioinformatics processing:

- **Spots Count**: provides the sequencing depth of the sample, i.e. the level of coverage of each genomic unit (base) by aligned sequence reads, allowing the researcher to find out if the sample is suitable for high throughput analyses or can be used only for general genetic expression analyses.
- **Bases Count**: number of sequenced bases. It correlates with the spots count and may be needed as a normalization parameter in a subsequent bioinformatics processing.
- **Average Read Length**: mean length of the sequence reads, a parameter needed by aligner tools.
- **Phred Quality Count**: gives a logarithmic estimation of the probability of finding an error in the identification of one base by the sequencer among a given number of bases [3]. For example, a phred score of 30 indicates a likelihood of incorrect base assignment of 0,1 %. High quality studies are those with phred scores over thirty-six for the great majority of the reads.

- **Layout**: indicates if the sequencing strategy was single or paired-end, a valuable parameter to decide which information can be extracted from the sample after further processing.
- **Organism**: the species from which the samples were obtained, parameter needed to choose the reference genome when aligning the reads in subsequent bioinformatics analyses.

Most researchers are average informatics users, thus their workflows for analyzing public data rely on a keyword search through the GEO website that list the potential relevant studies, followed by a thorough analysis of the suitability of all of them by checking technical and quality parameters summarized before of each experimental sample. For this, the scientists should browse from the initial list of studies found in GEO, to the details of a specific study also provided in GEO, and then jump to SRA website by clicking on a link called *SRA Run Selector* at the bottom of the study detail page. This link moves them to SRA database, where the study is hosted with a different identifier. Finally, each of the experimental samples related to the study must be checked one by one to obtain all their statistical metadata. This strategy is highly time consuming, as a standard query may provide more than a hundred of studies, and checking manually the quality of all the experiments in each study becomes into an unfeasible task.

An efficient use of GEO and SRA tools brings the scientists the opportunity to take advantage of public studies in order to compare them to their own data or to explore novel hypotheses, sparing both economical and biological resources. For example, taking the most of already generated computational data from biological samples allow researchers to accomplish with the *"Three Rs principle"* (replacement, reduction and refinement) that is aimed to minimize the use of animals in research [4].

Thus, researchers have the need of aggregates of all the statistical metadata records for a given GEO query that allow them to select the best available studies for further analyses efficiently. However, the current tool-set of NCBI does not even provide a native way to acquire the statistical metadata for all the experiments from a single study, and some of the libraries developed to fill this gap are incomplete and operated by command line, not being accessible for inexperienced users. Therefore, the main goal of this final degree project is to provide a user friendly application that automates the statistical metadata extraction of all the studies and experimental samples of a given NCBI GEO DataSets query.

2. Goals

The general and specific key results of the present final degree project are detailed below.

- **Goal 1**. Design an open-source automatic process that extracts the statistical metadata from every experiment belonging to the studies listed after any GEO query.
  1. Extract a hierarchy from identifiers of NCBI studies to SRA statistical metadata sets.
  2. Perform a traceable conversion of identity codes between the different databases.
  3. Store intermediate information acquired during the process.
  4. Keep data in an isolated and secured environment.
  5. Ensure dynamic escalation of the system to handle incoming traffic.
  6. Provide continuous feedback to the administrator about the progress of the system.
- **Goal 2**. Provide a CSV file with all statistical metadata for a given set of studies.
  1. Generate the final CSV file containing properly organized and filterable statistical metadata.
  2. Keep traceability of the original hierarchy in the final CSV output.
  3. Send CSV file by e-mail to the user that did the query.

4. Background

A number of studies have aimed to analyze and publish summaries of RNA-seq datasets in the past ten years. However, they have focused on the retrieval of final gene expression results from a huge number of studies simultaneously [5], [6], a goal that is currently covered by gene expression atlas [7], [8]. As discussed above, scientists usually claim for a little number of studies among the great amount provided in a general query that exactly fit their necessities, thus, tools providing relevant metadata statistics are now of more interest.

The NCBI documentation provide some integration tools to create extensions using their data, as the *E-Utilities* HTTP API [9]. This API offers the *esearch* method which can be used to fetch the same list of studies that scientists are looking for in the GEO user interface as shown in Figure 3.1, so it may serve as a starting point for the automation of the process.

Regarding the matching between the studies in GEO repository and the experiments in SRA database, no native way was found using the NCBI official tools unfortunately. Looking over the Internet, it was clear that this need was shared among many scientists and an open-source library was built to fill this gap, the name of the library is *Pysradb* [10].

Pysradb package was developed to provide programmatically access to SRA data, allowing to make queries by command line and to download data and metadata from SRA. It also provides support to translate a GEO study onto an SRA study and also to get all experiments for a given SRA study, also known as SRP. Additionally, it has a method to list some of the statistic metadata of interest. However, the main maintainer of *Pysradb* library, Saket Choudhary, discarded to add phred quality score to the outcome of his library. In addition, as has already being said, the fact that it must be operated by command line represents a handicap for many potential users.

Other proposals to address the metadata extraction [11] opt for retrieving the statistics of all the SRA entries and then convert it to JSON format that permits further transformations. However, this approach does not seem to be very efficient, since it does not allow to extract directly the metadata from a specific query.

In conclusion, a new method should be developed that might use one of the partial approaches mentioned above to provide an end to end functionality both easy to use and convenient.

5. System Analysis

1. Initial Research

The investigation started by looking at NCBI documentation to see which integration tools are provided to create extensions using their data. This leads to the discovery of the *E-Utilities* HTTP API[9].

This API offers the *esearch* method which can be used to fetch the same list of studies that scientists are looking for in the GEO user interface as shown in Figure 3.1, so it could be the starting point of the process.

Another open question was how to do the matching between the studies in GEO repository and the experiments in SRA database. Scientists fill this gap by accessing to each study in GEO database and clicking on a link called *SRA Run Selector* at the bottom of the study detail page. This link moves them to SRA database, where the study is hosted with a different identifier and the set of experiments related to that study can be found.

Unfortunately, no native way was found using NCBI official tools to do this match step between both data repositories. Looking over the Internet, it was clear that this need was shared among many scientists and an open-source library was built to fill this gap, the name of the library is *Pysradb*[10].

Pysradb provides support to translate a GEO study onto an SRA study and also to get all experiments for a given SRA study. Additionally, it has a method to provide some of the statistic metadata of interest, but it lacks support to obtain the phred quality score of a given experiment. The main maintainer of *Pysradb* library, Saket Choudhary [12], discarded to add phred quality score to the outcome of his library, so the usage of *Pysradb* in the project will be limited to do the matching between databases and to extract the experiment list.

To retrieve all statistic metadata, including the phred score, the method selected is the same used by the SRA website: once a scientist accesses one experiment, a request is sent to NCBI *Traces Service*[13] that returns all statistics in XML. By parsing this XML and linking the statistic metadata back to the GEO query, the information requested will be complete.

2. Requirements

Once the initial research concluded, a list of requirements for the system were created. They can be divided in functional and non functional requirements:

2.1. Functional Requirements

- A user interface, hereafter UI, available at the Internet should act as entry point of the application.
- The UI should allow scientists to send a text query to the system for processing.
- The UI should allow scientists to provide their e-mail address for receiving the results.
- An unique reference for each query should be communicated to scientists when the query is sent.
- An e-mail with results per query should be sent to the requester scientist.
- The mail should contain a CSV file attached with a header line followed by a set of rows.
- The CSV file will contain as many rows as SRA experiments related to all the GEO studies retrieved by the query.
- Each row will contain all statistic metadata for the given SRA experiment, i.e, spots count, base count, reads, layout, phred and organism.

2.2. Non-Functional Requirements

- The system should be available only for registered users.
- The system keeps secrets in a safe storage.

- The system should be able to cope with input queries that fetch hundreds of studies and, subsequently, thousands of experiments.
- The system should be able to cope with multiple scientists using the system at the same time.
- Administrator should have full observability of scientists' requests, logs of the application and logs of the servers.
- The system will persist all retrieved data for a given query for further analysis.
- The system can be easily replicated.
- The system is open-source.

3. System Architecture

The main technical challenge to meet the project requirements is scalability as the number of SRA experiments per GEO study is unknown beforehand. On this regard some exploratory tests were conducted to estimate the proportionality factor between studies and experiments. It was confirmed that such proportionality factor can differ greatly, being $k \approx 1$ for queries focusing on a disease like *"asthma"* and being clearly $k \gg 1$ for queries including some keywords related to the experimental methodology of the study like *"rna seq"*.

Furthermore, the requirements of ubiquity of the system via Internet and full observability of any user interaction made clear that the system should be composed of different layers, as depicted in Figure 5.1.

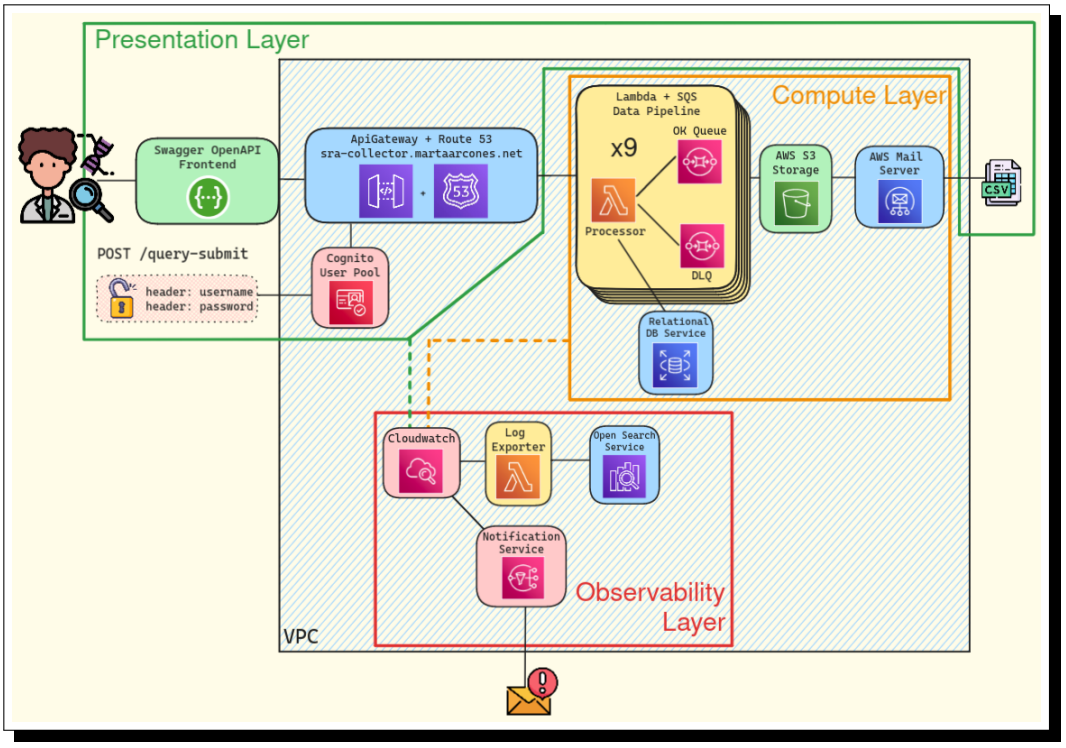

Figure 5.1: High Level Architecture Diagram

3.1. Presentation Layer

In this layer is where the user interaction happens. As it will be exposed in the Internet, a well-known unsafe environment, security access mechanisms should be implemented here to prevent the exploit of the system.

An easy-to-remember and static URL should be created as the application access point, for which an Internet domain should be purchased. Scientists will open the given URL in a browser using an authentication mechanism and a set of credentials provided beforehand by the administrator.

As credentials, scientist' mail and a password will be used. The purpose of using the mail address serves a dual function, as it will be also used to deliver the results.

Afterwards, scientists will write their query in an input field available at the frontend page. The query, alongside credentials information will be sent to the backend server. Once on the backend, firstly the credentials should be validated against the identity and access management service.

If the credentials validation fails, the backend will send a proper unauthorized HTTP code to the frontend. On the other hand, if the validation succeeds, the system will generate an unique identifier for the given query and will return it to the frontend so the scientist can store it as a reference of the request submitted.

After some time, depending on the size of the query and the load of the system due to other simultaneous queries, scientists should receive an e-mail with a CSV file attached that contains one row per SRA experiment, the parent study identifier and the statistical metadata.

3.2. Compute Layer

In this layer is where the data processing happens. The compute layer should be able to process huge number of requests in parallel to provide acceptable processing times.

The process starts by receiving the query alongside with the mail address from the presentation layer. This information is stored in the database as the initial entity from which the rest entities will inherit.

Subsequently, the system progress the request in a data pipeline that will do the required operations to obtain all the studies for the given query, transform the studies to experiments and finally gather the experiments' statistical metadata. For the shake of completeness and further development, alongside with these main entities, the system will also store secondary entities that are returned by NCBI API. Likewise, some errors are also stored, for example, when the experiment list of a study cannot be retrieved as a result of an outage in the NCBI API.

When the system finishes scanning and storing the statistical metadata of all experiments, it performs a query over the database to generate the final CSV file. Once the file is ready, the mail will be delivered.

3.3. Observability Layer

The system should be instrumented to provide enough insights to the administrator to fine tune it in case of high load or issues. Observability will be implemented in both push and pull approaches [14]:

- **Push approach**: a set of alarms will be configured to inform administrator by mail of any malfunction of the system. These alarms are triggered when system metrics surpass a certain threshold.
- **Pull approach**: administrator will have available dashboards with information about the requests that scientists are sending to the system, all output logs of the data pipeline and some insights regarding resource consumption. These dashboards should provide filtering capabilities to be able to locate the desired records easily.

6. Technology Stack

1. Git & GitHub

As the project needs to be open-source, a public repository to host the code was created in GitHub with name *SRA-Collector* [15] depicted in Figure 6.1. All code management has been done using *git* distributed version control system and *GitHub Flow* [16] pattern for branch naming.

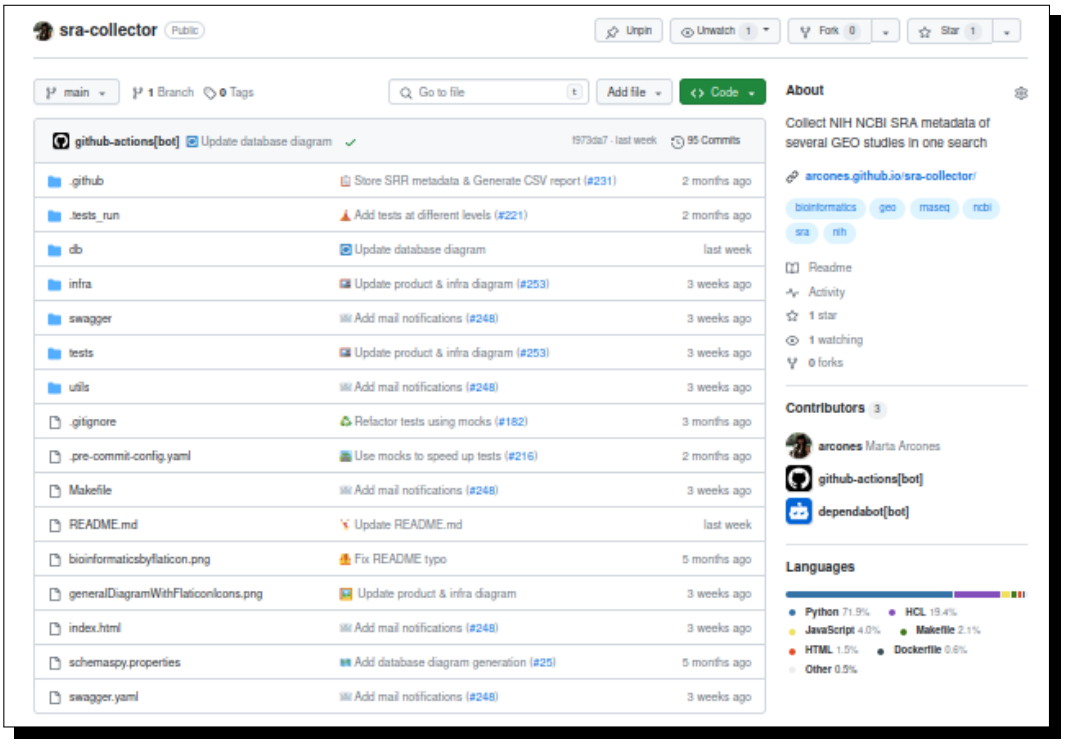

Figure 6.1: SRA-Collector GitHub Code Repository

Apart from hosting, *GitHub* also provides continuous integration and continuous delivery/deployment services, hence CI/CD, known as *Actions*. *GitHub Actions*, presented in Figure 6.2, were therefore used to automatically build, test, and deploy the code.

Another interesting capability of *GitHub Actions* is *Dependabot* [17]. *Dependabot* automatically updates project dependencies to the last stable version. This is a great help for the maintenance of the project as checking dependencies updates manually is an error prone and time consuming task. Likewise, the risk of having outdated vulnerable dependencies that increase the probability of a dependency confusion attack [18] is reduced greatly.

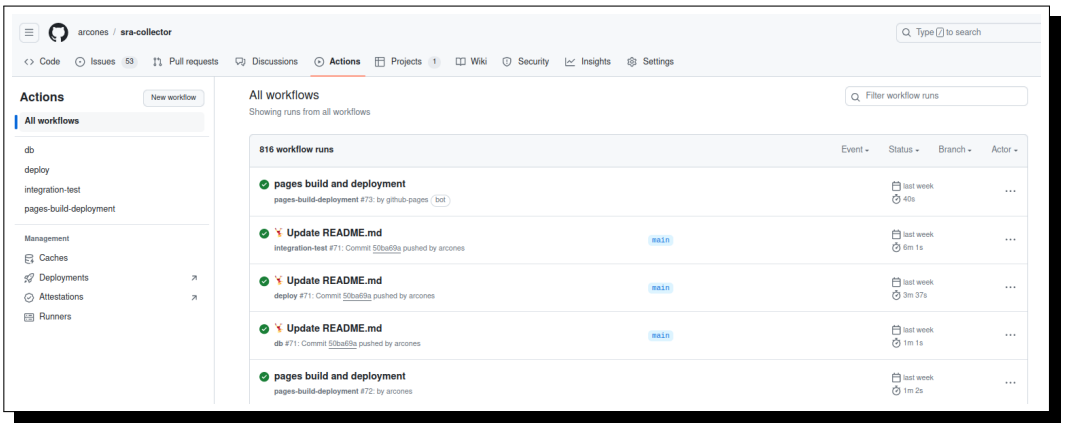

Figure 6.2: SRA-Collector GitHub Actions Pipelines

In addition, *GitHub* provides a project management toolset known as *GitHub projects* [19]. The GitHub project [20] shown in Figure 6.3 was created to organize all the tasks required. *GitHub* also provides milestones [21] that were used to prioritize development tasks for this academic presentation and more. Also it is possible to classify tasks with some custom field directly created. For example in this project, *"Domain"* field was created to classify tasks in one of the eight following domains:

- **Product**: any task contributing to application end-to-end functionality.
- **Bugs**: for fixing wrong or unexpected behaviour found while using the application.
- **Quality**: tasks related to unit, integration, load and manual tests.
- **Security**: tasks to prevent malicious usage of the system.
- **Observability**: tasks to control the behaviour of the system using logs, metrics and dashboards.
- **Developer Experience**: tasks to foster code readability and maintainability.
- **Performance**: tasks to speed up the processes of the system.
- **Documentation**: tasks for creating information about the system for other developers or users.

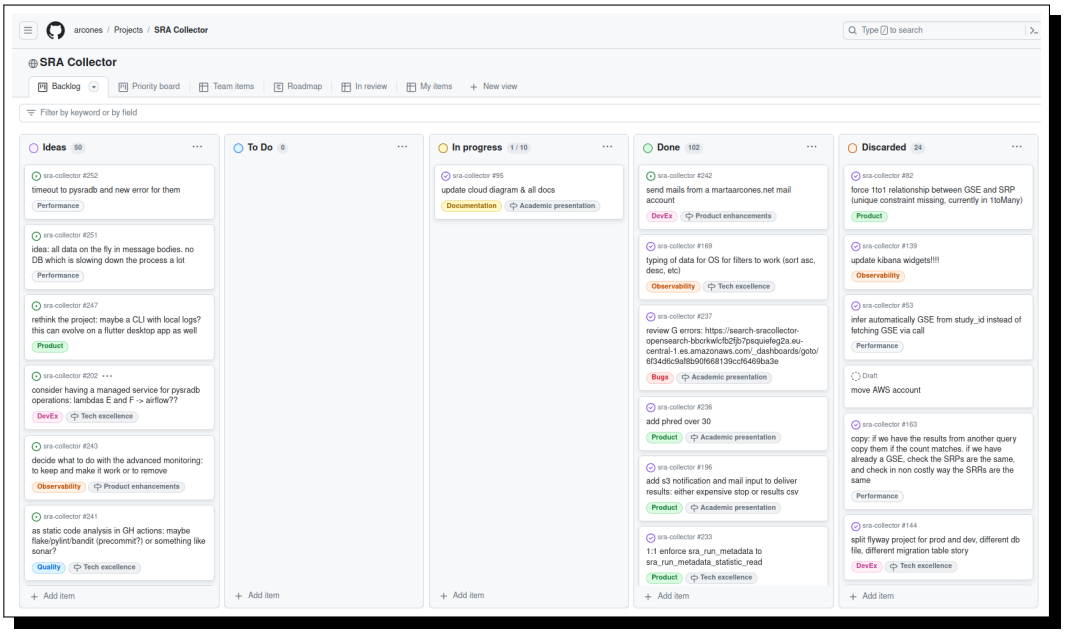

Figure 6.3: SRA-Collector GitHub Project

Finally, *GitHub* provides a functionality called *Pages* that can be used to expose a website to the Internet

directly using the files in the repository. This was the most straight forward way to provide a frontend to scientists where they can send their request to the backend infrastructure, as covered in detail in Section 16.

2. Amazon Web Services

As the project needs to be constantly exposed in the Internet, *Amazon Web Services* [22], henceforth *AWS*, was chosen as public cloud provider because its competitive prices [23], its superb documentation [24] and the big community forums available online [25], [26].

AWS provides infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS) tools [27]. In the project, the following *AWS* services where used:

- ***Virtual Private Cloud (VPC)*** [28]: as *AWS* is a public cloud, several customers can be using simultaneously the same hardware; thus it was mandatory to isolate the resources related to the project creating a *VPC*, which is a software defined network, also known as SDN [29]. To be accessible from the Internet, the *VPC* has configured an Internet gateway.
- ***Identity and Access Management (IAM)*** [30]: used to manage permissions. For example *GitHub Actions* required a set of credentials to deploy changes to the cloud. Also, any service to service communication happening between different *AWS* services in the project required an *IAM* policy, as by default all interactions are denied following the principle of least privilege [31].
- ***Route 53*** [32]: used to purchase the `martaarcones.net` Internet domain and creating the DNSs entries, so that the domain is accessible on the web with a static address.
- ***API Gateway*** [33]: used as interface between the frontend of the application and the backend. Using *API Gateway* an HTTP API was created with security mechanisms in place, like CORS support [34] or credentials verification. *API Gateway* also generates unique identifiers for each request processed, which can be used to cover the functional requirement regarding the feedback scientists should expect when they issue the query. Moreover it allows high concurrency on input requests due its scalable and distributed architecture [35], which will be key to be compliant with non-functional requirements.
- ***Cognito*** [36]: utilized as user pool where the identities of the users are created and maintained. This service was employed to register and therefore restrict the system usage only to authorized users.
- ***Lambda*** [37]: used as serverless processors [38] in the data pipeline, and to feed the observability platform. Most lambdas are written on *Python3* programming language, with an exception written on *NodeJS*. Data pipeline's *Lambdas* were key to achieve enough processing capacity for high loads due to bursts of requests, as they automatically scale depending on how many messages they need to process.
- ***Simple Queue Service (SQS)*** [39]: it was used as temporal message storage in the data pipeline. It allows to decouple the different components of the system by permitting them to communicate asynchronously. *SQS* holds the messages until a *Lambda* picks them. If the processing fail, *SQS* either retry the message or moves it to a dead letter queue where it can remain for longer periods.
- ***Relational Database Service (RDS)*** [40]: also called *RDS*, it was used as persistent storage of all data the lambdas extract from their interactions with NCBI servers and libraries, as explained in the non-functional requirements.
- ***Secrets Manager*** [41]: used to host confidential data as the NCBI API key or the password of *RDS*. Its support to rotate automatically *RDS* password was used as well. This service was key to be compliant with the secrets security non-functional requirement.
- ***Simple Storage Service (S3)*** [42]: it was used to store infrastructure metadata and lambdas source code and libraries. Besides, it also stores output CSV files from the data pipeline, consequently helping to the achievement of the functional requirement regarding the generated file.
- ***Simple Email Service (SES)*** [43]: it was used to deliver mails to the scientists once the process of their query has ended. These mails will contain attachments, hence covering all functional requirements regarding mail output.
- ***Simple Notification Service (SNS)*** [44]: it was used to deliver e-mails to the system administrator if any of the alerts triggered.

- ***CloudWatch*** [45]: observability service that captures all the logs and metrics emitted at an *AWS* account. In this project *Lambda*, *API Gateway* and *SQS* logs and metrics became specially relevant. *CloudWatch* was also used to configure alerts when any metric surpass a configured threshold.
- ***OpenSearch*** [46]: monitoring service used to present dashboards with filtering capabilities that were fed by *CloudWatch* logs. It was key to provide full observability on system functioning to the administrator.

3. Terraform

All the configuration required to setup the whole system in the cloud needs to be reproduced easily, hence, *Terraform* infrastructure as code technology was used [47]. With *Terraform*, the configuration of *AWS* services used is written in source code files, and it provides a simple command-line interpreter, hereafter CLI, to deploy the infrastructure in the *AWS* account selected. These features come handy to integrate with version control system and CI/CD systems.

Moreover, *Terraform* also hosts the registry website [48] portrayed in Figure 6.4. It can be used for downloading modules which are self-contained *Terraform* packages ready to be deployed. When resource configuration becomes tricky, but at the same time is a common problem, modules can alleviate the implementation process a lot. One example is *VPC* creation, for which it was used a module [49] invocation instead of creation from scratch [50].

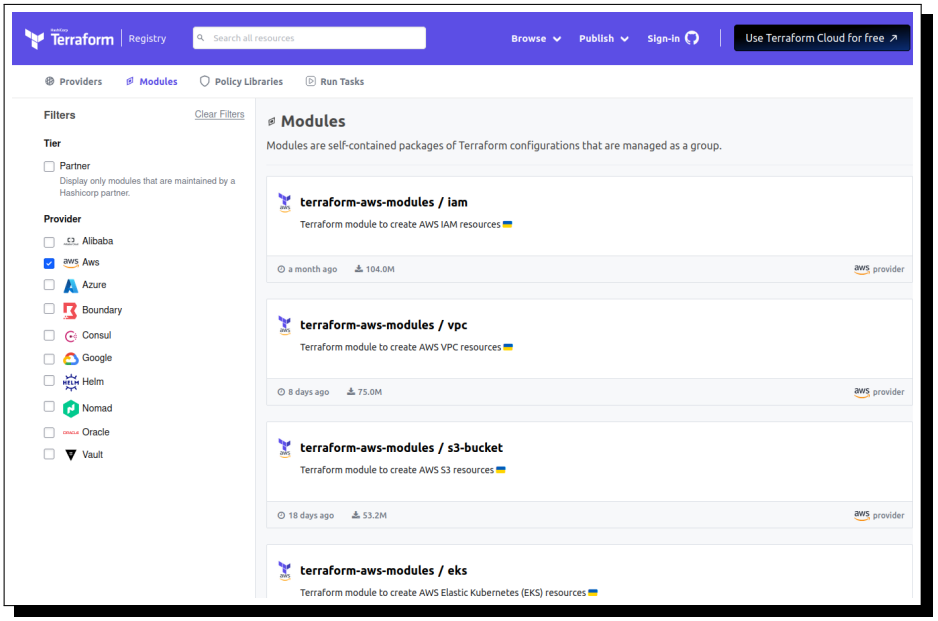

Figure 6.4: *AWS* Modules In Terraform Registry

Finally, within the project, it can be hosted local *Terraform* modules, which are groups of configurations that can be instantiated from other locations to create as many resources as they provide. This interesting capability was heavily used for data pipeline infrastructure as it is built among several sets of *Lambda* functions as processors and *SQS* queues as message holders as shown in Figure 5.1. During the development of *SRA-Collector*, the management of the infrastructure sources of the data pipeline become more and more tedious and tricky, so at some point a module [51] was created and later on the configuration of all pieces of the data pipeline [52] were easier to maintain.

4. Docker

Docker [53] software facilitates the creation of isolated lightweight containers to deploy applications. *Docker* was implicitly used in the CI/CD environment, as *GitHub Actions* runners rely on *Docker* to isolate the builds [54]. Likewise, *Lambda* functions also use *Docker* [55] as underlying technology.

Besides, there is an explicit usage of *Docker* in the project. As *Lambda* functions require several libraries to work, *Docker* was used to create from scratch the dependencies package as shown in Figure 6.5. Among the

dependencies required there are both libraries created specifically for the project and libraries downloaded from *PyPI* which is the biggest public *Python* package registry. A *Docker* container [56] created from *Python* base image [57] was used to build the dependencies package so any dependency conflict was prevented.

```
FROM python:3.11

ENV DB_CONNECTION_LIB_FOLDER=db_connection
ENV DB_CONNECTION_LIB_VERSION=0.0.5

ENV SQS_HELP_LIB_FOLDER=sqs_helper
ENV SQS_HELP_LIB_VERSION=0.0.1

ENV S3_HELP_LIB_FOLDER=s3_helper
ENV S3_HELP_LIB_VERSION=0.0.1


ENV DEPS_FOLDER=lambda-dependencies/python

RUN apt update && apt install -y zip && pip install wheel setuptools build

RUN mkdir -p ${DEPS_FOLDER}

RUN cd ${DEPS_FOLDER} \
    && pip install pysradb psycopg2-binary -t . \
    && cd .. \
    && zip -r9 ../dependencies.zip .

COPY ${DB_CONNECTION_LIB_FOLDER} ${DB_CONNECTION_LIB_FOLDER}
RUN cd ${DB_CONNECTION_LIB_FOLDER} && python -m build
RUN cd ${DEPS_FOLDER} \
    && pip install /${DB_CONNECTION_LIB_FOLDER}/dist/${DB_CONNECTION_LIB_FOLDER}-${DB_CONNECTION_LIB_VERSION}-py3-none-any.whl -t . \
    && cd .. \
    && zip -ur9 ../dependencies.zip .

COPY ${SQS_HELP_LIB_FOLDER} ${SQS_HELP_LIB_FOLDER}
RUN cd ${SQS_HELP_LIB_FOLDER} && python -m build
RUN cd ${DEPS_FOLDER} \
    && pip install /${SQS_HELP_LIB_FOLDER}/dist/${SQS_HELP_LIB_FOLDER}-${SQS_HELP_LIB_VERSION}-py3-none-any.whl -t . \
    && cd .. \
    && zip -ur9 ../dependencies.zip .

COPY ${S3_HELP_LIB_FOLDER} ${S3_HELP_LIB_FOLDER}
RUN cd ${S3_HELP_LIB_FOLDER} && python -m build
RUN cd ${DEPS_FOLDER} \
    && pip install /${S3_HELP_LIB_FOLDER}/dist/${S3_HELP_LIB_FOLDER}-${S3_HELP_LIB_VERSION}-py3-none-any.whl -t . \
    && cd .. \
    && zip -ur9 ../dependencies.zip .
```

Figure 6.5: SRA-Collector Lambdas Dockerfile

5. Python3

Each *Lambda* function of the data pipeline contains some code written in *Python3* programming language [58]. This choice was not forced by *AWS* as it supports a broad range of runtimes [59]. Instead, *Python3* was chosen because of the nature of the project, as it has great support for working with data structures.

In general, it is notorious the big online community for *Python3* [60] and the huge amount of libraries available online [61] at *PyPI*. In the bioinformatics field in particular, *Python3* is one of the most popular choices for this kind of projects [62].

6. NodeJS

The only *Lambda* function that was not written in *Python3* was written instead in *NodeJS*. This lambda does not belong to the data pipeline and instead covers an auxiliary function which is exporting the *CloudWatch*

logs produced at the data pipeline and infrastructure to *OpenSearch*. As the examples [63] provided by *AWS* to do this operation use *NodeJS*, the programming language choice was kept.

7. GNU Make

This build automation tool [64] was used to gather all procedures, in the shape of piped command-line *Linux* functions, required to manage the system. In *GNU Make* argot, these procedures are known as *targets*. The project has *targets* for data migrations, infrastructure deployments, testing, clean ups, etc as depicted in its Makefile from Figure 6.6.

```
SHELL=/bin/bash

FLYWAY_PASSWORD?='$(shell aws secretsmanager get-secret-value --secret-id rds\!db-3ce19e76-772e-4b32-b2b1-fc3e6d54c7f6 --region eu-central-1 --output json | jq -r .SecretString | jq -r .password)'
DATABASE_PASSWORD?=$(shell urlencode $(FLYWAY_PASSWORD))
DB_CONNECTION_LIB_VERSION=0.0.5
SQS_HELPER_LIB_VERSION=0.0.1
S3_HELPER_LIB_VERSION=0.0.1

db-migrations-integration-test:
	@docker run --rm -v $(shell pwd)/db/migrations:/flyway/sql -v $(shell pwd)/db/conf-integration-test:/flyway/conf -e FLYWAY_PASSWORD=$(FLYWAY_PASSWORD) flyway/flyway clean migrate

db-migrations-prod:
	@docker run --rm -v $(shell pwd)/db/migrations:/flyway/sql -v $(shell pwd)/db/conf-prod:/flyway/conf -e FLYWAY_PASSWORD=$(FLYWAY_PASSWORD) flyway/flyway migrate

db-migrations-unit-test:
	@docker run --rm -v $(shell pwd)/db/migrations:/flyway/sql -v $(shell pwd)/db/conf-unit-test:/flyway/conf -v $(shell pwd)/tmp/test-db:/db  flyway/flyway clean migrate
	sudo chown $(shell whoami):$(shell whoami) tmp/test-db/test.db.mv.db

update-diagram:
	@rm -rf tmp/diagrams && mkdir -p tmp/diagrams && chmod 777 tmp/diagrams && \
	docker run -v $(shell pwd)/tmp/diagrams:/output -v $(shell pwd)/schemaspy.properties:/schemaspy.properties schemaspy/schemaspy -p $(FLYWAY_PASSWORD) && \
	cp tmp/diagrams/diagrams/summary/relationships.real.large.png db/diagram.png

purge-queues:
	cd utils/purge_queues && \
	pip install -r requirements.txt && \
	python ./purge-queues.py

reset-alarms:
	cd utils/reset_alarms && \
	pip install -r requirements.txt && \
	python ./reset-alarms.py

clean-os-indices:
	curl -w "\n%{http_code}" --location --request DELETE 'https://search-sracollector-opensearch-bbcrkwlcfb2fjb7psquiefeg2a.eu-central-1.es.amazonaws.com/cwl-sra-collector-*' \
		--header 'Content-Type: application/json' \
		--data '{ "query": { "match_all": {} } }'

clean-builds:
	./utils/clean_temp/clean_temp.sh
```

Figure 6.6: Some Targets At SRA-Collector Makefile

The main advantage of working with *GNU Make* is that the targets can be used both while developing in an *Ubuntu* local machine, and in *GitHub Actions* platform, as the latter is based on *Docker* images running *Ubuntu* too. Thus, the project have a *Makefile* with all the *targets* that will be equally used in any platform.

8. PostgreSQL

SRA-Collector persists records from any data inbound event. When the scientist sends a query, passing through all responses got from the various APIs used by the system, all of them store rows in a relational database to keep track of the process. As it is cloud native and uses *RDS*, one of the supported engines by the *AWS* service should be selected. *PostgreSQL* [65] stands out as the most appropriate due the following reasons:

- It is a popular choice to be used with *Python3* [66]. Furthermore, the library *psycopg2* [67] is a de facto standard to interact from *Python3* code to *PostgreSQL* and is constantly patched and upgraded [68].
- Being an open-source engine, the price on a *RDS* instance running *PostgreSQL* is much lower that running any licensed database engine available as *Oracle* or *Microsoft SQL Server* [69].
- It has a strong reputation of reliability and performance [65].

9. Flyway

During the development of *SRA-Collector*, the database schema, in other words, the structure of tables and relationships among them, evolved rapidly. To control and keep track of all these changes, *Flyway* [70], an open-source database migration tool was utilized.

Flyway allows to keep database schema migration history, so all the tables' structure and relationships among them, is tracked in the version control system of the project [71] as illustrated in Figure 6.7.

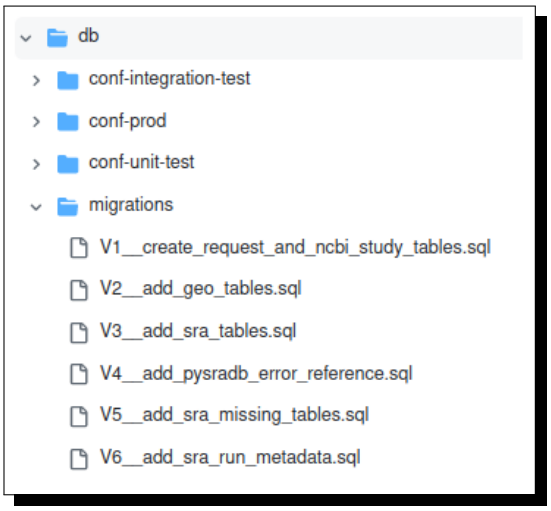

Figure 6.7: SRA-Collector Database Migration Files

Flyway also provides a CLI to apply schema changes with some advanced functions as rollbacks for buggy migrations. The CLI is used both in local development [72] and in CI/CD environment [73] in its dockerized version, as it was the most portable interface [72].

10. Quality Assurance

Several technologies have been used to assure the quality of the project and prevent malfunctioning releases. Following the "Test Pyramid" strategy [74], tests have been clustered in two groups: unit tests and integration tests.

10.1. Unit Tests

All the code that *Python3 Lambda* functions contain define data pipeline success. As a consequence, code changes done in any of the nine functions that make up the process should be carefully tested individually.

The most well-known testing tool for *Python3* is *Pytest*. Accordingly, a *Pytest* suite [75] was built in *SRA-Collector* as depicted in Figure 6.8. It was used to assert that given a set of input parameters, each *Lambda* function provides the expected outcome. One of the needs that arose during unit test suite creation was to have a mock database to simulate the interactions of *Lambda* functions. At the beginning, a different schema was created in *RDS* for these tests, but resulted in an unsustainable burden on test execution time, specially when the amount of test increased. As a result, *H2* [76] database engine was used during unit tests runs.

With *H2*, a simple local file takes the place of the database, so it was recreated on every new execution to prevent inconsistencies using *Flyway* scripts. Also, every interaction from data pipeline code with *H2* was extremely fast as both the code and the database were hosted in the same machine.

Mocking approach [77] was also used for supplanting NCBI APIs, as it seemed not fair to use production resources of NCBI while testing, specially as certain quotas per consumer are established. Likewise, mocking technique was also used with *Pysradb* library calls, for exactly the same reason, as this package does NCBI API calls on the background.

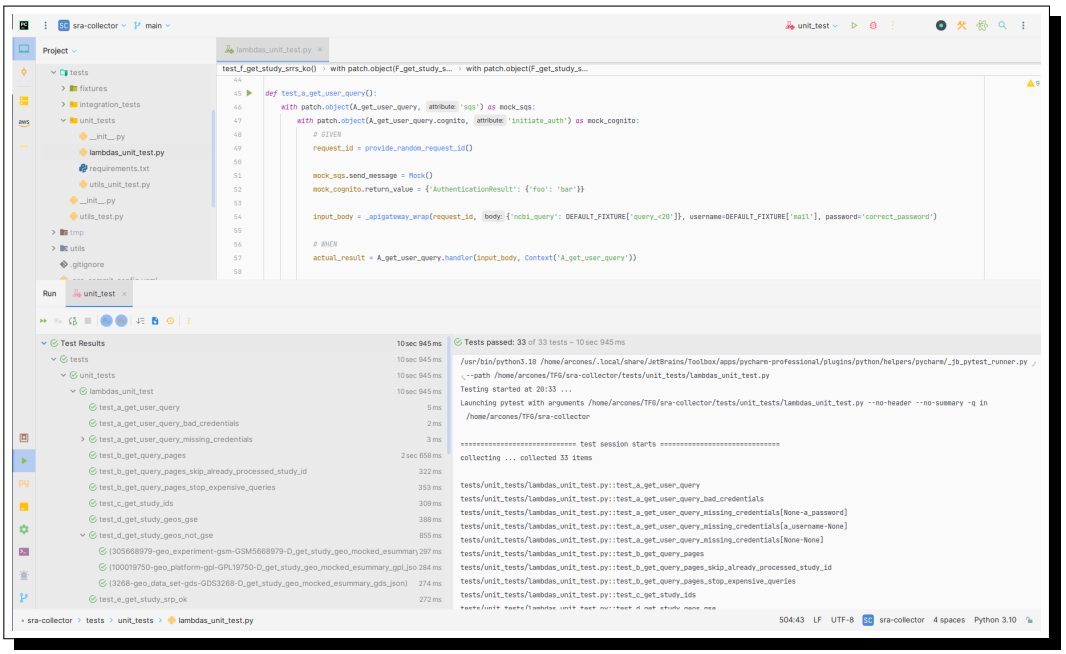

Figure 6.8: SRA-Collector Unit Tests Suite

Finally, unit test passing was added as a requisite for the infrastructure deployment in *GitHub Actions* CI/CD workflow [78] so any attempt for releasing should be compliant.

10.2. Integration Tests

During the evolution of the system, having only unit tests was shown to fall short on certain scenarios related to the configuration applied to *Lambda* functions. Apart from the *Python3* code, any *Lambda* function requires some setup on timeout, memory assigned, environment variables, logging and so on.

Unfortunately, unit tests do not cover such tweaks as they focus merely on the code, so on some releases where unit tests gave the green light to the infrastructure deployment, the data pipeline failed.

Besides, the design decision to use *H2* for unit testing, even though it seemed convenient, introduces an additional divergence between the production environment and the testing one.

Thereupon, an integration test suite was created to complement unit test. The requisites for this new suite were to cover as much scenarios uncovered by unit test suite as possible, with minimal overlapping.

The framework that seemed most suitable was *AWS Severless Application Model* [79], hereafter SAM. SAM provides both a client and a server. The server is responsible for emulating *Lambda* functioning locally, while the client, in the shape of a CLI, is responsible for sending the orders to the server like *"call Lambda A with this input"*. This interaction is shown in Figure 6.9.

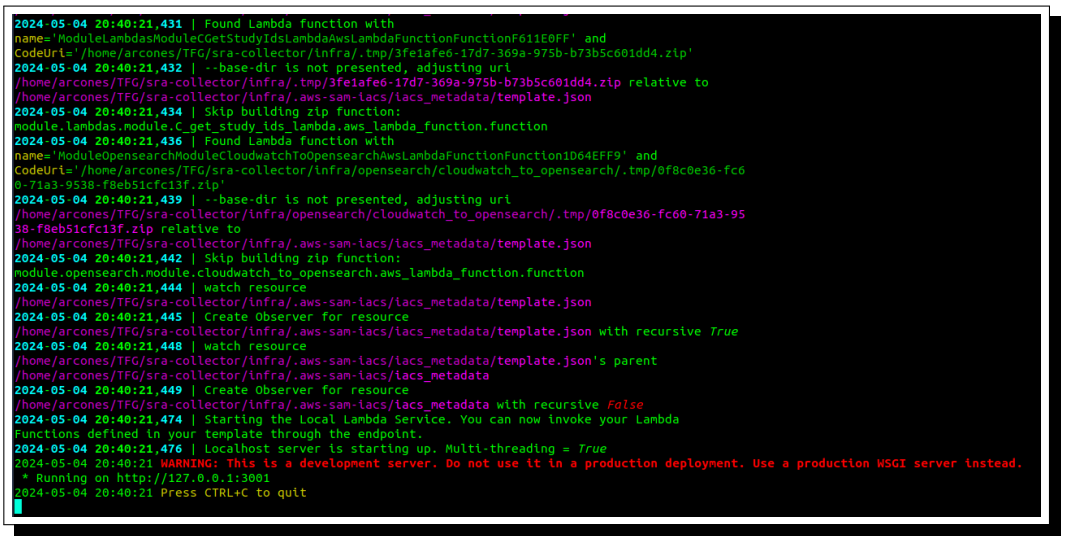

Figure 6.9: SAM Server Infrastructure Waiting For Calls

Before starting to do test assertions, the same challenge as in unit testing was faced, namely, how to configure a data storage that mocks *RDS* so *SAM Lambdas* interact with it. For this, a different data schema was used in *RDS*, which was the same setup that initially was set in unit testing. The advantage of this solution is that the database engine is the same as in production, hence connecting libraries and methods. The disadvantage was that every test run must connect to the cloud, and this affects greatly to execution time, but it was a trade off assumed as, following the testing pyramid [74] approach, the amount of integration tests were notoriously lower than unit tests.

Finally, once *Lambda* functions were running in *SAM* server, *Pytest* was again used to do assertions on exit codes and expected outcomes [80].

Similarly, the CLI of *SAM* was integrated in *GitHub Actions* workflow [81], so any *git* commit is evaluated for compliance.

11. Pysradb

As commented in Section 1, *Pysradb* library was used inside the data pipeline to outsource some transformations. This library abstracts the user from the requests needed to be done to NCBI *E-utilities* on complex transformations, like the GEO to SRA which is a matching that happens between these databases of NCBI.

The library demonstrated its reliability, as during some load tests where several millions of transformations were done, it only failed on certain inputs for which NCBI *E-utilities* was also failing, so *Pysradb* was chosen *not to reinvent the wheel* [82].

Moreover, the library has a public bug tracker [83] available where all problems that appeared during load tests are posted and ready to be patch by any open-source contributor. A screenshot of the public documentation page of the library is attached in Figure 6.10.

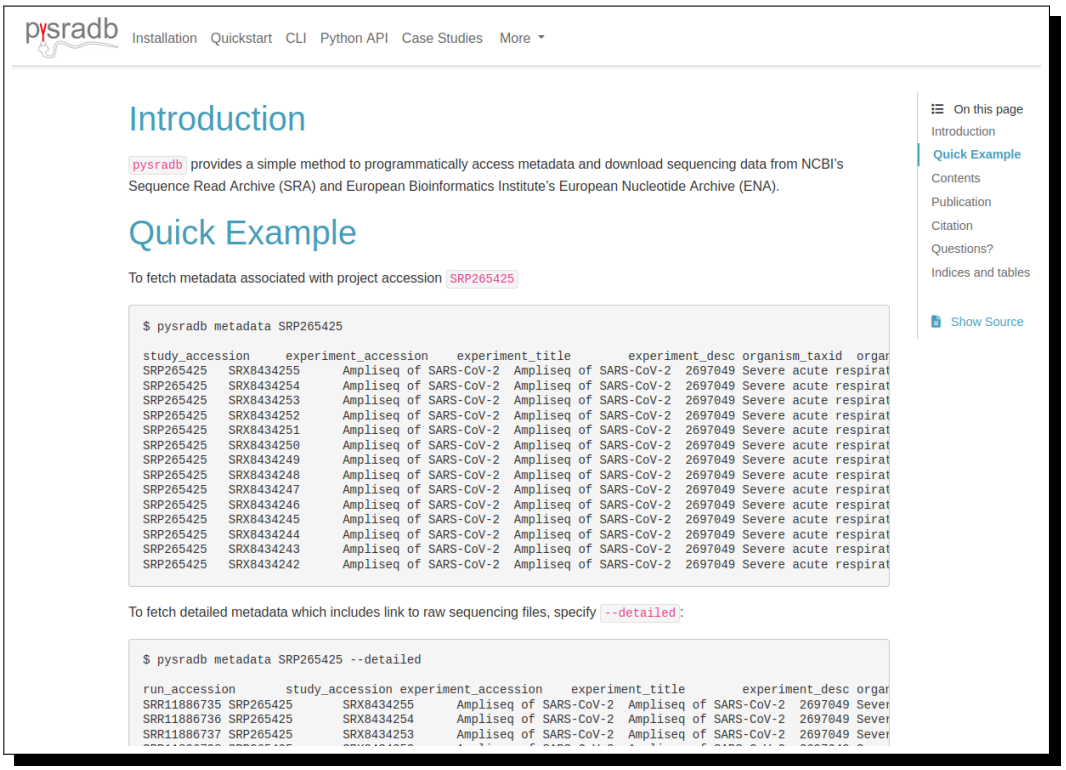

Figure 6.10: Pysradb Documentation

12. OpenSearch

Being the system a compendium of several infrastructure entities, monitoring it was not an easy task. At early stage, this was done by checking logs directly in *CloudWatch* console. *CloudWatch* console provide functionality to tail the logs [84], depicted in Figure 6.11, and also to do queries in SQL-like syntax using *CloudWatch Insights* [85].

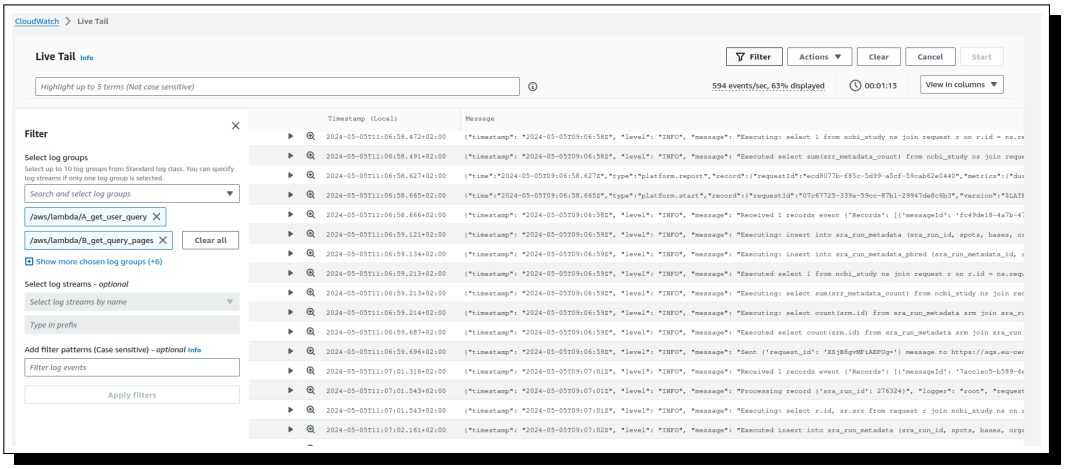

Figure 6.11: CloudWatch Live Tail Functionality

These methods fell short very quickly, so a research was done on observability platforms. The best candidate to integrate with *AWS* workloads is *OpenSearch* service. *OpenSearch* [86] is an open-source fork of the popular *Elasticstack* [87] search & analytics engine. *OpenSearch* system provides an API for submitting the logs that the developer wants to store as well as and then some highly-configurable dashboards to query the logs and filter them. In these dashboards visualizations like graphs, counters, histograms, etc were used. One of the dashboards created for the project is shown in Figure 6.12.

Furthermore, other interesting capabilities of *OpenSearch* have been used, as index automatic creation [88]. Indices in *OpenSearch* are the storage for incoming logs, so they are configured with the expected logs fields and data types in JSON format. With index auto-creation, the index is automatically created as soon as the first log, known in *OpenSearch* argot as *"document"*, arrives.

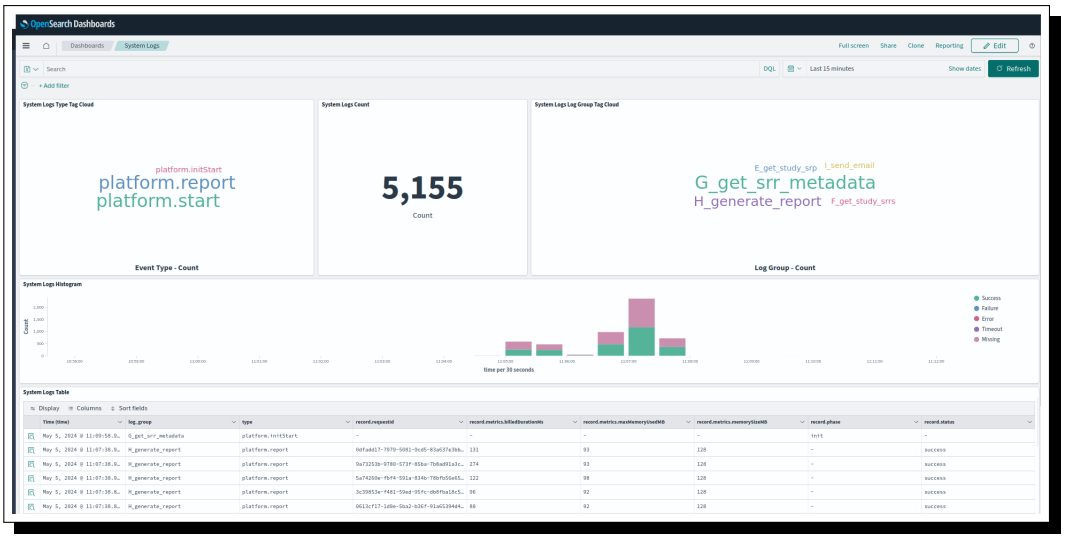

Figure 6.12: OpenSearch System Logs Dashboard

Another interesting feature of *OpenSearch* is document purge. As the instance of *OpenSearch* used was part of *AWS* free tier [89], its disk space was quite limited, so it was necessary to purge old documents from time to time. This feature made the task very easy and it was included in *Makefile* [90].

13. NCBI E-Utilities & Traces Service

NCBI *E-Utilities* [9] is an HTTP API that can be used to retrieve programmatically data from NCBI databases. It provides the data either in XML or in JSON on retriever demand. JSON was chosen as the management of these data structures in *Python3* is easier. This API has nine predefined functions [91] from which only *esearch* and *esummary* were directly used. Some of the other functions are used under the hood by *Pysradb* library but not from *SRA-Collector* explicitly.

Apart from *E-Utilities*, NCBI Traces Service [13] was used too. This is also an HTTP API but here the output format cannot be selected and is always XML, as is shown in Figure 6.13. This service provides all statistical metadata of a given SRR sample. Unlike *E-Utilities*, this API does not expose a DTD document for its XML responses, meaning that the data type of the fields and even which fields are expected and which are not, cannot be known beforehand. This setback was tackled by doing numerous tests to discover all expected results from the API.

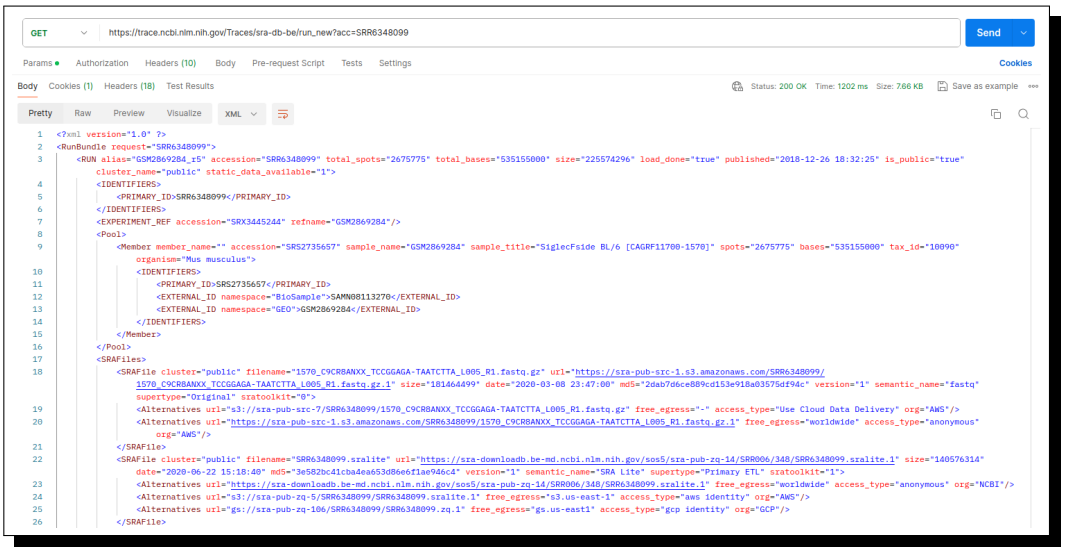

Figure 6.13: NCBI Traces Service

14. PyCharm IDE

To code the whole project, specially concerning source files in *Python3* and *Terraform*, *PyCharm* [92] IDE was used. *PyCharm* provides syntax highlighting, code completion, refactoring, code search and debugging capabilities among others. Also, *PyCharm* is very flexible as a wide amount of plugins are available to complete its functionality. Among the plugins used, it's worth to mention *AWS Toolkit* [93].

Combining *AWS Toolkit* and *PyCharm* runners, which can be configured in debugging mode, it was possible to debug line by line *Python3* code of *Lambda* functions. This configuration was not used from project inception and, at the beginning, the code refinement was accomplished deploying *Lambda* functions in the cloud and analyzing the logs, sometimes adding extra logs around certain functions likely to fail. This process was arduous, especially since any code change needed to be deployed in order to be tested. Once local setup to debug was configured, feedback cycles shortened greatly and thus, development times were reduced.

15. Pre-commit

With *Pre-commit* [94] hooks, a project managed by *git* can automate to run certain scripts before committing the code to the repository. As shown in the official documentation [95] there is a wide variety of hooks that cover tasks such as linting the code in different programming languages, upgrading outdated dependencies or ensuring the compliance of certain files.

Pre-commit action runs implicitly before *git commit* action and depending on the implementation of the hook that is running, it either fixes the issues found or makes the commit fail when some action is expected from the developer. This automatic service makes the tool very interesting to maintain certain basics like code readability and format adherence.

The hooks chosen for *SRA-Collector* [96] mostly focus on keeping certain standards in *Python3* and *Terraform* code but also ensure *GitHub Actions* configuration files are correct.

16. Swagger & OpenAPI

At some point, *SRA-Collector* was ready to be called by HTTP, using *cURL* [97] client for example. This method has the advantage of the ubiquity of *cURL*, which can be easily installed in any machine and comes in the default installation of many. The main disadvantage is accessibility, as only experienced command-line users will be comfortable working with it. Besides that, *cURL* does not provide documentation of the API being called, being up to the user to research about the input shape, i.e, path parameters, authentication headers, etc.

To make the HTTP interface self-explanatory and accessible in one go, it was decided to expose it with *Swagger* [98]. *Swagger* covers a dual purpose: it uses *OpenAPI* [99] specification to detail the HTTP interface of the API and in addition it allows to interact directly with it in its frontend [100]. As a conclusion, a more user-friendly interface was provided, covering all functional requirements regarding the UI. The result is illustrated in Figure 6.14.

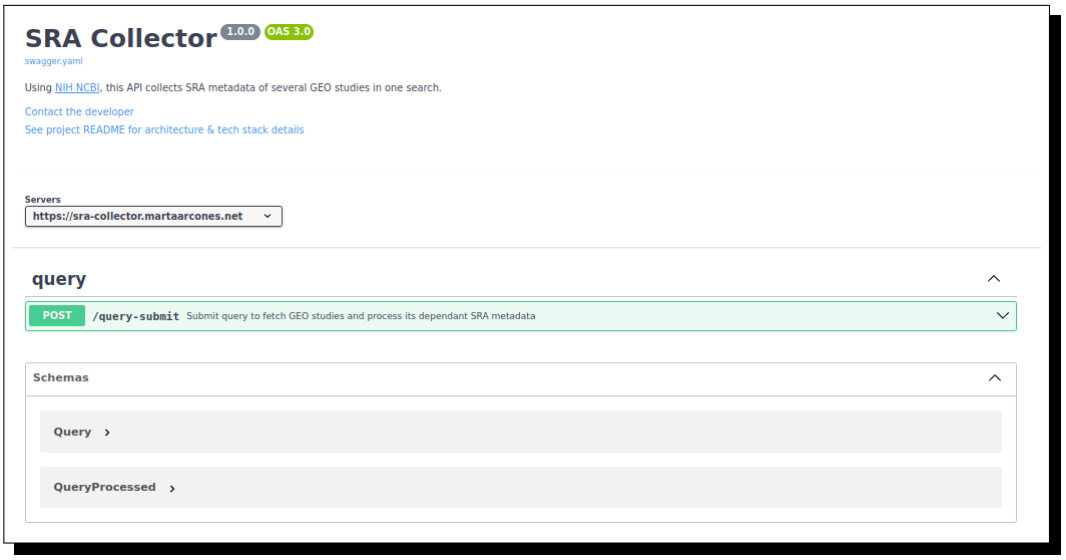

Figure 6.14: Swagger UI Interface

17. SchemaSpy

One particularly useful piece of documentation of any software project is its database entity relationship diagram, also known as ERD. In rapidly evolving projects, maintain this ERD up-to-date can be time consuming so a research was done on how to automate database diagram generation.

SchemaSpy [101] came forward as an easy to integrate option for such endeavours. This tool connects to the database and scans the schema tables and relations. To connect, the tool needs a configuration file with all the parameters [102]. The execution of this tool was integrated in CI/CD pipeline using *Makefile* [103] and *GitHub Actions* [104].

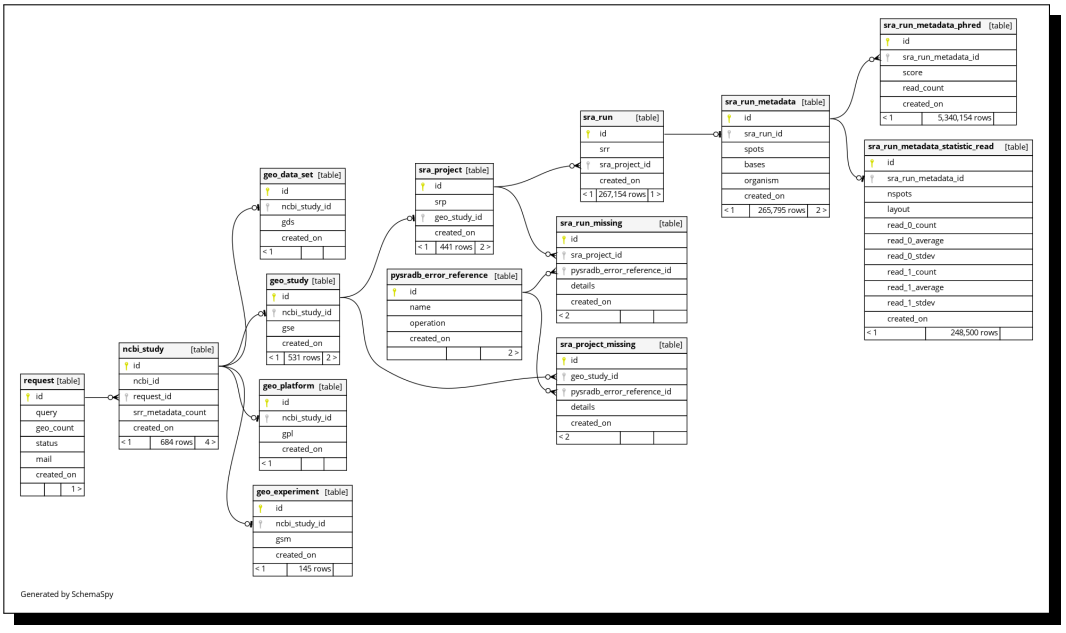

Figure 6.15: DB Diagram Created With SchemaSpy

To avoid tedious installations, *Docker* image of *SchemaSpy* [105] was used. The configuration file is provided as input to the image when building the container and then an ERD as the one in Figure 6.15 is generated as an output.

18. Ubuntu

Ubuntu Linux distribution [106] was used both as an operative system to code and as a build machine in *GitHub Actions* [107]. Having the same *Linux* distribution in both environments provides full compatibility on the *Makefile* commands used directly in local while development and later as a part of the CI/CD pipeline.

An advantage of *Ubuntu* itself is that being the most popular *Linux* distribution available for general purposes, most of the code samples and documentation available online for any application installation and usage consider it carefully. So usually it is only needed to follow the documentation of any tool to make it work in *Ubuntu* machine.

7. System Design and Implementation

1. Design Patterns

During the development of the system, the following design patters have been taken into account:

1.1. Fail-fast and Fail-safe Systems

These two design principles are opposite, whether a fail-fast [108] system will stop operation immediately when unexpected scenario happens, a fail-safe [109] will instead mitigate the issue or try to recover.

During the development of *SRA-Collector*, a hybrid approach among both principles was utilized, as depending on the scope of the problem it was sometimes more convenient to fail rapidly, as represented in Figure 7.1, while in other situations a retry mechanism was adopted.

```python
def handler(event, context):
    if event:
        logging.info(f'Received {len(event["Records"])} records event {event}')

        batch_item_failures = []
        sqs_batch_response = {}

        for record in event['Records']:
            try:...
            except Exception as exception:
                batch_item_failures.append({'itemIdentifier': record['messageId']})
                logging.error(f'An exception has occurred in {handler.__name__}: {str(exception)}')
        sqs_batch_response['batchItemFailures'] = batch_item_failures
        return sqs_batch_response
```

Figure 7.1: Except Block In Lambda Function To Fail Fast

Fail-fast was extremely convenient for *Lambda* functions. It was verified that when an issue occurs, is smarter to bounce back the message that causes the problem to the *SQS* queue and start processing another one. In particular, this approach validated its effectiveness with a recurrent scenario of *RDS* lacking enough connection threads. This happens when several *Lambda* functions are trying to connect with the database simultaneously. Eventually the database pool is rejecting connections, so *Lambda* functions start to fail. The input messages are therefore pushed back to the origin *SQS* were a fail-safe mechanism is set, as the *SQS* configuration retries the message when some time has passed.

Also, as *Lambda* code is where the logic of the system resides, having them failing fast enhances testing experience because the feedback cycles in case of problems is shorter. Additionally, in case of failure on production, *Lambda* functions emit error logs that are shipped automatically to *OpenSearch* and the *Lambda* error ratio metric of *CloudWatch* increases.

In contrast, for external dependencies to the *Lambda* functions, like any of the HTTP APIs used, a conservative approach was put in place as no control could be exerted over them. This fail-safe approach was implemented using an exponential backoff wait between a big amount of retries [110].

1.2. Single-responsibility Principle

The development of a data pipeline based on *Lambda* functions was a consequence of applying the single responsibility principle [111]. Single responsibility principle can be summarised as *"Do one thing. Do it well.*

Do it only" [112]. This motto is behind the development of any of the functions, being them purely focus on one task and ignorant of the rest.

INPUT	FUNCTION	OUTPUT
request_id & filename	I_send_mail	mail
request_id	H_generate_report	request_id & filename
sra_run_id	G_get_srr_metadata	request_id
sra_project_id	F_get_study_srrs	sra_run_id
geo_entity_id	E_get_study_srp	sra_project_id
ncbi_study_id	D_get_study_geo	geo_entity_id
request_id & retstart & retmax	C_get_study_ids	ncbi_study_id
request_id & mail & query	B_get_query_pages	request_id & retstart & retmax
mail & password & query	A_get_user_query	request_id & mail & query

Figure 7.2: SRA-Collector Functions In Protocol Tower Shape

Besides, the data pipeline behaves as networking protocol stack, being *Lambda* functions as layers in the sense of having a well defined interface to receive the inputs and in which they generate the outputs, as shown in Figure 7.2.

1.3. Best-effort Delivery

During some load tests done over the system, where some of the biggest NCBI queries were used, there were detected some studies that made *Pysradb* library fail because of the unexpected responses of NCBI servers.

As will not be worth to invalidate a query of several millions of studies just because a couple of them cannot be retrieved, a best-effort approach [113] was implemented. The system then stores for further reference those "well-known *Pysradb* errors" and prevent them to progress any further in the data pipeline as outlined in Figure 7.3.

```
-- Full traceability of errored entities
select query as scientist_query,
       gse as entity,
       name as pysradb_error_name,
       operation as pysradb_operation,
       details as error_details
from sra_project_missing
join sracollector.pysradb_error_reference per on per.id = sra_project_missing.pysradb_error_reference_id
join sracollector.geo_study gs on sra_project_missing.geo_study_id = gs.id
join sracollector.ncbi_study ns on ns.id = gs.ncbi_study_id
join sracollector.request r on ns.request_id = r.id
```

Output | Full traceability of errored entities | Tx | 39 rows

scientist_query	entity	pysradb_error_name	pysradb_operation	error_details
alzheimer and rna seq	GSE199762	ATTRIBUTE_ERROR	gse_to_srp	'NoneType' object has no attribute 'rename'
alzheimer and rna seq	GSE237495	VALUE_ERROR	gse_to_srp	All arrays must be of the same length
alzheimer and rna seq	GSE212251	KEY_ERROR	gse_to_srp	'Summary'
alzheimer and rna seq	GSE173316	VALUE_ERROR	gse_to_srp	All arrays must be of the same length
alzheimer and rna seq	GSE226663	KEY_ERROR	gse_to_srp	'Summary'
alzheimer and rna seq	GSE141509	ATTRIBUTE_ERROR	gse_to_srp	'NoneType' object has no attribute 'rename'

Figure 7.3: SRA-Collector Error Traceability Persisted

This technique resembles the Internet protocol, making all things possible to progress on a task but giving no guarantees over the outcome of it [114].

1.4. Asynchronous Event-driven Architecture

During the initial investigation phase mentioned in Section 1, a spike application called *Kilombo* was created [115]. This application was a web server running *Python3* and *FastAPI* [116]. The server expected an input query and do the rest of the process: listing all NCBI studies for the query, using *Pysradb* for the conversions,

etc. At the end the application returned the expected data to the consumer as HTTP response body as depicted in Figure 7.4.

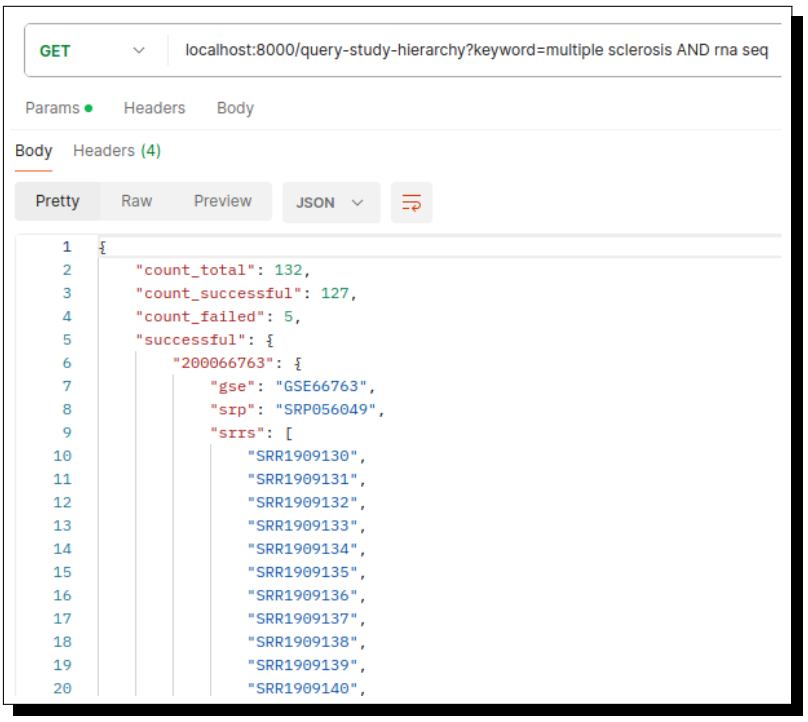

Figure 7.4: Kilombo Application In Use

This approach saturated quite fast as the amount of steps that the queries require to be converted on statistical metadata is big, and those steps are not particularly fast. For example some requests with just around 80 NCBI studies required almost half an hour to complete, and it was not acceptable as to open an HTTP channel with the client all this time has great risk of failing by any network interruption.

Subsequently, the architecture shifted to asynchronous [117] and event-driven [118]. The process is split in single responsibility tasks, such as *"from this GSE, obtain its equivalent SRP"* or *"from these SRR obtain its statistical metadata"* as commented in Section 1.2. The initiator of all these tasks is the reception on an event in a *SQS* queue, that is configured to trigger a specific *Lambda* function. When the function finishes, it pushes an output event in another queue and so on.

The scientist that submits the query receives an acknowledgement from the system of its reception but nothing more at that initial moment. Then, when all the processing of the data pipeline finishes, a e-mail is sent to the scientist with the results.

1.5. Cloud-native Computing

Once the final architecture model was selected, a research was done on how to implement such event-driven system. There are solutions like *RabbitMQ* [119] that can be installed in any machine and used as message broker to decouple services but the tuning can be tricky. In the other hand, cloud provider message queuing services like *AWS SQS* [39] or *Azure Queue Storage* [120] seemed more out-of-the-box in terms of configuration and, when using serverless consumers provided by the same cloud provider, the integration between the queue and them is transparent.

These advantages and the possibility to use the *Python3* code implemented for *Kilombo* in serverless functions lean the design towards a cloud-native approach [121].

1.6. Test-driven Development

Once embraced the event-driven architecture, the responsibility of every *Lambda* function was clear. For example between the *SQS* queue that holds messages containing the SRPs and the *SQS* queue that contains messages with SRRs, a *Lambda* function should do this transformation.

As these *Lambda* functions cover one single responsibility, it was easy to work using a test-driven development approach [122], from now on TDD. Following TDD, the tests for each *Lambda* function were created before the function itself, establishing clearly which is the expected outcome of the function for the different scenarios, i.e, a happy path scenario, a flawed input or an unexpected outcome of the calculations performed inside the function. An example of such tests is evidenced in Figure 7.5.

```python
def test_c_get_study_ids(lambda_client):
    with PostgreConnectionManager() as (database_connection, database_cursor):
        # GIVEN
        request_id_1 = provide_random_request_id()
        input_body_1 = json.dumps({'request_id': request_id_1, 'retstart': 0, 'retmax': 500})
        store_test_request( database_holder: (database_connection, database_cursor), request_id_1, ncbi_query: 'multiple sclerosis AND Astrocyte-produced HB-EGF and WGBS')

        request_id_2 = provide_random_request_id()
        input_body_2 = json.dumps({'request_id': request_id_2, 'retstart': 0, 'retmax': 500})
        store_test_request( database_holder: (database_connection, database_cursor), request_id_2, ncbi_query: 'stroke AND single cell rna seq AND musculus')

        # WHEN
        invocation_result = lambda_client.invoke(FunctionName='C_get_study_ids', Payload=sqs_wrap( bodies: [input_body_1, input_body_2], dumps=True))

        # THEN
        lambda_response = json.loads(invocation_result['Payload']._raw_stream.data.decode('utf-8'))

        assert 'errorMessage' not in lambda_response
        assert 'errorType' not in lambda_response

        assert lambda_response['batchItemFailures'] == []
```

Figure 7.5: Integration Tests Example

1.7. Security by design

Security by design development approach [123] was used to prototype and implement the system. This strategy aims for creating a system which is foundationally secure, instead of adding security at the end of development phase. An alternative naming for this principle is shift left security [124].

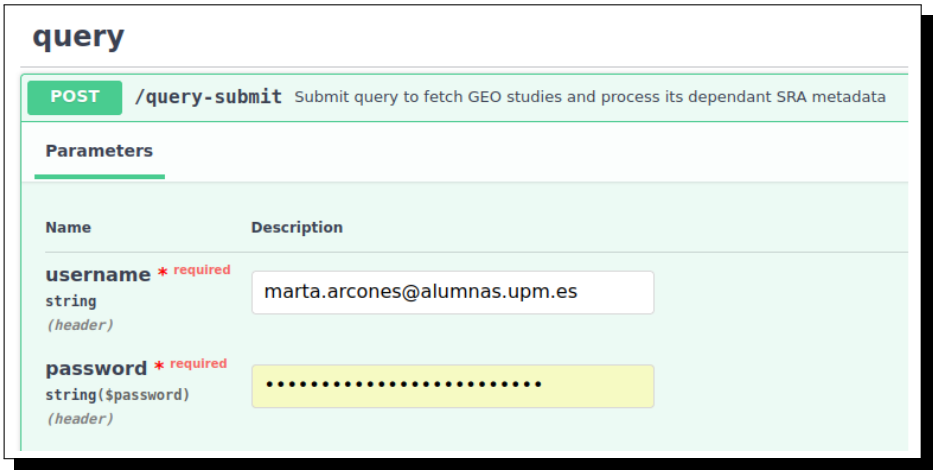

Figure 7.6: Credentials UI Mandatory Parameter

With this principle in mind, all systems exposed to the Internet like *Swagger* facade of Section 16 or *OpenSearch* dashboards count with some mechanisms to be used only to who is authorized to do so. For example, user and password authentication is required to send queries to the system as indicated in Figure 7.6 and IP filtering is in place to access *OpenSearch* dashboards. Moreover, database is using credentials automatically generated by *AWS* which additionally rotates the password every two weeks.

Remaining secrets are stored in a dedicated security vault in the cloud, rather than in plain text alongside the source code. Additionally, the various components of the system are designed with a mutually suspicious, least privilege approach [31] in mind. This means that any entity, such as a *Lambda* function, is only permitted to interact with expected entities, and there is no carte blanche on the operational policies created. Consequently, the policy documents for each entity, as exemplified in Figure 7.7, were narrowed as much as possible.

```
resource "aws_iam_role_policy" "input_sqs_policy" {
  count = var.queues.input_sqs_arn == null ? 0 : 1
  name  = "input_sqs_policy"
  role  = aws_iam_role.lambda_role.name
  policy = jsonencode({
    Version = "2008-10-17"
    Statement = [
      {
        Action = [
          "sqs:ReceiveMessage",
          "sqs:DeleteMessage",
          "sqs:GetQueueAttributes"
        ]
        Effect   = "Allow"
        Resource = var.queues.input_sqs_arn
      },
    ]
  })
}
```

Figure 7.7: IAM Lambda Policy

Furthermore, to access the cloud itself, there is a permission segregation in place, being CI/CD and local development access configured with different roles so a specific auditing can be done on each credential. For local development, multi-factor authentication is in place having to enter a password and a code received in a mobile device. Additionally, credentials expire automatically having passed some time.

Creation of users and password for scientists have been taken seriously too, as with a custom developed solution, thought easier to manage like some credentials table in database, the risk of data leak is greater. For this reason, a cloud built-in solution *Cognito* was used as it provides auditing, credentials rotation, e-mail verification, etc.

1.8. Infrastructure As Code

Cloud-native approach triggered the ensuing design pattern, infrastructure as code [125]. As one of the non-functional requirements commented in Section 2.2 is that the system needs to be replicated easily, all cloud services configuration needs to be stored in some type of living documentation that allows to recreate the system in other *AWS* cloud account.

Infrastructure as code approach emphasizes typical source code development techniques for the system operations field [126]. For example, in the past, setting up a mail server in a remote datacenter often involved running scripts to spin up the service in a *SSH* session with the facility. While that approach may be swift, it lacks repeatability unless accompanied by a script documenting the steps followed, which is where infrastructure as code technologies come into play as the evolution of such scripts.

With an infrastructure as code project, any provisioning or changing of the infrastructure is recorded in version control system and using the CI/CD system, the written configuration can be deployed in the cloud. As a gotcha, infrastructure as code languages use mostly declarative programming approach [127] which makes the code task even simpler as the control flow is outsourced to the tool, in this case *Terraform*, having to focus only to describe infrastructure elements in compliant syntax.

```
resource "aws_apigatewayv2_domain_name" "apigateway_domain_name" {
  domain_name = "sra-collector.${var.subdomain}"

  domain_name_configuration {
    certificate_arn = aws_acm_certificate.certificate.arn
    endpoint_type   = "REGIONAL"
    security_policy = "TLS_1_2"
  }

  depends_on = [aws_acm_certificate_validation.certificate_validation]
}

resource "aws_route53_record" "record" {
  name    = aws_apigatewayv2_domain_name.apigateway_domain_name.domain_name
  type    = "A"
  zone_id = data.aws_route53_zone.zone.zone_id

  alias {
    name                   = aws_apigatewayv2_domain_name.apigateway_domain_name.domain_name_configuration[0].target_domain_name
    zone_id                = aws_apigatewayv2_domain_name.apigateway_domain_name.domain_name_configuration[0].hosted_zone_id
    evaluate_target_health = false
  }
}
```

Figure 7.8: Terraform Declarative Programming

1.9. Don't Repeat Yourself

The code principle don't repeat yourself [128] has shaped development mindset during the construction of the system. The fact of having nine *Lambda* functions, many of them doing similar stuff like connecting with database or fetching and pushing messages to *SQS* queues made clear that some reusable pieces of code were needed. As a consequence, some libraries for doing common tasks were developed [129].

- **db-connection library**: packed with utility functions to connect with database with retries in place as shown in Figure 7.9. It also provides support for executing read, write and bulk write operations with the lowest amount of boilerplate code in the client as possible. Moreover, depending on the present environment variables at runtime it connects with the production schema or with the testing one in a transparent way for the consumer.

- **s3-helper library**: providing helper methods to upload and download files from *S3* buckets.

- **sqs-helper library**: coming with streamlined functions to send single or batch messages to *SQS* queues. It also provides environment support for using production queues or testing ones transparently for the invoking *Lambda*.

```
class DBConnectionManager:
    def __init__(self):...

    def __enter__(self):...

    def initialize_postgres(self):
        while self.database_connection is None:
            try:
                self.database_connection = psycopg2.connect(self.connection_string)
                logger.info(f'Successfully connected with database in attempt #{self.connection_attempts}')
            except OperationalError as operationalError:
                if self.connection_attempts == self.MAX_TRIES:
                    logger.error(f'Not able to connect with database after {self.connection_attempts} attempts')
                    logger.debug(str(operationalError))
                    raise operationalError
                else:
                    logger.debug(f'Not able to connect with database in attempt #{self.connection_attempts}')
                    logger.debug(str(operationalError))
                    self.connection_attempts += 1
                    time.sleep(1)
        self.database_cursor = self.database_connection.cursor()

    def __exit__(self, exc_type, exc_val, exc_tb):...
```

Figure 7.9: Initialize PostgreSQL Function At DB Connection Library

2. Implementation

The process can be divided into two planes, analogous to networking planes [130, 131]: the data plane, which encompasses all actions related to product handling, and the control plane, which covers system observability.

2.1. Data Plane

The data plane spans from the moment a scientist issues a query in the *Swagger* UI to the delivery of the e-mail containing the results. A diagram presenting the inner components of this data pipeline is presented in Figure 7.10.

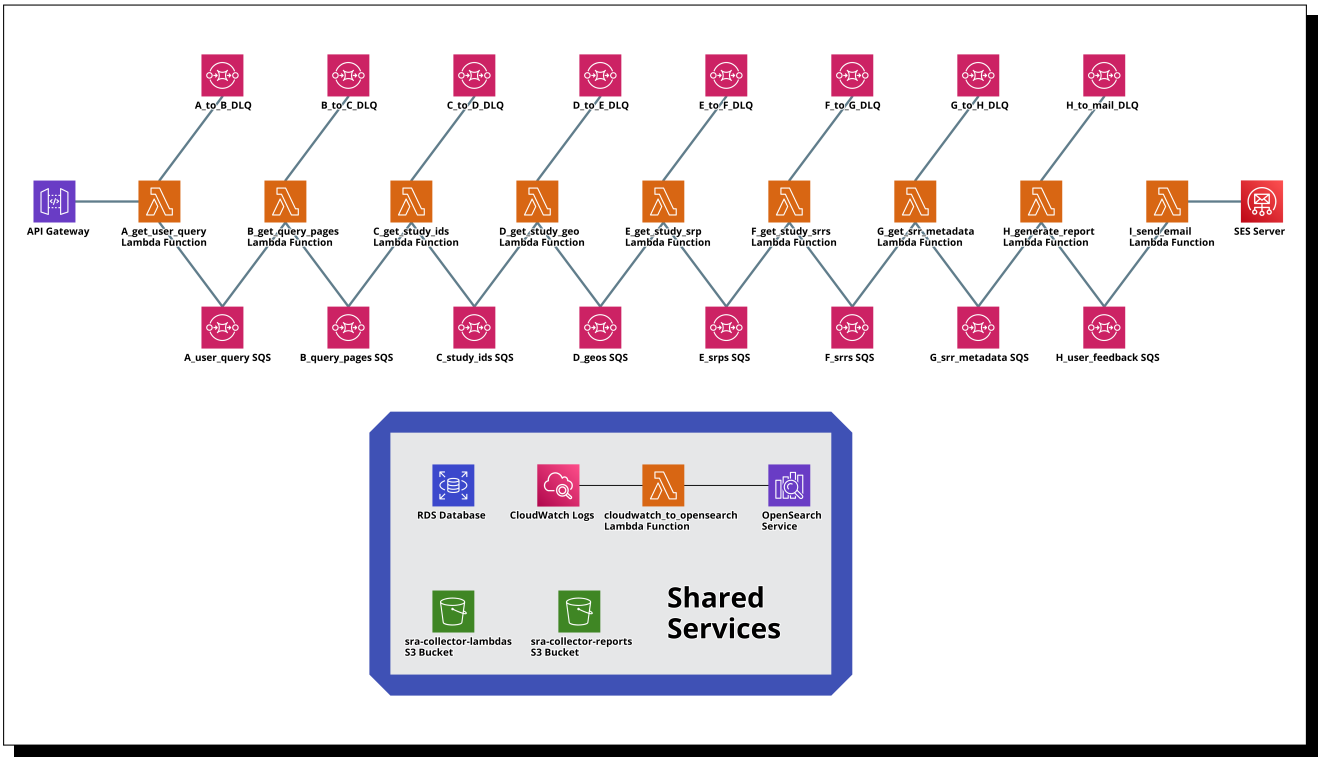

Figure 7.10: Data Pipeline Backbone

In the following sections, the process is described step by step. The database entities specifications used in each operation can be also found in Appendix 8.

Authentication & Request ID Generation

As depicted in Figure 7.6, credentials are required to issue any request to the system. If either wrong or missing credentials are provided, the system emits a 401 HTTP response code for *"Unathorized"* [132]. Hence, some credentials must be acquired before any interaction. This duty corresponds to the administrator, who will issue credentials by creating a new user and password in *Cognito* user pool as shown in Figure 7.11.

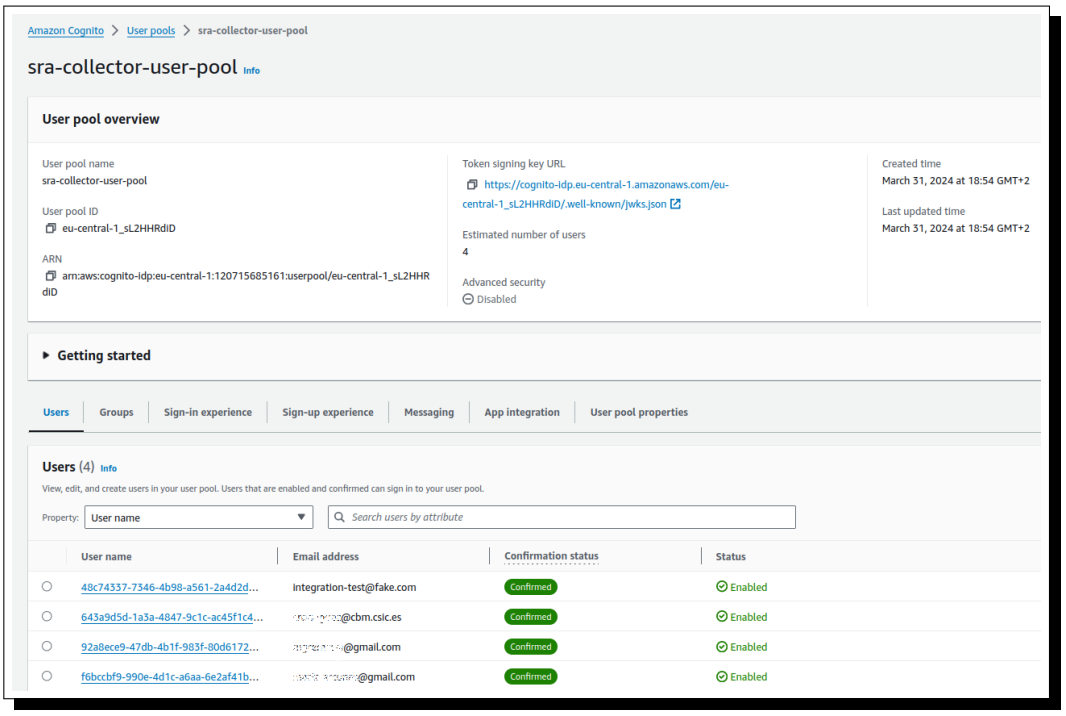

Figure 7.11: Cognito User Pool

Once the scientist has the credentials, he or she can create a request using *Swagger* UI that will be processed by the first *Lambda* function [133], see Figure 7.12 for its code detail. The name of this function is `A_get_user_query`.

```python
def handler(event, context):
    try:
        logging.info(f'Received event {event}')

        if authenticate_user(event['headers']):
            request_id = event['requestContext']['requestId']

            request_body = json.loads(event['body'])

            ncbi_query = request_body['ncbi_query']

            message_body = {'request_id': request_id, 'ncbi_query': ncbi_query}

            SQSHelper(sqs, context.function_name).send(message_body={**message_body, 'mail': event['headers']['username']})

            return {'statusCode': 201, 'body': json.dumps(message_body), 'headers': {'content-type': 'application/json'}}
        else:
            return {'statusCode': 401}
    except Exception as exception:
        logging.error(f'An exception has occurred in {handler.__name__}: {str(exception)}')
        raise exception
```

Figure 7.12: Detail Of Lambda `A_get_user_query` Code

The code is intentionally straight forward and simple, as the *API Gateway* service that links the UI request with the backend has a hard limit for providing a response of thirty seconds. Hence, the code is simply authenticating using *Cognito* API and placing a *SQS* message like the one depicted in Figure 7.13. The `request_id` value is generated out-of-the-box by *API Gateway* so the *Lambda* function is simply reusing it.

```
{
    "request_id": "XTlTBhAzFiAEPyQ=",
    "ncbi_query" : "autism and children and rna seq",
    "mail": "margarita.salas@cbm.csic.es"
}
```

Figure 7.13: Output Message Of Lambda `A_get_user_query`

As noted in the output message, the body includes a primary key of the database, specifically the `request_id`. This pattern is consistently utilized across all integrations between the *Lambdas* of the system, as it has proven effective in distinguishing between entities. Considering that multiple scientists may be querying for overlapping NCBI records simultaneously, or that different NCBI studies can relate to the same SRRs samples, primary keys serve as the only reliable means to differentiate them. Last but not least, they also facilitate the proper construction of the hierarchy with foreign keys, beginning with the `request_id`.

Get Study Count & Pagination

The message created by `A_get_user_query` is placed in a standard *SQS* queue. Standard *SQS* queues do not prevent duplicate messages due to their distributed nature, in which several servers store the same message to prevent data loss [134]. This results in occasions where the same message is processed twice or more. To avoid this, *AWS* also provides *"First-In-First-Out"*, FIFO for short, queues [135]. FIFO queues are consequently slower on the processing as they keep the same data loss prevention while ensuring the message can only be processed once and in order. Unfortunately, FIFO queues are 25 % more expensive than standard queues [136]. Henceforth, for budget and performance reasons, standard queues were used across the project and the application code performs some checks to ensure the idempotent nature of the process [137] as illustrated in the code detail of Figure 7.14.

```python
def is_request_pending_to_be_processed(database_holder, request_id: str, ncbi_query: str) -> bool:
    try:
        statement = 'select id from request where id=%s and query=%s and status=%s;'
        parameters = (request_id, ncbi_query, 'PENDING')
        return not database_holder.execute_read_statement(statement, parameters)
    except Exception as exception:
        logging.error(f'An exception has occurred in {is_request_pending_to_be_processed.__name__}: {str(exception)}')
        raise exception


def store_request_in_db(database_holder, request_id: str, ncbi_query: str, study_count: int, mail: str, status: str = None):
    try:
        statement = 'insert into request (id, query, geo_count, mail, status) values (%s, %s, %s, %s, %s) on conflict do nothing;'
        parameters = (request_id, ncbi_query, study_count, mail, status if status is not None else 'PENDING')
        database_holder.execute_write_statement(statement, parameters)
    except Exception as exception:
        logging.error(f'An exception has occurred in {store_request_in_db.__name__}: {str(exception)}')
        raise exception
```

Figure 7.14: Detail Of Lambda `B_get_user_query` Code

When it comes about `B_get_query_pages` [138], the process starts by checking if the message was processed before or not and, according to this:

- If the message was not processed before, there is no record for the `request_id` received yet in `REQUEST` table, so:
    1. A call is made to the NCBI *E-utilities* in order to retrieve a count of the number of studies returned by the query (study information is not retrieved in this operation, solely the numerical amount).
    2. As *E-utilities* strictly defines the limit of items each request can resolve [9], a pagination process is initiated, where the parameters `retstart` and `retmax` play a key role. For instance, if a query returns three thousand studies with a page size of five hundred, six messages are generated to *SQS*.

Each message maintains the page size of five hundred as `retmax`, while `retstart` increments from zero, to five hundred and one, and so forth.

3. Messages containing the pairs of `retstart` and `retmax` alongside with the `request_id` are pushed to the next *SQS* queue. An example of such messages can be found in Figure 7.15.

- If the message was processed before, the record exists in `REQUEST` table and therefore, the system does nothing.

```
{
    "request_id": "XTlTBhAzFiAEPyQ=",
    "retstart" : 501,
    "retmax" : 500
}
```

Figure 7.15: Output Message Of Lambda `B_get_user_query`

Study Identifiers Retrieval

Next *Lambda* function to come into action is `C_get_study_ids` [139]. The process this function follows is:

1. It picks the message that contains the primary key of `REQUEST` table and pagination information.
2. Using the primary key, it queries the `REQUEST` table for the text query that the scientist has submitted.
3. It emits a *esearch* request to the *E-utilities* NCBI server asking for study identifiers related to the query, among the pagination range provided.
4. It stores the study identifiers retrieved in `NCBI_STUDY` table.
5. It sends a message with each `NCBI_STUDY` primary key just stored in the database to the next queue as shown in Figure 7.16.

```
{
    "ncbi_study_id": 45
}
```

Figure 7.16: Output Message Of Lambda `C_get_study_ids`

```
def store_study_ids_in_db(database_holder, request_id: str, ncbi_ids: [int]):
    try:
        statement = 'insert into ncbi_study (request_id, ncbi_id) values (%s, %s) on conflict do nothing returning id;'
        parameters = [(request_id, ncbi_id) for ncbi_id in ncbi_ids]
        return database_holder.execute_bulk_write_statement(statement, parameters)
    except Exception as exception:
        logging.error(f'An exception has occurred in {store_study_ids_in_db.__name__}: {str(exception)}')
        raise exception


def get_query(database_holder, request_id: str):
    try:
        statement = 'select query from request where id=%s;'
        return database_holder.execute_read_statement(statement, (request_id,))[0][0]
    except Exception as exception:
        logging.error(f'An exception has occurred in {get_query.__name__}: {str(exception)}')
        raise exception
```

Figure 7.17: Detail Of Lambda `C_get_study_ids` Code

Due to the unreliable nature of *SQS* message uniqueness explained above, the study identifier store operation is done using a *PostgreSQL* extension to provide an idempotent experience, which is adding `ON CONFLICT DO`

`NOTHING` suffix to any `INSERT` statement. A detail of the code executing statements with this feature is shown in Figure 7.17.

GEO Identifier Acquisition

The message with the NCBI study identifier is then picked by `D_get_study_gse` function. The main purpose of this *Lambda* is to obtain the GEO database identifier that corresponds to the study. The relation between NCBI study identifiers and GEO identifiers is one to one. The GEO identifier obtained can be one of four kinds:

- **GSE**: a GEO study. This is the most interesting identifier for this project, as entities of GSE kind can be linked with SRP identifiers in SRA database.
- **GSM**: a GEO sample. Whenever a GSM appears as identifier the system stores it but there is no progress in the data pipeline with them.
- **GPL**: a GEO platform. The same logic applies as for GSM identifiers.
- **GDS**: a GEO dataset. The same logic applies as for GSM identifiers.

The *Python3* classes created for these identifiers can be shown in Figure 7.18.

```python
class GeoEntityType(Enum):
    GSE = {'table': 'geo_study', 'short_name': 'gse'}
    GSM = {'table': 'geo_experiment', 'short_name': 'gsm'}
    GPL = {'table': 'geo_platform', 'short_name': 'gpl'}
    GDS = {'table': 'geo_data_set', 'short_name': 'gds'}


class GeoEntity:
    def __init__(self, identifier: str):
        self.identifier = identifier.upper()
        self.geo_entity_type = self.set_type()

    def set_type(self):
        if self.identifier.startswith('GSE'):
            return GeoEntityType.GSE
        elif self.identifier.startswith('GSM'):
            return GeoEntityType.GSM
        elif self.identifier.startswith('GPL'):
            return GeoEntityType.GPL
        elif self.identifier.startswith('GDS'):
            return GeoEntityType.GDS
        else:
            raise ValueError(f'Unknown identifier prefix: {self.identifier}')
```

Figure 7.18: Detail Of Lambda `D_get_study_gse`

The process of obtaining a GEO identifier necessitates an authenticated call to the NCBI *E-Utilities* service. While authentication is optional, NCBI servers offer enhanced quotas for authenticated requests. Specifically, non-authenticated requests are limited to three requests per second, whereas authenticated requests enjoy a quota of ten requests per second [91]. Consequently, an API key was acquired and stored to facilitate calls, ensuring a better performance.

This key is not a secret itself, as it appears in the URL path parameters clearly, as in `/eutils/esummary.fcgi?db=gds&api_key=theAPIKey` example. But it represents an identity in NCBI servers, as it is associated with an account created using their registration form [140]. Therefore, it was stored in *AWS Secrets Manager* and the *Lambda* function retrieves it before making the call.

The most notable performance characteristic of this *Lambda* function is its ability to read messages from the input *SQS* in batches. For this reason, multiple input messages can be processed together during each execution of the function. Leveraging this capability, calls to *E-Utilities* are made to request information for several studies at once, exploiting the API's support for batch processing. This approach significantly optimizes resource usage, as it allows the reuse of connection sockets. Thus, each request to NCBI from the *Lambda* function can potentially fetch data for approximately fifty studies at a time.

The process that *Lambda* `D_get_study_gse` follows is:

1. Using the received primary keys of `NCBI_STUDY`, it queries the table for the NCBI server study identifiers. Therefor, from `NCBI_STUDY.ID` it fetches `NCBI_STUDY.NCBI_ID`.
2. It fetches the API key from *Secrets Manager*.
3. It builds a request URL to NCBI that contains the API key and all the studies the system will be querying for.
4. It extracts the desired GEO entities, i.e, GSE, GPL, GDS or GSM from the response.
5. It stores the retrieved GEO entities in the database, with each GEO entity type having its corresponding table. For instance, the `GEO_STUDY` table is designated to store GSE identifiers. Similar to the previous *Lambda* function, this one also utilizes a *PostgreSQL* extension to skip storage if it has already been completed.
6. It sends messages with the primary keys just stored for GSE entities to the next *SQS* as illustrated in Figure 7.19.
7. For studies not containing GSE entities, stated differently, having only GDS, GSM or GPL identifiers, it sets the `SRR_METADATA_COUNT` field in `NCBI_STUDY` to zero, signaling that no SRR metadata will be possible to extract.

```
{
    "geo_entity_id": 988
}
```

Figure 7.19: Output Message Of Lambda `D_get_study_geo`

From GEO Identifier to SRA Project Identifier

The *Lambda* that picks the messages with GSE identifiers is `E_get_study_srp`. This is the first function making use of *Pysradb* library allowing an straight forward transformation of the GSE from GEO database into an SRP entity from SRA one.

As mentioned in previous sections, heavy load tests have been run against the library, making it convert several millions of GSE and SRA entities. During those testing, certain recurrent failures where detected when NCBI servers do not have the mapping properly settled. For this reason, the conversion operation is coded with some fallbacks for classifying known errors in case those happen. A detail on the process for capturing these errors in detailed in Figure 7.20.

```
except AttributeError as attribute_error:
    logging.info(f'For {gse}, pysradb produced attribute error with name {attribute_error.name}')
    store_missing_srp_and_srr_count(database_holder, geo_entity_id, PysradbError.ATTRIBUTE_ERROR, str(attribute_error))
except ValueError as value_error:
    logging.info(f'For {gse}, pysradb produced value error: {value_error}')
    store_missing_srp_and_srr_count(database_holder, geo_entity_id, PysradbError.VALUE_ERROR, str(value_error))
except KeyError as key_error:
    logging.info(f'For {gse}, pysradb produced key error: {key_error}')
    store_missing_srp_and_srr_count(database_holder, geo_entity_id, PysradbError.KEY_ERROR, str(key_error))
```

Figure 7.20: Detail Of Lambda `E_get_study_srp`

The process executed by `E_get_study_srp` is as follows:

1. With primary key of the GSE row, firstly it gets from `GEO_STUDY` table the GSE NCBI identifier. In other words, using `GEO_STUDY.ID` it retrieves the `GEO_STUDY.GSE`.
2. It uses *Pysradb* library to convert the GSE into an SRP. This correspondence is always one to one.
3. If the conversion goes smooth:
   - It stores in the `SRA_PROJECT` table the result employing an idempotent storage action facilitated by *PostgreSQL* extensions.
   - It sends to the next *SQS* queue a message with the primary key of the `SRA_PROJECT` row just inserted as shown in Figure 7.21.
4. If the conversion fails:
   - It stores in the database the information about the problematic GSE and the error that arose.
   - It sets the `SRR_METADATA_COUNT` field in `NCBI_STUDY` to zero with the same purpose as in previous *Lambdas*.

```
{
    "sra_project_id": 125
}
```

Figure 7.21: Output Message Of Lambda `E_get_study_srp`

SRA Runs For Each SRA Project

Similarly, next *Lambda* that comes into play is `F_get_study_srrs` that does also a *Pysradb* operation to extract the SRRs from the SRP that was acquired in the step before. The process is analogous to `E_get_study_srp` in terms of trying a successful extraction and if not, categorize and save the errors in database.

```
try:
    srp = get_srp_sra_project(database_holder, sra_project_id)
    raw_pysradb_data_frame = SRAweb().srp_to_srr(srp, detailed=True)
    srrs = list(raw_pysradb_data_frame['run_accession'])
    srrs = [srr for srr in srrs if srr.startswith('SRR')]

    if srrs:
        logging.info(f'For {srp}, SRRs are {srrs}')
        sra_run_ids = store_srrs_and_count(database_holder, srrs, sra_project_id)
        message_bodies = [{'sra_run_id': sra_run_id} for sra_run_id in sra_run_ids]
        SQSHelper(sqs, context.function_name).send(message_bodies=message_bodies)
    else:
        logging.info(f'No SRR for {srp} found via pysradb')
        store_missing_srr_and_count(database_holder, sra_project_id, PysradbError.NOT_FOUND, details: 'No SRR found')
except AttributeError as attribute_error:
    logging.info(f'For SRP with id {sra_project_id}, pysradb produced attribute error with name {attribute_error.name}')
    store_missing_srr_and_count(database_holder, sra_project_id, PysradbError.ATTRIBUTE_ERROR, str(attribute_error))
except TypeError as type_error:
    logging.info(f'For SRP with id {sra_project_id}, pysradb produced type error with name {type_error}')
    store_missing_srr_and_count(database_holder, sra_project_id, PysradbError.TYPE_ERROR, str(type_error))
```

Figure 7.22: Detail Of Lambda `F_get_study_srrs`

The algorithm, of which an excerpt can be found in Figure 7.22, can be summarised as follows:

1. The *Lambda* function receives the primary key of an SRP row so, firstly, it gets from `SRA_PROJECT` table the SRA NCBI identifier, put differently, it goes from `SRA_STUDY.ID` to `SRA_PROJECT.SRA`.
2. The function then uses *Pysradb* library to extract the SRRs belonging to the SRP. This correspondence is one SRP to n SRRs.
3. If the conversion goes smooth:

- It stores in idempotent manner the results in SRA_RUN table.
- It sends to the next *SQS* queue as much messages as SRRs have been extracted, each of them containing one of the primary keys just inserted in SRA_RUN. An example of one of these messages can be shown in Figure 7.23.

4. If the conversion fails:
   - It stores in database information about the problematic SRP and the error that arose.
   - It sets the SRR_METADATA_COUNT field in correspondent NCBI_STUDY to zero with the same purpose as in previous *Lambdas*.

```
{
    "sra_run_id": 4578
}
```

Figure 7.23: Output Message Of Lambda F_get_study_srrs

SRA Run Metadata Extraction

```
def get_srr_metadata(srr: str) → SRRMetadata:
    try:
        srr_metadata = SRRMetadata(srr)

        url = f'https://trace.ncbi.nlm.nih.gov/Traces/sra-db-be/run_new?acc={srr_metadata.srr}'
        response = http.request( method: 'GET', url).data
        root = ElementTree.fromstring(response)

        run_node = root.findall('.//RUN')
        if len(run_node) > 0:
            run_element = run_node[0]
            spots = run_element.get('total_spots')
            bases = run_element.get('total_bases')
            if spots is not None and spots.isdigit():
                srr_metadata.set_spots(int(spots))
            else:
                srr_metadata.set_spots(0)
            if bases is not None and bases.isdigit():
                srr_metadata.set_bases(int(bases))
            else:
                srr_metadata.set_bases(0)

        quality_count_node = root.findall('.//RUN/QualityCount')
        if len(quality_count_node) > 0:
            quality_count_element = quality_count_node[0]
            phred = {}
            for quality_element in quality_count_element:
                phred[int(quality_element.get('value'))] = int(quality_element.get('count'))

            srr_metadata.set_phred(phred)
```

Figure 7.24: Detail Of Lambda G_get_srr_metadata

With the message having SRA_RUN primary key that F_get_study_srrs function placed in the queue, G_get_srr_metadata does the last operation involving NCBI servers. After retrieving the actual SRR NCBI identifier and the request_id, it calls *Traces Service* that, given an SRR, returns in XML format all the statistical metadata. A detail on the parsing activity done by the *Lambda* can be found in Figure 7.24.

Following the extraction process, each metadata statistic is transformed into a row within the SRA_RUN_METADATA table, which has two associated child entities. The first is SRA_RUN_METADATA_PHRED, featuring a one to many optional relationship, that stores phred score values for the SRR. The second is

SRA_RUN_METADATA_STATISTIC_READ, characterized by a one to one optional relationship, that records the sequence reads information for the sample.

In SRA_RUN_METADATA_STATISTIC_READ the data is stored denormalized. Each SRR metadata record typically includes one or two reads, never more. Initially, a one to many relationship was considered but it was not suitable for the next step of the process, where a CSV file is generated. In that step, each SRR is represented as a row, and all the metadata is included in columns. Henceforth, the normalization approach was swapped.

In this *Lambda*, *PostgreSQL* extensions are also used to guarantee that only one row per each unique statistic is inserted at a time, thus preventing duplicates.

The last thing the function does is checking that the sum of SRA_RUN_METADATA rows stored is equal to the aggregation of NCBI_STUDY.SRR_METADATA_COUNT for all the studies fetched by the initial request. This integrity check is actually asking if the extraction of the metadata is done for request_id as a whole. If the counts match, G_get_srr_metadata places an *SQS* message in the next queue with the request_id of which the metadata was just obtained as illustrated in Figure 7.25.

```
{
    "request_id": "XTlTBhAzFiAEPyQ="
}
```

Figure 7.25: Output Message Of Lambda G_get_srr_metadata

CSV Report Generation

The function H_generate_report catches the message with the request_id and simply does an *SQL* query that creates one row per each SRR with all the statistical metadata details, as shown in Figure 7.26. Afterwards, it creates a CSV report with all those rows and locates it in a *S3* bucket.

```
def generate_report(database_holder, request_id: str) -> [[]]:
    try:
        statement = ('SELECT R.QUERY, NS.NCBI_ID, GS.GSE, SP.SRP, SR.SRR, SRM.SPOTS AS TOTAL_SPOTS, '
                     'SRM.BASES AS TOTAL_BASES, SRM.ORGANISM, SRMSR.LAYOUT, '
                     'SUM(CASE WHEN SRMP.SCORE >= 30 THEN SRMP.READ_COUNT ELSE 0 END) / SRM.BASES AS PHRED_READ_COUNT_FROM_30, '
                     'SUM(CASE WHEN SRMP.SCORE >= 37 THEN SRMP.READ_COUNT ELSE 0 END) / SRM.BASES AS PHRED_READ_COUNT_FROM_37, '
                     'SRMSR.READ_0_COUNT, SRMSR.READ_0_AVERAGE, SRMSR.READ_0_STDEV, '
                     'SRMSR.READ_1_COUNT, SRMSR.READ_1_AVERAGE, SRMSR.READ_1_STDEV '
                     'FROM SRA_RUN_METADATA SRM '
                     'JOIN SRA_RUN_METADATA_PHRED SRMP ON SRM.ID = SRMP.SRA_RUN_METADATA_ID '
                     'JOIN SRA_RUN_METADATA_STATISTIC_READ SRMSR ON SRMSR.SRA_RUN_METADATA_ID = SRM.ID '
                     'JOIN SRA_RUN SR ON SR.ID = SRM.SRA_RUN_ID '
                     'JOIN SRA_PROJECT SP ON SR.SRA_PROJECT_ID = SP.ID '
                     'JOIN GEO_STUDY GS ON SP.GEO_STUDY_ID = GS.ID '
                     'JOIN NCBI_STUDY NS ON GS.NCBI_STUDY_ID = NS.ID '
                     'JOIN REQUEST R ON NS.REQUEST_ID = R.ID '
                     'WHERE R.ID =%s '
                     'GROUP BY R.QUERY, NS.NCBI_ID, GS.GSE, SP.SRP, SR.SRR, SRM.SPOTS, SRM.BASES, '
                     'SRM.ORGANISM, SRMSR.LAYOUT, SRMSR.READ_0_COUNT, SRMSR.READ_0_AVERAGE, '
                     'SRMSR.READ_0_STDEV, SRMSR.READ_1_COUNT, SRMSR.READ_1_AVERAGE, SRMSR.READ_1_STDEV; ')
        return database_holder.execute_read_statement(statement, (request_id,))
    except Exception as exception:
        logging.error(f'An exception has occurred in {generate_report.__name__}: {str(exception)}')
        raise exception
```

Figure 7.26: Detail Of Lambda H_generate_report

To prevent duplicates due to the already mentioned nature of *SQS* input messages, before executing this query, the *Lambda* checks that the REQUEST.STATUS row for the incoming identifier is in PENDING status, meaning that the CSV has not been generated yet. For this reason, just after pushing the file to *S3*, it updates that STATUS column to EXTRACTED value. The last operation done is pushing a message to the last queue including, apart from the request_id, the name of the CSV file just generated. The body of the message can be seen in Figure 7.27.

```
{
    "request_id": "XTlTBhAzFiAEPyQ=",
    "filename": "Report_XTlTBhAzFiAEPyQ=.csv"
}
```

Figure 7.27: Output Message Of Lambda `H_generate_report`

E-Mail Delivery

The final *Lambda* function in the system, `I_send_email`, is relatively straightforward. It utilizes the `request_id` to obtain the submitter e-mail stored in the `MAIL` column of the `REQUEST` table, and uses the `filename` field also provided in the message to download the *S3* file. Subsequently, it composes an e-mail attaching the downloaded file that is sent using *AWS SES* service. The last operation of this function, and of the data pipeline as a whole, is updating the `REQUEST.STATUS` to `SENT` so, in case the message is received again, no more e-mails are delivered.

The code that generates the e-mail with the CSV as attachment is detailed in Figure 7.28.

```python
def compose_mail(request_id: str, recipient: str, attachment_path: str = None, reason: str = None) -> str:
    if (attachment_path is None) == (reason is None):
        raise ValueError('Either attachment_path or reason should be provided, not both')

    mail = MIMEMultipart()
    mail['Subject'] = f'Results for {request_id} query to SRA-Collector'
    mail['From'] = 'noreply@martaarcones.net'
    mail['To'] = recipient
    mail['Bcc'] = os.environ.get('WEBMASTER_MAIL')

    if attachment_path is not None:
        with open(attachment_path, 'rb') as attachment:
            attachment = MIMEApplication(attachment.read())
            attachment.add_header( _name: 'Content-Disposition', _value: f'attachment; filename="{os.path.basename(attachment_path)}"')
            mail.attach(attachment)
    else:
        mail_body = MIMEText(reason, _subtype: 'plain')
        mail.attach(mail_body)

    return mail.as_string()
```

Figure 7.28: Detail Of Lambda `I_send_email`

2.2. Control Plane

The control plane concerns all the operations and infrastructure configured to make the system observable, i.e, the administrator can evaluate if the internals of the system are correct through by its external outputs [141].

CloudWatch Alarms

There are two types of alarms configured [142]. The first type has to be with the *Lambda* functions [143] and checks the error ratio in their invocations. Every five minutes, the alarm gets the sum of invocations and the sum of errors for the last time window and calculates the error rate percentage. If the error rate exceeds a particular threshold, an alert is delivered by e-mail to the administrator using *SNS* service. If the error ratio goes then back to normal status, another message is sent to inform accordingly.

During the development, specially in the load testing phase, it was asserted that each function, depending on its responsibilities, will need a different error ratio threshold. For example, converting a GSE coming from GEO DataSets into an SRP from SRA is more likely to fail than the operation of converting a query provided by the scientist on a set of NCBI study identifiers. Therefore the thresholds were adjusted for each *Lambda* independently [144].

The second type of alarm configured was a dead letter queue, hereafter DLQ, overfill. DLQs store all messages that *Lambda* functions were unable to process, leading to a process failure. As explained before, some retries

are configured in each queue similarly as shown in Figure 7.29. So, in case a message ends up in a DLQ, it means that the retries limit has been reached without success.

```
resource "aws_sqs_queue" "D_geos" {
  name                       = "D_geos"
  visibility_timeout_seconds = 600
  redrive_policy = jsonencode({
    deadLetterTargetArn = aws_sqs_queue.D_to_E_DLQ.arn,
    maxReceiveCount     = 3
  })
}
```

Figure 7.29: Dead Letter Queue Configuration

Thus, the DLQ overfill alarms check how many messages are in DLQ and, if the amount is greater than zero, the alerts are triggered. The particularities of each operation of the system have been addressed with the retries configuration on each queue. Thus, common configuration for all DLQ alarms resulted effective.

Similarly to *Lambda* error ratio alarms, in case that any of the DLQ alerts is triggered or goes back to normal status, *SNS* service is used to send an e-mail to the administrator.

An interesting functionality that was also developed to speed up testing process over the alarms is a script to reset them to normal status as depicted in Figure 7.30 image.

```
metric_alarms = cloudwatch.describe_alarms()['MetricAlarms']

metric_alarms_names = [metric_alarm['AlarmName'] for metric_alarm in metric_alarms]

logger.info(f'There are {len(metric_alarms_names)} alarms to restart')

for metric_alarm_name in metric_alarms_names:
    alarm_reset_result_code = cloudwatch.set_alarm_state(
        AlarmName=metric_alarm_name,
        StateValue='OK',
        StateReason='Manually restarted'
    )['ResponseMetadata']['HTTPStatusCode']
    if alarm_reset_result_code == 200:
        logger.info(f'Alarm reset for {metric_alarm_name} was successful')
    else:
        logger.error(f'Alarm reset for {metric_alarm_name} failed')
```

Figure 7.30: Detail Of Purge Alarms Script

OpenSearch Service

OpenSearch has been a great aid for checking the progress in the data pipeline and troubleshooting issues, as the dashboards provided and the filtering capabilities are way ahead from *CloudWatch* native functionality. However, to make use of *OpenSearch* service, any log created by *API Gateway* or *Lambda* functions must be forwarded to it.

Significantly, all logs produced by *CloudWatch* are structured [145] meaning that they are in JSON format. This is one of the key requirements to get the best out of an observability platform like *OpenSearch*.

Ideally, the process of shipping the logs to *OpenSearch* should trigger automatically whenever the infrastructure is emitting logs and, once the logs are in the observability server, dashboards should be instantly showing the results.

To achieve this, serverless execution model was used too [38] as the examples *AWS* suggests mark in this direction [63]. Consequently, an additional *Lambda* function is invoked any time a log is received at *CloudWatch* service. The function duty is to push the logs to *OpenSearch* using its HTTP API facade. To optimize the process, this *Lambda* function incorporates a time window during which it collects incoming logs. Once the time window elapses, the function gathers all the accumulated logs, zip them into a single archive, and sends them in bulk.

On top of that, this *Lambda* enriches the logs by adding some metadata to them. In particular, there are three types of logs forwarded by this function:

- **Access Logs**: these logs are generated by *API Gateway* service each time a scientist request is produced. They contain information about the request itself such as the `request_id` generated, the source IP, the user agent, the connection latency, the HTTP response status, etc.

- **System Logs**: these logs are generated by the *Lambda* engine each time a function is triggered. They contain information about the unique identifier of each invocation, the memory used by each *Lambda* function, the billed duration of each run and the end status of the execution, such as success, timeout or failure.

- **Application Logs**: these logs are generated by the *Python3* code within the *Lambda* functions, detailing the progress of various entities, such as studies, GSEs, SRPs, and SRRs inside the data pipeline. The logs are categorized by severity and emitting module, which may include any runtime libraries or the main program itself. Additionally, each log entry contains the identifier of the corresponding *Lambda* invocation, facilitating the correlation with system logs.

Furthermore, for each of the three log types, the exporter *Lambda* tags the logs with an additional field in the JSON document indicating its type. This enables storing each log type in a distinct *OpenSearch* index. This approach is particularly beneficial because the log volumes vary significantly across the three indices, ensuring that a burst in one does not impact the performance of the others. Additionally, maintenance tasks such as reindexing, shrinking or purging can be performed independently on each index.

In addition, for application logs, the exporter *Lambda* includes a tag indicating which of the nine functions in the data pipeline produced the log. Last but not least, because logs are sent in batches to the observability platform, they sometimes arrive with identical timestamps, making ordering by reception timestamp ineffective. To address this issue, the exporter *Lambda* assigns a log offset field, which is a simple integer sequence, as it gathers the logs before shipping them. This ensures that, when multiple logs share the same timestamp in *OpenSearch*, the chronological order can be preserved by including the log offset field in the ordering.

Once all types of logs are stored in *OpenSearch* platform, three dashboards were built, one for each type of logs. As an example, the application logs dashboard is depicted in Figure 7.31.

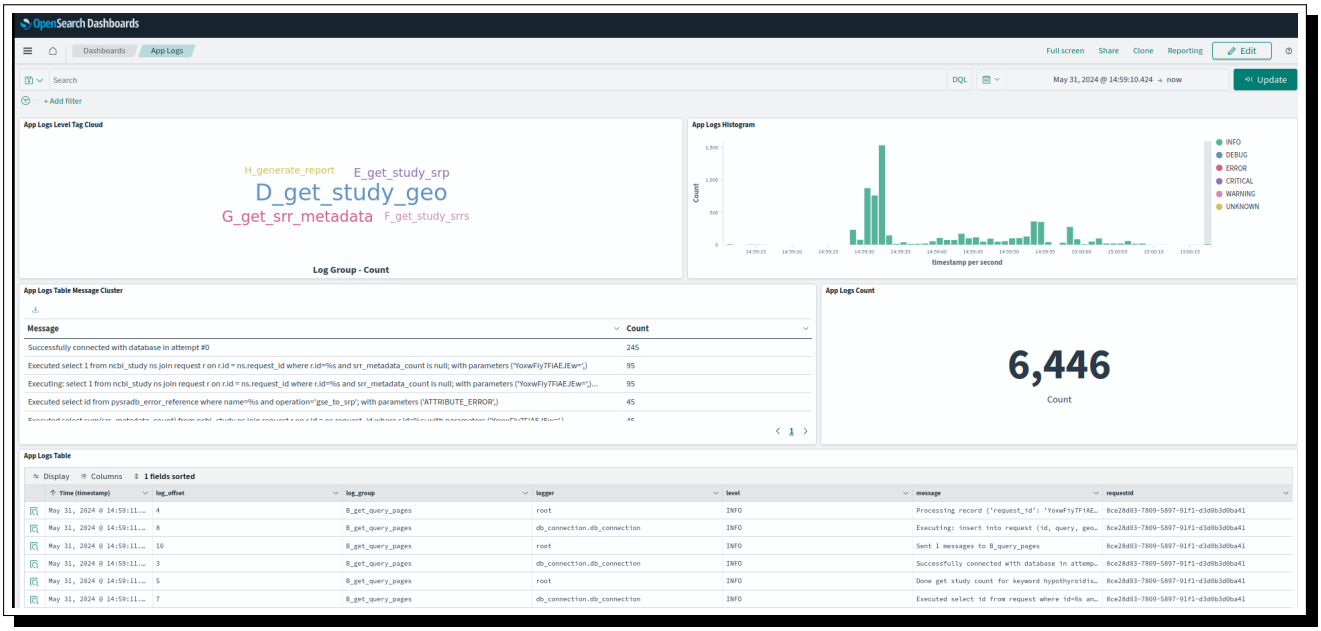

Figure 7.31: OpenSearch Application Logs Dashboard

As illustrated above, dashboards are a composition of panels. The panels used in *SRA-Collector* dashboards are:

- **Histogram**: a vertical bar chart segregated by time. Here, the burst of logs per timestamp can be observed, and each bar can be split in different colors by using another filter. Hence, in access logs the bars are split by HTTP response code, in application logs by log severity and in system logs by status field.

- **Count**: a simple panel with an integer number. It shows the count of logs for the specific time filter introduced.

- **Tag Cloud**: it allows to see how many logs per *Lambda* are present for the time filter selected. The *Lambda* names in the cloud decrease or increase proportionally to the quantity of logs for each function. It is also used to show which kind of system logs are arriving the most.

- **Logs Table**: a simple table showing the logs found for the time filter selected. As logs are structured, their JSON fields are present as columns in the table that can be ordered and filtered.

- **Clean Logs Groups**: oftentimes log messages follow a pattern, for example, the log pattern `''Attempt # to connect to database''` appear for any of the five retries the system has configured, each time with a different number in the position of # symbol. As a result, having those patterns identified as groups allow to filter in and out them by a single action. The purpose the clean logs panel serves for is to show the detected logs patterns and the hits for each one so, apart from the filtering, fast feedback on the most recurrent patterns is provided too.

Once the three dashboards were assembled as sets of the aforementioned panels, it was important to version them, so the effort configuring the dashboards in the *OpenSearch* UI is not lost easily. For this reason, using *OpenSearch* exporting capabilities, a file containing all server entities such as panels, dashboards and indices was generated and versioned in the *GitHub* repository of the project[146].

Similarly to *CloudWatch* alarms, after intense load testing with millions of logs generated, the need of cleaning the indices of *OpenSearch* instance arose. This need is produced by the configuration of the instance itself, being the cheapest instance available at *AWS* [147]. Interestingly, for cleaning the logs of indices in *OpenSearch*, its HTTP API provides also an endpoint. Using this, a *Makefile* target was created specifically for this purpose [148].

```
clean-os-indices:
    curl -w "\n%{http_code}" --location --request DELETE 'https://search-sracollector-opensearch-bbcrkwlcfb2fjb7psquiefeg2a.eu-central-1.es.amazonaws.com/cwl-sra-collector-*' \
        --header 'Content-Type: application/json' \
        --data '{ "query": { "match_all": {} } }'
```

Figure 7.32: Makefile Target For Cleaning OpenSearch Indices

8. Resulting Report

Once the data pipeline process has finished, the scientist will receive an e-mail as presented in Figure 8.1. The traceability is kept by having at the e-mail subject the `request_id` provided to the scientist when the query was submitted using the UI. This is specially relevant if a scientist is creating several queries in parallel. Additionally, it can be noticed that the sender e-mail is a "noreply" account that *SES* service configures out-of-the-box for the mail server instance.

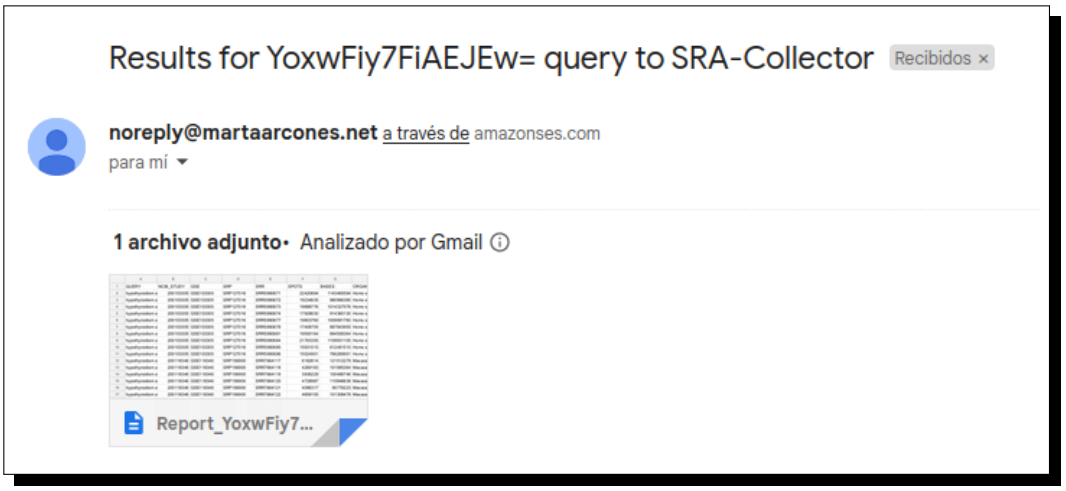

Figure 8.1: Resulting Report Received By E-Mail

Attached to the e-mail is the CSV file, containing seventeen columns of denormalized information:

- **Query**: text field containing the query string submitted by the scientist in the UI.
- **NCBI Study**: text field containing the study identifiers fetched by the text query in NCBI database.
- **GSE**: text field containing the GEO identifier for each NCBI study.
- **SRP**: text field containing the SRA project identifier relative to the GEO entity.
- **SRR**: text field containing the SRA run identifier for each SRP entity.
- **Spots**: integer number representing the sequencing depth of the SRR.
- **Bases**: integer number representing the amount of sequenced bases in the SRR.
- **Organism**: text field containing from which species the SRR was obtained.
- **Layout**: text field containing one of the two possibles sequencing strategies, i.e., single or paired.
- **Phred Score From 30**: float number field showing the percentage of reads above a phred score of 30. In other words, this value shows the percentage of reads with an accuracy of 99.9 %.
- **Phred Score From 37**: float number field showing the percentage of reads above a phred score of 37. Accordingly, this value shows the percentage of reads with an accuracy of 99.98 %.
- **Read 0 Count**: integer number field representing the count of reads in the main read direction.
- **Read 0 Average**: float number field representing the average sequence length for the main read direction.
- **Read 0 Stdev**: float number field representing the standard deviation of the sequence length average in the main direction.
- **Read 1 Count**: integer number field representing the count of reads in the reverse read direction. This field will be filled only in paired layout samples as those require both reads.

- **Read 1 Average**: float number field representing the average sequence length for reverse read direction.
- **Read 1 Stdev**: float number field representing the standard deviation of the sequence length average in the reverse read direction.

An example of the CSV file content can be shown in Figure 8.2.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	QUERY	NCBI_STUDY	GSE	SRP	SRR	SPOTS	BASES	ORGANISM	LAYOUT	PHRED_READ_FROM_30	PHRED_READ_FROM_37	READ_0_COUNT	READ_0_AVERAGE	READ_0_STDEV	READ_1_COUNT	READ_1_AVERAGE	READ_1_STDEV
2	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990671	22420894	1143465594	Homo sapiens	SINGLE	0.961298873151753	0.785665390121043	22420894	51	0	0	0	0
3	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990672	19234635	980966385	Homo sapiens	SINGLE	0.959051745692591	0.780645913774099	19234635	51	0	0	0	0
4	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990673	19888776	1014327576	Homo sapiens	SINGLE	0.95834229296355	0.775252386513053	19888776	51	0	0	0	0
5	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990674	17928630	914360130	Homo sapiens	SINGLE	0.963614020440721	0.791543713744386	17928630	51	0	0	0	0
6	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990677	19803760	1009991760	Homo sapiens	SINGLE	0.956787781120115	0.780047121374535	19803760	51	0	0	0	0
7	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990678	17408705	887843955	Homo sapiens	SINGLE	0.961420426633417	0.789526215786422	17408705	51	0	0	0	0
8	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990681	19500164	994508364	Homo sapiens	SINGLE	0.962132414001498	0.786707527378825	19500164	51	0	0	0	0
9	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990684	21763355	1109931105	Homo sapiens	SINGLE	0.960805032128548	0.783611418836667	21763355	51	0	0	0	0
10	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990685	15931010	812481510	Homo sapiens	SINGLE	0.961494413577486	0.787367217747515	15931010	51	0	0	0	0
11	hypothyroidism and rna	200103305	GSE103305	SRP127016	SRR5990686	15024901	766269951	Homo sapiens	SINGLE	0.962405682276324	0.788251474838271	15024901	51	0	0	0	0
12	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664117	6182614	121512279	Macaca mulatta	SINGLE	0.031888151813859 1	0	6182614	19.65	7.67	0	0	0
13	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664118	4269183	101995364	Macaca mulatta	SINGLE	0.031836309736587 6	0	4269183	23.89	9.65	0	0	0
14	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664119	3936229	100486746	Macaca mulatta	SINGLE	0.033478345492449 3	0	3936229	25.53	11.76	0	0	0
15	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664120	4728987	110846638	Macaca mulatta	SINGLE	0.040181732891168 1	0	4728987	23.44	10.19	0	0	0
16	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664121	4096317	95776223	Macaca mulatta	SINGLE	0.031257935489897 1	0	4096317	23.38	9.75	0	0	0
17	hypothyroidism and rna	200118346	GSE118346	SRP156900	SRR7664122	4659155	101308478	Macaca mulatta	SINGLE	0.044323812662549 3	0	4659155	21.74	8.26	0	0	0
18	hypothyroidism and rna	200133110	GSE133110	SRP202034	SRR9332890	41062539	2053126950	Mus musculus	SINGLE	0.961680104096827	0.777724246910304	41062539	50	0	0	0	0
19	hypothyroidism and rna	200133110	GSE133110	SRP202034	SRR9332900	26223837	1850760834	Mus musculus	PAIRED	0.871425919206587	0	26223837	35.08	2.61	26223837	35.49	0.81
20	hypothyroidism and rna	200133110	GSE133110	SRP202034	SRR9332901	42782351	3023454933	Mus musculus	PAIRED	0.940417551446268	0	42782351	35.17	2.37	42782351	35.5	0.78
21	hypothyroidism and rna	200133110	GSE133110	SRP202034	SRR9332914	26286599	1859452715	Mus musculus	PAIRED	0.873350777301159	0	26286599	35.25	2.09	26286599	35.49	0.81
22	hypothyroidism and rna	200133110	GSE133110	SRP202034	SRR9332915	53686549	3801066313	Mus musculus	PAIRED	0.928933155657892	0	53686549	35.3	1.9	53686549	35.5	0.78

Figure 8.2: CSV Content Generate With The Query “hypothyroidism and rna”

9. Next Steps

1. Performance Improvements

The system is capable of extracting the statistics from very big queries, including those that fetch more than one million of studies but, the performance leaves room for improvement as it sometimes takes several hours to complete. On this regard, there are some amendments that can be done without affecting functionality that should speed up the process:

- **Horizontal Scaling** [149]: by increasing the concurrent execution limit of the *Lambda* functions, more processors can work in parallel, thus handling biggest workloads in the same time. Obviously, this should be mirrored by an upgrade on the database engine, that should accept many more connections simultaneously. For the characteristics of this project, that is either idle or processing bursts of traffic, it might be interesting upgrading the database to *Aurora* engine [150]. *Aurora* has the capability of scaling the database resources automatically to match the load. Moreover, the observability platform needs to be upgraded too to cope with the new magnitude of the system. Horizontal scaling improvement has the advantage of being extremely simple to implement by just doing some configuration changes in the infrastructure code. The main drawback is the high monetary costs it implies.
- **Redesign of the flow for secondary GEO entities**: as commented in previous chapters, among the GEO entities, only those of GSE type progress in the data pipeline while the rest (GDS, GPL and GSM) are just stored in database. While storing GDS, GPL or GSM can serve for future development, currently is not giving any value and is delaying the CSV report generation. Having in mind that usually there is a large number of secondary entities that need to be stored in the database, the following options can be considered:
  - In the *Lambda* function that retrieves GEO entities, a sorting can be done with the extracted identifiers, so firstly the GSE entities can be stored and their messages delivered to *SQS*, and later the storing process with secondary GEO entities can happen.
  - As this is probably not enough due to the nature of serverless processors, where several functions are triggered in parallel, another option can be to outsource the storing of those secondary entities to a different fork in the flow. Put in another way, instead of inserting them in the database in the same *Lambda* that retrieves them, this function can send a message to a different *SQS* queue for triggering yet another function that has the single responsibility of storing them in database. This solution can be combined with the sorting previously commented for further effectiveness.
  - Another reinforcing strategy is to create a different database just for those entities. In this case the connection pool of the main database will not get saturated by connections to store secondary entities, which will be saved aside.
  - Of course, a simpler but more drastic approach could be to stop storing these secondary entities until there is no clarity on how to leverage them.

2. Lambda Timeout Limitation

The configuration of some *Lambda* functions for the system [151] has reached its limit of nine hundred seconds timeout [152]. Enlarging the timeout of serverless processors is a clear anti pattern [153] that might indicate a poor design decision as *Lambda* functions fit better for fast and lightweight operations. Interestingly, in *SRA-Collector* workloads, it was observed that for the same operation some invocations perform much faster than others. Consequently, some investigation was carried out on why certain invocations were notoriously slower than others, discovering the following findings:

- The first invocations of some *Lambda* functions for an incoming burst of messages performed noticeable much faster than the subsequent ones. This was due to database connection pool exhaustion, so late invocations need to wait while others are still processing for establishing a successful connection with the

database. Having the database instance configured to be the cheapest *AWS* provides is the key limitation here, as the thread pool admits less that one hundred concurrent connections at most [154].

- Depending on the input, especially in the case of SRPs, the processing time that *Pysradb* library takes to convert it differs greatly. There are some SRPs that have just a bunch of SRRs related while others have thousands of them. This situation just happens in some outliers and the asymmetry it generates can be dealt by the system with acceptable timeouts.

As the database connection limitation was the culprit in the most of the scenarios, the proposed solution to palliate the issue is, again, to upgrade the database engine so the thread pool is bigger and it can assume higher concurrent operations with a decent performance.

3. Mail Server Constraints

Another bottleneck of the system is the method chosen to deliver the results. All commercial mail servers have a limit on the size of the attachments [155] so generating bigger and bigger CSV files as output will eventually reach those maximums.

To solve this issue, how the system outputs the results should be changed. For example, a mail method can be still used but instead of including the file attached, it can include a link to the file in a public *S3* bucket. Of course this solution brings new challenges if the aim is to make each scientist only able to download the file specifically generated by its query and not the rest. For this matter, *Cognito* identities can help as every scientist should acquire one in order to submit queries to *SRA-Collector* and they can be used later to restrict access to S3 objects [156].

4. Expenses Optimization

Cloud expenses have been the main hurdle on the development of the project. Currently every compute engine used is provisioned in its bare minimum. Even with that, on load testing, simulating several input queries in parallel with a number of millions of studies related made the costs spike. To mitigate this problem, especially with regards to the compute instances that support the *PostgreSQL* database and the *OpenSearch* engine, *AWS* offers *Savings Plans* [157]. *Savings Plans* provide more competitive prices for such instances in exchange for some time commitment with the provider, that can span from one to three years. The system clearly has *AWS* vendor lock-in [158] and it is not portable to any other platform as it is deployed, so *Savings Plans* seems a convenient option.

5. Query Prioritization

One feature that, if the system becomes popular will eventually pop up, is query prioritization. Currently, the messages appearing at the queues are consumed without any specific order and such a big query could block the system during its processing for a long time, while smaller queries would stuck pending.

At this moment, the system does count the number of studies related to a query before sending the paginated requests to NCBI servers, so one potential solution for query prioritization would be to use that count to segregate queries among queues. Unfortunately, *AWS* does not provide a native way to prioritize messages in the same queue [159], so the flow should be forked with different queues and processors. An initial approach could be to have two parallel flows coexisting, one for queries bigger than, for example, ten thousand studies and another flow for smaller queries. Also to keep in mind are the shared resources like the database, which would require a mandatory upgrade so the new architecture could have a real impact in performance, otherwise its saturation would prevent any noticeable improvement.

6. User Sign-up Self Service

As commented in Section 2.1, an administrator should create manually an account for any scientist wanting to use the system. This approach can be adequate for the current situation of the system, where there is only a handful of users. But if the interest on using the system raises, manual creation will probably fall short as it is slow and error prone and a self service sign-up method should be settled.

Luckily, *Cognito* user pool already configured in the project supports sign-up self service [160]. To integrate the sign-up capability on the current UI with *Swagger*, another endpoint should be created to register the user, where the scientist would add their mail address and password. *API Gateway* would allow the integration between that request and a *Lambda* in charge of creating the account in *Cognito* and providing feedback. This *Lambda* would use *Cognito* API to validate input parameters, i.e, ensuring that the user does not already exist in the pool and that the password meets the requirements, and then it would emit a response back to the frontend, like a 200 HTTP code for successful user account creation or 409 code for conflicts. Once the user account is created, the scientist can start using the endpoint to submit their queries to the system.

7. *OpenSearch* Credential-Based Authentication

The access to observability dashboards of *OpenSearch* is currently restricted using IP filtering. IP filtering is a good approach for the initial stages of the development where there is no other security mechanisms in place, but it cannot be used permanently as IP assignment tends to be dynamic, with Internet service providers usually changing the addresses of the clients. Notably, the IPs configured to have access to the observability platform [161] have been updated in several occasions during the lifespan of the project.

To prevent this recurrent patching, an enhanced authentication method should be integrated. After some research, the more seamless method could be using the already existing *Cognito* user pool to store and manage identities with access to *OpenSearch* [162]. With the current configuration, where the user pool and the observability platform are already deployed, the process will require only to create some *IAM* policies to allow the administrator identity to access *OpenSearch* domain. *Cognito* can also provide a sign-in default page that can be configured to appear once any user tries to open *OpenSearch* UI. This sign-in frontend would issue the request directly to the *Cognito* API and, if the credentials match, the access would be granted.

8. Custom UI Development

Following the minimum viable product strategy, also known as MVP [163], *Swagger* was used to host and present the UI to the scientists. Although this technology covers the requirement to have a frontend facade in place, it is far from providing a seamless user experience, as it is mainly meant for API documentation instead. For this reason, another improvement would be the design of a custom UI webpage using technologies like *React* or *AngularJS* [164] to replace *Swagger* UI.

With those technologies, a tailored frontend can be created based on open-source components [165] like text fields, buttons, etc which are both easy to integrate and highly customizable. The resulting webpage would bring a more self-explanatory experience to users facilitating interactions. For example, an application regularly used by biomedical scientists is *ProsperousPlus* [166], which has a custom UI pictured in Figure 9.1 with features that can be implemented in *SRA-Collector* website.

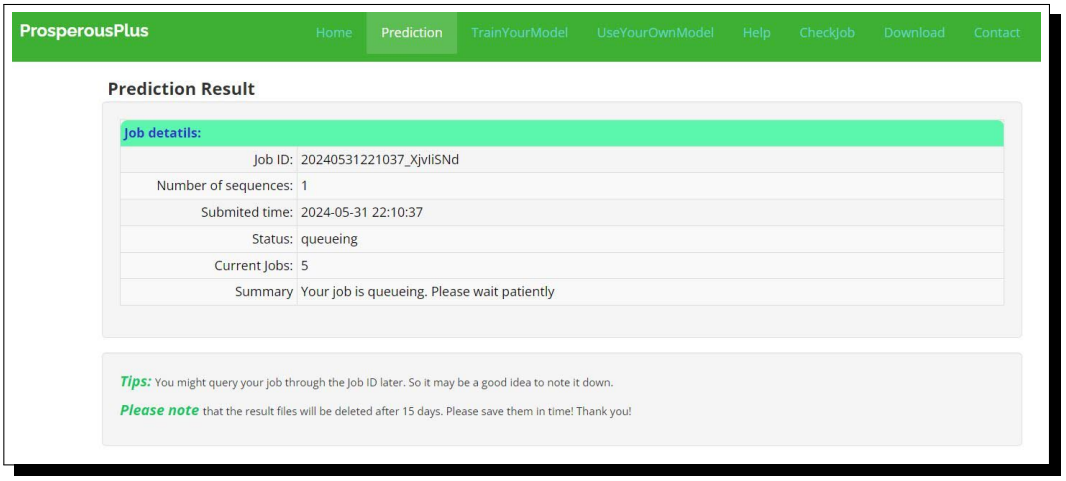

Figure 9.1: ProsperousPlus UI

Appendix A: Domain Specifications

This appendix will detail all domain entities created in the database and their relationships. Needless to say that the domain play a key role on sustaining the processing of the system.

A.1. Entity-Relationship Diagram

The entity-relationship diagram in Figure 9.2, hereafter ERD, shows all the entities of the system and the relationship among them.

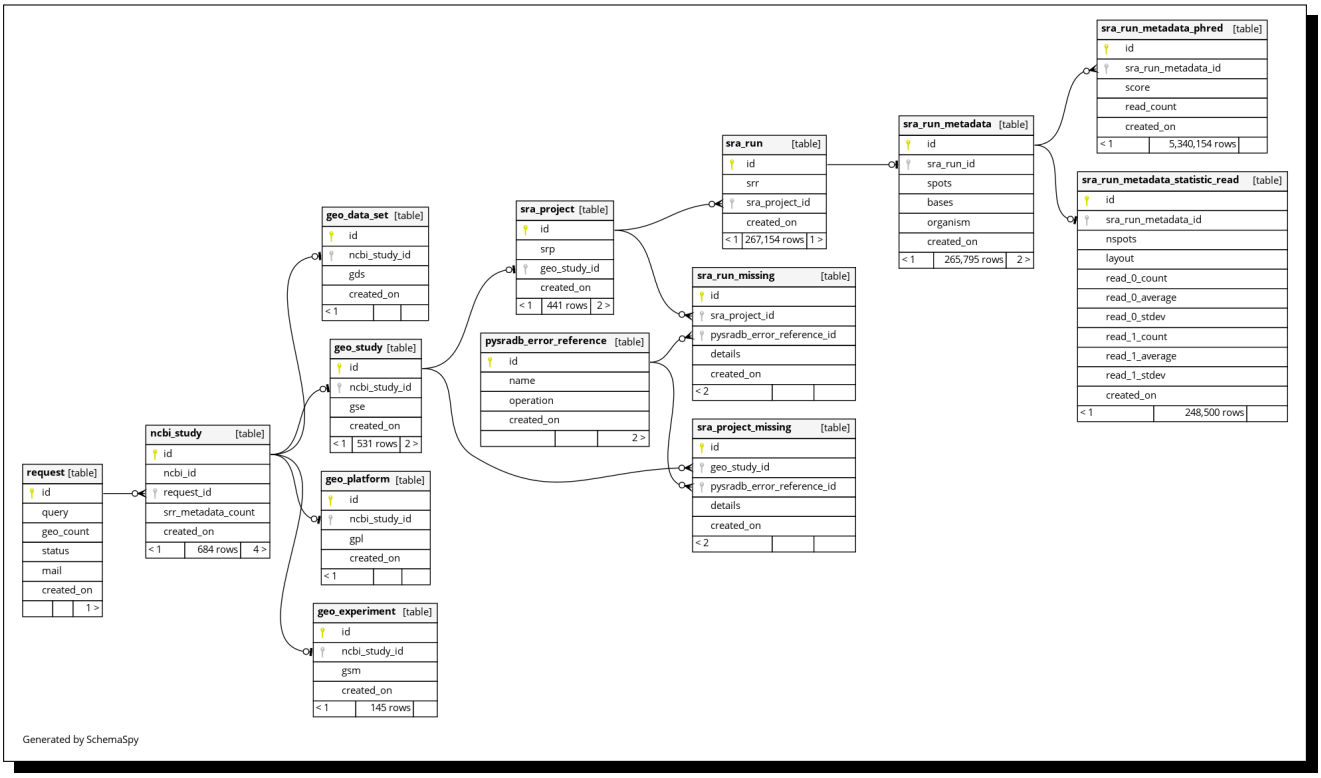

Figure 9.2: Database Diagram Created With SchemaSpy

As shown in the ERD, the connectors that link the entities have symbols that are *"Crow's Foot Notation"* [167], whose meanings are:

- A ring, that appears at the end of all connectors, means "zero".
- A dash, that appears after the ring in some connectors, means "one".
- A crow's foot, that appears after the ring in some connectors, means "many".

So, in this specific diagram, the relationships established are:

- Tables that are linked with the connector ending in a ring and a dash have a one to one optional relationship. An entity can exist in the left table with no related entity in the right table. On the other hand, if a link is established between the left table and the right table, it will be one to one necessarily.
- Tables that are linked with the connector ending in a ring and a crow's foot have a one to many optional relationship. An entity can exist in the left table with no related entities in the right table. On the other hand, the right table can store as many entities related to the one in left table as needed.

A.2. REQUEST Entity

A.2.1. REQUEST Table Structure

Figure 9.3 shows the data definition language (DDL) script that created the table REQUEST.

```
CREATE TABLE REQUEST
(
    ID          VARCHAR UNIQUE,
    QUERY       VARCHAR(500) NOT NULL,
    GEO_COUNT   INTEGER      NOT NULL,
    STATUS      VARCHAR(50)  NOT NULL DEFAULT 'PENDING',
    MAIL        VARCHAR(50)  NOT NULL,
    CREATED_ON TIMESTAMP             DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (ID)
);
```

Figure 9.3: REQUEST Table DDL

REQUEST table column purposes are:

- ID: primary key.
- QUERY: the text query submitted by the scientist.
- GEO_COUNT: count of the NCBI studies retrieved by the query from GEO DataSets database.
- STATUS: either PENDING, EXTRACTED or SENT. It marks specific moments of the flow: PENDING when a query is received and the transformations are being done, EXTRACTED when the metadata CSV report is created and SENT when the mail with the CSV report attached is delivered.
- MAIL: the mail of the requester scientist.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.2.2. REQUEST Table Relationships

- REQUEST table has a one to many optional relationship with NCBI_STUDY table.

A.2.3. REQUEST Table Consumers

- B_get_query_pages: it creates from scratch the REQUEST row for a new input query with REQUEST.STATUS set to PENDING. It checks the existence of the REQUEST row to keep the process idempotent.
- C_get_study_ids: it gets the REQUEST.QUERY for the REQUEST primary key received.
- G_get_srr_metadata: it gets the REQUEST.ID related to a given SRA_RUN primary key. Later, it uses that primary key to compare the expected amount of metadata to extract with the actual count extracted.
- H_generate_report: it checks if REQUEST.STATUS has PENDING value so the data still needs to be collected. If so, it uses a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV. Once done, it updates REQUEST.STATUS to EXTRACTED value.
- I_send_email: it checks if REQUEST.STATUS has EXTRACTED value. If so, it sends the mail and updates the field to SENT.

A.2.4. REQUEST Table Example Rows

In Figure 9.4 some example rows of REQUEST table are shown.

id	query	geo_count	status	mail	created_on
WKKyahuUliAEJtw=	alzheimer and rna seq	408	PENDING	marta.arcones@gmail.com	2024-04-13 10:01:52.104070
WbttkgpNliAEMcw=	tourette syndrome	35	SENT	marta.arcones@gmail.com	2024-04-18 17:47:46.238713
WbuWPgTZliAEJZw=	autism and rna seq	241	SENT	marta.arcones@gmail.com	2024-04-18 17:52:02.240117
XSjB8gvMFiAEPUg=	parkinson and rna seq	49	SENT	marta.arcones@gmail.com	2024-05-05 09:04:41.201282
XTna8iXZliAEMHg=	body lewi dementian and rna seq	11	PENDING	marta.arcones@gmail.com	2024-05-05 16:51:37.411044
XTlTBhAzFiAEPyQ=	multiple sclerosis and rna seq	214	SENT	marta.arcones@gmail.com	2024-05-05 16:37:05.376422

Figure 9.4: `REQUEST` Table Rows

A.3. `NCBI_STUDY` Entity

A.3.1. `NCBI_STUDY` Table Structure

Figure 9.5 shows the DDL operation that created the table `NCBI_STUDY`.

```
CREATE TABLE NCBI_STUDY
(
    ID                 SERIAL PRIMARY KEY,
    NCBI_ID            INTEGER NOT NULL,
    REQUEST_ID         VARCHAR NOT NULL REFERENCES REQUEST (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    SRR_METADATA_COUNT INTEGER   DEFAULT NULL,
    CREATED_ON         TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (REQUEST_ID, NCBI_ID)
);
```

Figure 9.5: `NCBI_STUDY` Table DDL

`NCBI_STUDY` table column purposes are:

- `ID`: primary key.
- `NCBI_ID`: the identifier of the study in NCBI servers.
- `REQUEST_ID`: foreign key to `REQUEST` table.
- `SRR_METADATA_COUNT`: counter for the SRRs the study has related. It is used to control the flow. At the moment of row creation it is null, but later it is updated with the number of expected SRR runs for a given study.
- `CREATED_ON`: automatically filled with the timestamp when the row was created.

A.3.2. `NCBI_STUDY` Table Relationships

- `NCBI_STUDY` table has a one to one optional relationship with `GEO_STUDY` table.
- `NCBI_STUDY` table has a one to one optional relationship with `GEO_DATA_SET` table.
- `NCBI_STUDY` table has a one to one optional relationship with `GEO_PLATFORM` table.
- `NCBI_STUDY` table has a one to one optional relationship with `GEO_EXPERIMENT` table.

A.3.3. `NCBI_STUDY` Table Consumers

- `C_get_study_ids`: it creates from scratch the `NCBI_STUDY` row with the `NCBI_ID` that is extracted from NCBI servers. `SRR_METADATA_COUNT` is kept with the default value which is `NULL`.
- `D_get_study_geo`: it gets the `NCBI_STUDY.NCBI_ID` for the `NCBI_STUDY` primary key received. It updates `NCBI_STUDY.SRR_METADATA_COUNT` to zero in extracted entities different from GSE type.

- E_get_study_srp: it updates NULL rows on NCBI_STUDY.SRR_METADATA_COUNT to zero in any problematic transformation from GSE to SRP.
- F_get_study_srrs: it updates NULL rows on NCBI_STUDY.SRR_METADATA_COUNT to the count of SRRs extracted for a given SRP.
- G_get_srr_metadata: it extracts the NCBI_STUDY.ID with a query joining all parent tables to SRA_RUN of which it has the primary key. Then it compares NCBI_STUDY.SRR_METADATA_COUNT with the number of stored rows in SRA_RUN_METADATA for a given study.
- H_generate_report: NCBI_STUDY is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.3.4. NCBI_STUDY Table Example Rows

In Figure 9.6 some example rows of NCBI_STUDY table are shown.

id	ncbi_id	request_id	srr_metadata_count	created_on
24	200252132	WKKyahuUliAEJtw=	8	2024-04-13 10:01:54.919692
7	200229417	WKKyahuUliAEJtw=	15	2024-04-13 10:01:54.919692
3	200226937	WKKyahuUliAEJtw=	33	2024-04-13 10:01:54.919692
49	200199762	WKKyahuUliAEJtw=	0	2024-04-13 10:01:54.919692
6	200229418	WKKyahuUliAEJtw=	16	2024-04-13 10:01:54.919692
23	200243177	WKKyahuUliAEJtw=	7	2024-04-13 10:01:54.919692
51	200237495	WKKyahuUliAEJtw=	0	2024-04-13 10:01:54.919692
15	200245035	WKKyahuUliAEJtw=	20	2024-04-13 10:01:54.919692
12	200227157	WKKyahuUliAEJtw=	4	2024-04-13 10:01:54.919692
19	200254205	WKKyahuUliAEJtw=	102	2024-04-13 10:01:54.919692
26	200245658	WKKyahuUliAEJtw=	32	2024-04-13 10:01:54.919692
8	200229416	WKKyahuUliAEJtw=	32	2024-04-13 10:01:54.919692
2	200226938	WKKyahuUliAEJtw=	36	2024-04-13 10:01:54.919692

Figure 9.6: NCBI_STUDY Table Rows

A.4. GEO_STUDY Entity

A.4.1. GEO_STUDY Table Structure

Figure 9.7 shows the DDL operation that created the table GEO_STUDY.

```
CREATE TABLE GEO_STUDY
(
    ID            SERIAL PRIMARY KEY,
    NCBI_STUDY_ID INTEGER UNIQUE NOT NULL REFERENCES NCBI_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    GSE           VARCHAR(50)    NOT NULL CHECK (GSE LIKE 'GSE%'),
    CREATED_ON    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (NCBI_STUDY_ID, GSE)
);
```

Figure 9.7: GEO_STUDY Table DDL

GEO_STUDY table column purposes are:

- ID: primary key
- NCBI_STUDY_ID: foreign key to NCBI_STUDY table.

- GSE: GEO identifier retrieved.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.4.2. GEO_STUDY Table Relationships

- GEO_STUDY table has a one to one optional relationship with SRA_PROJECT table.
- GEO_STUDY table has a one to one optional relationship with SRA_PROJECT_MISSING table.

A.4.3. GEO_STUDY Table Consumers

- D_get_study_geo: it creates from scratch the GEO_STUDY row with the identifier extracted from NCBI servers.
- E_get_study_srp: it retrieves the GEO_STUDY.GSE using the GEO_STUDY.ID received in the message. GEO_STUDY.ID is also used to find the corresponding entity on NCBI_STUDY table.
- F_get_study_srrs: the table is traversed in a join to find the NCBI_STUDY entity using SRA_PROJECT primary key.
- G_get_srr_metadata: the table is traversed in a join to find the REQUEST entity using SRA_RUN primary key.
- H_generate_report: GEO_STUDY is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.4.4. GEO_STUDY Table Example Rows

In Figure 9.8 some example rows of GEO_STUDY table are shown.

id	ncbi_study_id	gse	created_on
1	3	GSE226937	2024-04-13 10:02:09.170554
2	24	GSE252132	2024-04-13 10:02:09.230343
3	9	GSE207821	2024-04-13 10:02:09.227830
4	23	GSE243177	2024-04-13 10:02:09.241731

Figure 9.8: GEO_STUDY Table Rows

A.5. GEO_EXPERIMENT Entity

A.5.1. GEO_EXPERIMENT Table Structure

Figure 9.9 shows the DDL operation that created the table GEO_EXPERIMENT.

```
CREATE TABLE GEO_EXPERIMENT
(
    ID            SERIAL PRIMARY KEY,
    NCBI_STUDY_ID INTEGER UNIQUE NOT NULL REFERENCES NCBI_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    GSM           VARCHAR(50)    NOT NULL CHECK (GSM LIKE 'GSM%'),
    CREATED_ON    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (NCBI_STUDY_ID, GSM)
);
```

Figure 9.9: GEO_EXPERIMENT Table DDL

GEO_EXPERIMENT table column purposes are:

- ID: primary key.

- NCBI_STUDY_ID: foreign key to NCBI_STUDY table.
- GSM: GEO identifier retrieved.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.5.2. GEO_EXPERIMENT Table Relationships

Apart from the relationship with NCBI_STUDY already commented, GEO_EXPERIMENT has no further relationships.

A.5.3. GEO_EXPERIMENT Table Consumers

- D_get_study_geo: it creates from scratch the GEO_EXPERIMENT row with the identifier extracted from NCBI servers.

A.5.4. GEO_EXPERIMENT Table Example Rows

In Figure 9.10 some example rows of GEO_EXPERIMENT table are shown.

id	ncbi_study_id	gsm	created_on
1	397	GSM3732962	2024-04-13 10:02:11.396073
2	398	GSM3732959	2024-04-13 10:02:13.239774
3	333	GSM7040879	2024-04-13 10:02:14.058608
4	334	GSM7040878	2024-04-13 10:02:14.081739
5	347	GSM6782961	2024-04-13 10:02:14.101617
6	348	GSM6783598	2024-04-13 10:02:14.122043

Figure 9.10: GEO_EXPERIMENT Table Rows

A.6. GEO_DATA_SET Entity

A.6.1. GEO_DATA_SET Table Structure

Figure 9.11 shows the DDL operation that created the table GEO_DATA_SET.

```
CREATE TABLE GEO_DATA_SET
(
    ID            SERIAL PRIMARY KEY,
    NCBI_STUDY_ID INTEGER UNIQUE NOT NULL REFERENCES NCBI_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    GDS           VARCHAR(50)    NOT NULL CHECK (GDS LIKE 'GDS%'),
    CREATED_ON    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (NCBI_STUDY_ID, GDS)
);
```

Figure 9.11: GEO_DATA_SET Table DDL

GEO_DATA_SET table column purposes are:

- ID: primary key.
- NCBI_STUDY_ID: foreign key to NCBI_STUDY table.
- GDS: GEO identifier retrieved.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.6.2. GEO_DATA_SET Table Relationships

Apart from the relationship with NCBI_STUDY already commented, GEO_DATA_SET has no further relationships.

A.6.3. GEO_DATA_SET Table Consumers

- D_get_study_geo: it creates from scratch the GEO_DATA_SET row with the identifier extracted from NCBI servers.

A.6.4. GEO_DATA_SET Table Example Rows

In Figure 9.12 some example rows of GEO_DATA_SET table are shown.

id	ncbi_study_id	gds	created_on
1	1017	GDS4484	2024-05-31 12:59:31.544843
2	1026	GDS3698	2024-05-31 12:59:31.803112
3	1036	GDS469	2024-05-31 12:59:32.063115
4	1027	GDS3697	2024-05-31 12:59:32.193872

Figure 9.12: GEO_DATA_SET Table Rows

A.7. GEO_PLATFORM Entity

A.7.1. GEO_PLATFORM Table Structure

Figure 9.13 shows the DDL operation that created the table GEO_PLATFORM.

```
CREATE TABLE GEO_PLATFORM
(
    ID            SERIAL PRIMARY KEY,
    NCBI_STUDY_ID INTEGER UNIQUE NOT NULL REFERENCES NCBI_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    GPL           VARCHAR(50)    NOT NULL CHECK (GPL LIKE 'GPL%'),
    CREATED_ON    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (NCBI_STUDY_ID, GPL)
);
```

Figure 9.13: GEO_PLATFORM Table DDL

GEO_PLATFORM table column purposes are:

- ID: primary key.
- NCBI_STUDY_ID: foreign key to NCBI_STUDY table.
- GPL: GEO identifier retrieved.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.7.2. GEO_PLATFORM Table Relationships

Apart from the relationship with NCBI_STUDY already commented, GEO_PLATFORM has no further relationships.

A.7.3. GEO_PLATFORM Table Consumers

- D_get_study_geo: it creates from scratch the GEO_PLATFORM row with the identifier extracted from NCBI servers.

A.7.4. GEO_PLATFORM Table Example Rows

In Figure 9.14 some example rows of GEO_PLATFORM table are shown.

id	ncbi_study_id	gpl	created_on
4	1326	GPL19750	2024-06-02 09:51:13.613916
5	1237	GPL19751	2024-06-02 09:51:13.613916
6	1328	GPL24889	2024-06-02 09:51:13.613916

Figure 9.14: GEO_PLATFORM Table Rows

A.8. SRA_PROJECT Entity

A.8.1. SRA_PROJECT Table Structure

Figure 9.15 shows the DDL operation that created the table SRA_PROJECT.

```
CREATE TABLE SRA_PROJECT
(
    ID           SERIAL PRIMARY KEY,
    SRP          VARCHAR(50)    NOT NULL CHECK (SRP LIKE 'SRP%'),
    GEO_STUDY_ID INTEGER UNIQUE NOT NULL REFERENCES GEO_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    CREATED_ON   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (GEO_STUDY_ID, SRP)
);
```

Figure 9.15: SRA_PROJECT Table DDL

SRA_PROJECT table column purposes are:

- ID: primary key.
- SRP: the SRP identifier successfully extracted using *Pysradb*.
- GEO_STUDY_ID: foreign key to GEO_STUDY table.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.8.2. SRA_PROJECT Table Relationships

- SRA_PROJECT table has a one to many optional relationship with SRA_RUN table.
- SRA_PROJECT table has a one to many optional relationship with SRA_RUN_MISSING table.

A.8.3. SRA_PROJECT Table Consumers

- E_get_study_srp: it creates from scratch the SRA_PROJECT row with the SRP that *Pysradb* extracts.
- F_get_study_srrs: it uses the primary key from SRA_PROJECT received in the incoming message to find the SRP. Same primary key is also used to find the corresponding NCBI_STUDY.
- G_get_srr_metadata: the table is traversed in a join to find the REQUEST entity using SRA_RUN primary key.
- H_generate_report: SRA_PROJECT is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.8.4. SRA_PROJECT Table Example Rows

In Figure 9.16 some example rows of SRA_PROJECT table are shown.

id	srp	geo_study_id	created_on
1	SRP432025	6	2024-04-13 10:02:13.560005
2	SRP480332	2	2024-04-13 10:02:13.601383
3	SRP385693	3	2024-04-13 10:02:13.607446
4	SRP426297	1	2024-04-13 10:02:13.614336
5	SRP485933	5	2024-04-13 10:02:13.526716
6	SRP432040	12	2024-04-13 10:02:17.011605

Figure 9.16: SRA_PROJECT Table Rows

A.9. PYSRADB_ERROR_REFERENCE Entity

A.9.1. PYSRADB_ERROR_REFERENCE Table Structure

Figure 9.17 shows the DDL operation that created the table PYSRADB_ERROR_REFERENCE.

```
CREATE TABLE PYSRADB_ERROR_REFERENCE
(
    ID          SERIAL PRIMARY KEY,
    NAME        VARCHAR(50) NOT NULL,
    OPERATION   VARCHAR(50) NOT NULL,
    CREATED_ON TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO PYSRADB_ERROR_REFERENCE (OPERATION, NAME)
VALUES ('gse_to_srp', 'ATTRIBUTE_ERROR'),
       ('gse_to_srp', 'VALUE_ERROR'),
       ('gse_to_srp', 'KEY_ERROR'),
       ('gse_to_srp', 'NOT_FOUND'),

       ('srp_to_srr', 'ATTRIBUTE_ERROR'),
       ('srp_to_srr', 'NOT_FOUND'),
       ('srp_to_srr', 'TYPE_ERROR')
;
```

Figure 9.17: PYSRADB_ERROR_REFERENCE Table DDL

PYSRADB_ERROR_REFERENCE table column purposes are:

- ID: primary key.
- NAME: the exception name that *Pysradb* raises.
- OPERATION: in which operation *Pysradb* failed. The value of this column can be gse_to_srp or srp_to_srr only.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.9.2. PYSRADB_ERROR_REFERENCE Table Relationships

- PYSRADB_ERROR_REFERENCE table has a one to many optional relationship with SRA_PROJECT_MISSING table.
- PYSRADB_ERROR_REFERENCE table has a one to many optional relationship with SRA_RUN_MISSING table.

A.9.3. PYSRADB_ERROR_REFERENCE Table Consumers

- E_get_study_srp: it searches in PYSRADB_ERROR_REFERENCE the ID of the error appeared on gse_to_srp conversion.
- F_get_study_srrs: it searches in PYSRADB_ERROR_REFERENCE the ID of the error appeared on srp_to_srr conversion.

A.9.4. PYSRADB_ERROR_REFERENCE Table Example Rows

Only rows inserted on DDL script are present in PYSRADB_ERROR_REFERENCE.

A.10. SRA_RUN Entity

A.10.1. SRA_RUN Table Structure

Figure 9.18 shows the DDL operation that created the table SRA_RUN.

```
CREATE TABLE SRA_RUN
(
    ID             SERIAL PRIMARY KEY,
    SRR            VARCHAR(50) NOT NULL CHECK (SRR LIKE 'SRR%'),
    SRA_PROJECT_ID INTEGER     NOT NULL REFERENCES SRA_PROJECT (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    CREATED_ON     TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (SRA_PROJECT_ID, SRR)
);
```

Figure 9.18: SRA_RUN Table DDL

SRA_RUN table column purposes are:

- ID: primary key.
- SRR: the SRR identifier successfully extracted using *Pysradb*.
- SRA_STUDY_ID: foreign key to SRA_PROJECT table.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.10.2. SRA_RUN Table Relationships

- SRA_RUN table has a one to one optional relationship with SRA_RUN_METADATA table.

A.10.3. SRA_RUN Table Consumers

- F_get_study_srrs: it creates from scratch the SRA_RUN row with the SRR that *Pysradb* extracts.
- G_get_srr_metadata: the primary key of the table arrives in the incoming message and is used to get the SRA_RUN.SRR and the corresponding entities in REQUEST and NCBI_STUDY tables.
- H_generate_report: SRA_RUN is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.10.4. SRA_RUN Table Example Rows

In Figure 9.19 some example rows of SRA_RUN table are shown.

id	srr	sra_project_id	created_on
1	SRR27368437	2	2024-04-13 10:02:18.581223
2	SRR27368438	2	2024-04-13 10:02:18.581223
3	SRR27368439	2	2024-04-13 10:02:18.581223
4	SRR27368440	2	2024-04-13 10:02:18.581223
5	SRR27368441	2	2024-04-13 10:02:18.581223
6	SRR27368442	2	2024-04-13 10:02:18.581223

Figure 9.19: SRA_RUN Table Rows

A.11. SRA_PROJECT_MISSING Entity

A.11.1. SRA_PROJECT_MISSING Table Structure

Figure 9.20 shows the DDL operation that created the table SRA_PROJECT_MISSING.

```
CREATE TABLE SRA_PROJECT_MISSING
(
    ID                          SERIAL PRIMARY KEY,
    GEO_STUDY_ID                INTEGER      NOT NULL REFERENCES GEO_STUDY (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    PYSRADB_ERROR_REFERENCE_ID INTEGER      NOT NULL REFERENCES PYSRADB_ERROR_REFERENCE (ID),
    DETAILS                     VARCHAR(500) NOT NULL,
    CREATED_ON                  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (ID, GEO_STUDY_ID)
);
```

Figure 9.20: SRA_PROJECT_MISSING Table DDL

SRA_PROJECT_MISSING table column purposes are:

- ID: primary key.
- GEO_STUDY_ID: foreign key to GEO_STUDY table.
- PYSRADB_ERROR_REFERENCE_ID: foreign key to PYSRADB_ERROR_REFERENCE table.
- DETAILS: error log that *Pysradb* outputs.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.11.2. SRA_PROJECT_MISSING Table Relationships

Apart from the relationship with GEO_STUDY and PYSRADB_ERROR_REFERENCE already commented, SRA_PROJECT_MISSING has no further relationships.

A.11.3. SRA_PROJECT_MISSING Table Consumers

- E_get_study_srp: it creates from scratch the SRA_PROJECT_MISSING row with the GEO_STUDY.ID that makes *Pysradb* to fail in gse_to_srp operation.

A.11.4. SRA_PROJECT_MISSING Table Example Rows

In Figure 9.21 some example rows of SRA_PROJECT_MISSING table are shown.

id	geo_study_id	pysradb_error_reference_id	details	created_on
1	22	1	'NoneType' object has no attribute 'ren…	2024-04-13 10:02:20.219601
2	23	1	'NoneType' object has no attribute 'ren…	2024-04-13 10:02:22.462373
3	36	2	All arrays must be of the same length	2024-04-13 10:02:22.465380
4	50	3	'Summary'	2024-04-13 10:02:25.553937
5	61	2	All arrays must be of the same length	2024-04-13 10:02:27.542384
6	66	3	'Summary'	2024-04-13 10:02:28.027369

Figure 9.21: SRA_PROJECT_MISSING Table Rows

A.12. SRA_RUN_MISSING Entity

A.12.1. SRA_RUN_MISSING Table Structure

Figure 9.22 shows the DDL operation that created the table SRA_RUN_MISSING.

```
CREATE TABLE SRA_RUN_MISSING
(
    ID                          SERIAL PRIMARY KEY,
    SRA_PROJECT_ID              INTEGER      NOT NULL REFERENCES SRA_PROJECT (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    PYSRADB_ERROR_REFERENCE_ID INTEGER      NOT NULL REFERENCES PYSRADB_ERROR_REFERENCE (ID),
    DETAILS                     VARCHAR(500) NOT NULL,
    CREATED_ON                  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (ID, SRA_PROJECT_ID)
);
```

Figure 9.22: SRA_RUN_MISSING Table DDL

SRA_RUN_MISSING table column purposes are:

- ID: primary key.
- SRA_PROJECT_ID: foreign key to SRA_PROJECT table.
- PYSRADB_ERROR_REFERENCE_ID: foreign key to PYSRADB_ERROR_REFERENCE table.
- DETAILS: error log that *Pysradb* outputs.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.12.2. SRA_RUN_MISSING Table Relationships

Apart from the relationship with SRA_PROJECT and PYSRADB_ERROR_REFERENCE already commented, SRA_RUN_MISSING has no further relationships.

A.12.3. SRA_RUN_MISSING Table Consumers

- F_get_study_srrs: it creates from scratch the SRA_RUN_MISSING row with the SRA_PROJECT.ID that makes *Pysradb* to fail in srp_to_srr operation.

A.12.4. SRA_RUN_MISSING Table Example Rows

In Figure 9.23 some example rows of SRA_RUN_MISSING table are shown.

id	sra_project_id	pysradb_error_reference_id	details	created_on
1	197	5	'NoneType' object has no attribute 'columns'	2024-04-13 10:02:59.793361
2	164	5	'NAType' object has no attribute 'startswith'	2024-04-13 10:02:52.240628
3	334	5	'NoneType' object has no attribute 'columns'	2024-04-18 17:52:35.818623

Figure 9.23: SRA_RUN_MISSING Table Rows

A.13. SRA_RUN_METADATA Entity

A.13.1. SRA_RUN_METADATA Table Structure

Figure 9.24 shows the DDL operation that created the table SRA_RUN_METADATA.

```
CREATE TABLE SRA_RUN_METADATA
(
    ID          SERIAL PRIMARY KEY,
    SRA_RUN_ID INTEGER UNIQUE NOT NULL REFERENCES SRA_RUN (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    SPOTS       BIGINT          NOT NULL,
    BASES       BIGINT          NOT NULL,
    ORGANISM    VARCHAR(500)    NOT NULL,
    CREATED_ON TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (SRA_RUN_ID, SPOTS, BASES, ORGANISM)
);
```

Figure 9.24: SRA_RUN_METADATA Table DDL

SRA_RUN_METADATA table column purposes are:

- ID: primary key.
- SRA_RUN_ID: foreign key to SRA_RUN table.
- SPOTS: how many spots the SRA_RUN has.
- BASES: how many bases the SRA_RUN has.
- ORGANISM: from which organism the SRA_RUN was obtained.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.13.2. SRA_RUN_METADATA Table Relationships

- SRA_RUN_METADATA table has a one to many optional relationship with SRA_RUN_METADATA_PHRED table.
- SRA_RUN_METADATA table has a one to one optional relationship with SRA_RUN_METADATA_STATISTIC_READ table.

A.13.3. SRA_RUN_METADATA Table Consumers

- G_get_srr_metadata: it creates from scratch the SRA_RUN_METADATA rows with the main statistics extracted from NCBI *Traces* service. It checks if the metadata extraction process is done getting the count of rows in SRA_RUN_METADATA related to the same REQUEST entity.
- H_generate_report: SRA_RUN_METADATA is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.13.4. SRA_RUN_METADATA Table Example Rows

In Figure 9.25 some example rows of SRA_RUN_METADATA table are shown.

id	sra_run_id	spots	bases	organism	created_on
1	5	34017166	10273184132	Homo sapiens	2024-04-13 10:02:23.901969
2	3	42800063	12925619026	Homo sapiens	2024-04-13 10:02:23.937788
3	2	32420217	9790905534	Homo sapiens	2024-04-13 10:02:24.116230
4	4	35419937	10696820974	Homo sapiens	2024-04-13 10:02:24.027115
5	1	39337155	11879820810	Homo sapiens	2024-04-13 10:02:24.076081
6	13	24207920	1815594000	Mus musculus	2024-04-13 10:02:24.977753
7	15	21724736	1629355200	Mus musculus	2024-04-13 10:02:25.075868
8	7	35192467	10628125034	Homo sapiens	2024-04-13 10:02:25.136102
9	30	27197396	2284581264	Homo sapiens	2024-04-13 10:02:25.267120
10	14	23734660	1780099500	Mus musculus	2024-04-13 10:02:25.847113
11	8	36913942	11148010484	Homo sapiens	2024-04-13 10:02:25.876127
12	31	11879277	997859268	Homo sapiens	2024-04-13 10:02:26.013501

Figure 9.25: SRA_RUN_METADATA Table Rows

A.14. SRA_RUN_METADATA_PHRED Entity

A.14.1. SRA_RUN_METADATA_PHRED Table Structure

Figure 9.26 shows the DDL operation that created the table SRA_RUN_METADATA_PHRED.

```
CREATE TABLE SRA_RUN_METADATA_PHRED
(
    ID                  SERIAL PRIMARY KEY,
    SRA_RUN_METADATA_ID INTEGER NOT NULL REFERENCES SRA_RUN_METADATA (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    SCORE               INT,
    READ_COUNT          BIGINT,
    CREATED_ON          TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 9.26: SRA_RUN_METADATA_PHRED Table DDL

SRA_RUN_METADATA_PHRED table column purposes are:

- ID: primary key.
- SRA_RUN_METADATA_ID: foreign key to SRA_RUN_METADATA table.
- SCORE: quality measurement of the SRR.
- READ_COUNT: how many reads the SRA_RUN has.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.14.2. SRA_RUN_METADATA_PHRED Table Relationships

Apart from the relationship with SRA_RUN_METADATA already commented, SRA_RUN_METADATA_PHRED has no further relationships.

A.14.3. SRA_RUN_METADATA_PHRED Table Consumers

- G_get_srr_metadata: it creates from scratch the SRA_RUN_METADATA_PHRED rows with the phred statistics extracted from NCBI *Traces* service.
- H_generate_report: SRA_RUN_METADATA_PHRED is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.14.4. SRA_RUN_METADATA_PHRED Table Example Rows

In Figure 9.27 some example rows of SRA_RUN_METADATA_PHRED table are shown.

id	sra_run_metadata_id	score	read_count	created_on
1	1	2	95414	2024-04-13 10:02:24.666899
2	1	11	444606490	2024-04-13 10:02:24.666899
3	1	25	603136659	2024-04-13 10:02:24.666899
4	1	37	9225345569	2024-04-13 10:02:24.666899
5	2	2	121312	2024-04-13 10:02:24.775429
6	2	11	536115173	2024-04-13 10:02:24.775429
7	2	25	744841034	2024-04-13 10:02:24.775429

Figure 9.27: SRA_RUN_METADATA_PHRED Table Rows

A.15. SRA_RUN_METADATA_STATISTIC_READ Entity

A.15.1. SRA_RUN_METADATA_STATISTIC_READ Table Structure

Figure 9.28 shows the DDL operation that created the table SRA_RUN_METADATA_STATISTIC_READ.

```
CREATE TABLE SRA_RUN_METADATA_STATISTIC_READ
(
    ID                  SERIAL PRIMARY KEY,
    SRA_RUN_METADATA_ID INTEGER UNIQUE NOT NULL REFERENCES SRA_RUN_METADATA (ID) ON UPDATE CASCADE ON DELETE CASCADE,
    NSPOTS              BIGINT,
    LAYOUT              VARCHAR(50) CHECK (LAYOUT IN ('PAIRED', 'SINGLE')),
    READ_0_COUNT        BIGINT,
    READ_0_AVERAGE      FLOAT,
    READ_0_STDEV        FLOAT,
    READ_1_COUNT        BIGINT,
    READ_1_AVERAGE      FLOAT,
    READ_1_STDEV        FLOAT,
    CREATED_ON          TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (SRA_RUN_METADATA_ID, NSPOTS, LAYOUT, READ_0_COUNT, READ_0_AVERAGE, READ_0_STDEV, READ_1_COUNT, READ_1_AVERAGE, READ_1_STDEV)
);
```

Figure 9.28: SRA_RUN_METADATA_STATISTIC_READ Table DDL

SRA_RUN_METADATA_STATISTIC_READ table column purposes are:

- ID: primary key.
- SRA_RUN_METADATA_ID: foreign key to SRA_RUN_METADATA table.
- NSPOTS: how many spots the SRA_RUN has.
- LAYOUT: in which direction the read was done, can be either SINGLE for one direction or PAIRED for both directions.
- READ_0_COUNT: number of reads in the main direction.
- READ_0_AVERAGE: average read in the main direction.
- READ_0_STDEV: standard deviation of reads in the main direction.
- READ_1_COUNT: number of reads in the reverse direction.
- READ_1_AVERAGE: average read in the reverse direction.
- READ_1_STDEV: standard deviation of reads in the reverse direction.
- CREATED_ON: automatically filled with the timestamp when the row was created.

A.15.2. SRA_RUN_METADATA_STATISTIC_READ Table Relationships

Apart from the relationship with SRA_RUN_METADATA already commented, SRA_RUN_METADATA_STATISTIC_READ has no further relationships.

A.15.3. SRA_RUN_METADATA_STATISTIC_READ Table Consumers

- G_get_srr_metadata: it creates from scratch the SRA_RUN_METADATA_STATISTIC_READ rows with the read statistics extracted from NCBI *Traces* service.
- H_generate_report: SRA_RUN_METADATA_STATISTIC_READ is used in a query joining all tables in the system using the REQUEST.ID to extract the data that will be shaped as CSV.

A.15.4. SRA_RUN_METADATA_STATISTIC_READ Table Example Rows

In Figure 9.29 can be shown some example rows of SRA_RUN_METADATA_STATISTIC_READ table.

id	sra_run_metadata_id	nspots	layout	read_0_count	read_0_average	read_0_stdev	read_1_count	read_1_average	read_1_stdev	created_on
1	1	34017166	PAIRED	34017166	151	0	34017166	151	0	2024-04-13 10:02:24.685342
2	2	42800063	PAIRED	42800063	151	0	42800063	151	0	2024-04-13 10:02:24.782901
3	3	32420217	PAIRED	32420217	151	0	32420217	151	0	2024-04-13 10:02:24.859615
4	4	35419937	PAIRED	35419937	151	0	35419937	151	0	2024-04-13 10:02:24.870446
5	5	39337155	PAIRED	39337155	151	0	39337155	151	0	2024-04-13 10:02:24.916674
6	6	24207920	SINGLE	24207920	75	0	0	0	0	2024-04-13 10:02:25.166499
7	7	21724736	SINGLE	21724736	75	0	0	0	0	2024-04-13 10:02:25.277706

Figure 9.29: SRA_RUN_METADATA_STATISTIC_READ Table Rows

Appendix B: Monetary Budget

Given that *SRA-Collector* is a living project hosted in the cloud, its expenses vary depending on the operational period. Thus, the budget outlined above is on monthly basis, aligning with the typical billing cycle of cloud providers, such as *AWS*. It is also key to consider the elasticity of the system, as it can automatically adjust its processing power to manage sudden spikes of incoming traffic. This dynamic nature results in fluctuating cloud provider costs, making the calculated budget an estimate based on an expected usage of 500 queries per month, each comprising an average of 350 studies, in a 30-day period.

To calculate these costs, *Cloudcraft* online service [168] was used, as it offers insights into the expenses of individual cloud services based on the inputs provided. On the other hand, apart from cloud provider invoice, there is a separate budget allocation covering the costs associated with the administrator, responsible for regular system updates and patches, ensuring system's smooth operation and security.

Type	Item	Measure	Amount	Rate	Taxes	Total
Administrator Monthly Expenses	Administrator Monthly Expenses	Hours	8	90.00 €	0.21	871.20 €
Cloud Monthly Expenses	8 * Lambda Service (128MiB & 53s/request average)	Requests	823600	0.000820 €	0.21	817.18 €
Cloud Monthly Expenses	1 * Lambda Service (256MiB & 2s/request average)	Requests	36500	0.000140 €	0.21	6.18 €
Cloud Monthly Expenses	Lambda Service (1024MiB & 5s/request average)	Requests	13500	0.000080 €	0.21	1.31 €
Cloud Monthly Expenses	RDS Database Service	Hours	720	0.018060 €	0.21	15.73 €
Cloud Monthly Expenses	S3 Storage Service	Storage GiB	100	0.045000 €	0.21	5.45 €
Cloud Monthly Expenses	API Gateway Networking Service	Requests	500	0.000000 €	0.21	0.00 €
Cloud Monthly Expenses	OpenSearch Analytics Service	Hours	720	0.040000 €	0.21	34.85 €
Cloud Monthly Expenses	SQS Integration Service	Requests	1092000	0.000006 €	0.21	7.93 €
Total Monthly Expenses Taxes						1,759.82 €

Table 9.1: Monetary Monthly Budget

Appendix C: Ethical, Economical, Social, and Environmental Facets

C.16. Introduction

This final degree project addresses the requirements set forth by the EUR-ACE and ABET international accreditation bodies [169, 170]. The project not only focuses on the practical application of engineering principles but also tackles some ethical, environmental, social and economical facets. By embracing this holistic approach, the project aims to meet the criteria of the aforementioned accreditations and to contribute meaningfully to the field of engineering and bioinformatics.

C.17. Relevant Aspects

In the landscape of bioinformatics, advancements occur at a rapid pace, with new tools continuously becoming available, some those relatively novel. The variety of techniques and procedures inherent to biomedical research coupled with the enormous datasets utilized, emphasizes the need for streamlined and automated protocols. A simple online search reveals the proliferation of bespoke tools, often developed by the community, showing the speed of innovation in the field.

With this background, the implementation of *SRA-Collector* system represents a step forward to enhance and accelerate scientific research. As its development progressed, the potential for transformative impact became increasingly evident encouraging determined attention to social, ethical, environmental and economical challenges. Consequently, the project fine-grained goals have been carefully aligned with broader societal objectives, such as those outlined in the 2030 agenda [171].

C.18. Impact Detail

C.18.1. Social Facet

The nature of this project, aimed at accelerating biomedical scientists' investigations, presents considerable potential for societal advancement. As this application speeds up some steps of the biomedical research dealing with sequencing studies, it may finally contribute to the discovery of more effective drugs, ultimately contributing to better health outcomes and increased longevity. Moreover, by enabling researchers to process larger volumes of data in shorter periods, this initiative can also foster an increase in studies focused on preventive measures tailored to patients with genetic predispositions. This strategic shift towards preventive medicine holds the promise of not only mitigating disease progression but also preemptively addressing potential health risks, ultimately leading to healthier populations and reduced healthcare costs.

C.18.2. Ethical Facet

Biomedical research has traditionally relied on animal models to conduct experiments [172]. However, ethical concerns have been always significant, as the animals involved are sometimes subjected to aggressive procedures and face premature death. Additionally, both researchers and all the assistant staff involved in animal experimentation frequently experience mental health issues due to the distressing nature of their work. In view of the advancements in highly reliable computational models, the perspective within biomedical research has increasingly shifted towards minimizing animal experimentation whenever feasible. This shift leverages the ability of highly-available simulation frameworks and artificial intelligence technologies [4, 173].

SRA-Collector system aligns with this trend by enabling scientists to find and process vast datasets from existing studies, categorized by organism, such as mice or rats, and other pertinent characteristics. As a result, an efficient application usage allows to easily find data produced by other researchers, thus significantly reducing the necessity to conduct redundant experiments that would inexorably require animal testing. This not only addresses ethical concerns but also prevent the mental toll suffered by the staff in scientific

institutions. This capability to explore new investigation lines and implement innovative solutions with reduced ethical implications contributes to a more humane and effective approach to biomedical research.

C.18.3. Environmental Facet

Limiting animal experimentation not only reduces animal suffering but also has positive environmental implications. By minimizing the need for animal studies, the emissions associated with animal breeding and experimentation facilities are decreased. These facilities often consume significant amounts of resources, including water, food, animal bedding and electricity, and produce animal waste, contributing to environmental pollution and resource exhaustion. By reducing the demands for such facilities, the environmental burden associated with their operation is mitigated, leading to a reduction in overall environmental impact.

Moreover, by enabling studies to proceed with fewer hours of work, thanks to the automations brought by *SRA-Collector* system, the energy consumption and associated footprint of research activities are reduced too. Traditionally, research experiments involving animals require extensive time and effort from researchers and supporting staff, leading to increased energy consumption and greenhouse gas emissions. However, by streamlining data collection and analysis processes, *SRA-Collector* minimizes the time and means required for research, thereby reducing its environmental impact.

By prioritizing efficiency and minimizing resource consumption, researchers can conduct cutting-edge research while minimizing their environmental footprint, fostering a more environmentally conscious scientific community.

C.18.4. Economical Facet

Reducing the number of animals used in research not only fosters a more compassionate and environmentally friendly approach but also leads to significant cost savings for research institutions. By decreasing the number of animals required for experiments, researchers can minimize expenses associated with animal breeding, care, and housing. Additionally, mitigating the healthcare costs related to the mental health toll on personnel involved in animal research is another indirect benefit.

Moreover, with fewer hours dedicated to animal-related tasks, which are often error-prone and time-consuming, personnel can redirect their time and skills to other areas of research, potentially enhancing productivity and efficiency across organizations.

Overall, the combination of fewer animals and reduced labor hours results in substantial cost savings for research institutions, promoting greater financial sustainability in the long run. This is particularly advantageous for research groups with limited budgets that often struggle to secure financing options. Consequently, these cost reductions would facilitate further advancements in scientific knowledge and innovation in the field of biomedicine.

C.19. Summary

In this appendix, a comprehensive exploration of the social, ethical, environmental and economical dimensions inherent to the implementation of *SRA-Collector* system have been presented. This aligns seamlessly with the modern and innovative engineering approach highlighted in the preceding chapters of this essay. While the pursuit of ABET and EUR-ACE accreditations is significant, the impact detailed in the facets is even more important. The multifaceted approach and forward-thinking methodologies employed reinforce the potential for obtaining these accreditations, but their true significance lies in the positive impact they can have on the society, the economy, ethics and the environment.

Glossary

API *Application Programming Interface, a way to communicate two or more computer systems*

AWS *Amazon Web Services, on demand cloud computing platform*

CORS *Cross-Origin Resource Sharing, mechanism to allow a web page to access restricted resources from a server on a domain different that the domain that served the page*

CI/CD *Combined practices of continuous integration and continuous delivery/deployment. It comprises different techniques as frequent merges of small changes in main branch, automatized deployment processes, etc*

CLI *Command-Line Interface, way of interacting with an application using text input*

CSV *Text file format that uses commas to separate values and newlines to separate records*

DB *DataBase, organized collection of data in a computer system that allows interactions with it such as read, modify or delete*

DDL *Data Description Language, syntax for creating and modifying database objects such as tables, indices and users*

DML *Data Manipulation Language, syntax for adding, deleting or modifying data in a database*

DNS *Domain Name System, naming system for services in the Internet network*

ERD *Entity Relationship Diagram, used to describe interrelated things of interest in a specific domain of knowledge*

DTD *Document Type Definition, specification file from an XML document where data types and expected fields are listed. It can be used to validate an XML document*

FIFO *First In First Out, data manipulation way where the oldest element in queue is processed first*

GEO *Gene Expression Omnibus, the name of one data repository of NCBI*

GDS *GEO DataSet, a type of GEO database entity*

GPL *GEO Platform, a type of GEO database entity*

GSE *GEO Series, a type of GEO database entity*

GSM *GEO Sample, a type of GEO database entity*

HTTP *Hypertext Transfer Protocol, application layer protocol in the Internet for information systems*

IaaS *Infrastructure a Service, operating model in which computing resources are supplied by cloud service provider*

IAM *Identity and Access Management, framework of policies and technologies to ensure the right users have the appropriate access to technology resources. It is also the name AWS gives to its service to manage users, roles and policies*

IDE *Integrated Development Environment, software application that provides comprehensive facilities for software development*

IP *Internet Protocol address, numerical label assigned to a device to connect to a computer network*

ISP *Internet Service Provider, company that provides Internet access to individuals or businesses*

JSON *JavaScript Object Notation, data interchange format consisting on attribute-value pairs*

MVP *Minimum Viable Product, a version of a product with just enough features to be usable*

NCBI *National Center for Biotechnology Information, a website from USA government that hosts several databases for scientific studies and related information*

PaaS *Platform as a Service, cloud computing service model that allow customers to provision, run and manage a provided computing bundle*

PyPI *Python Package Index, a public repository of python packages*

RDS *AWS Relational Database Service. It allows users to operate relational databases without having to manage the underlying infrastructure*

RNA-Seq *RNA Sequencing. A technique that reveal the presence and quantity of RNA molecules in a biological sample*

S3 *Amazon Simple Storage Service, web based object storage service*

SaaS *Software as a Service, software licensing model based on a subscription and hosted centrally*

SAM *Serverless Application Model, open-source framework by AWS to create Lambda functions locally, without using the cloud*

SDN *Software-define networking, programmatic network management*

SES *Amazon Simple Email Service, on demand email service*

SNS *Amazon Simple Notification Service, notification service integrated with several notification platforms*

SQL *Structured Query Language, domain-specific language used to manage data especially in relational databases*

SQS *Amazon Simple Queue Service, a distributed message queuing service*

SRA *Sequence Read Archive, the name of one data repository of NCBI*

SSH *Secure Shell Protocol, cryptographic network protocol for operating remote machines securely over an insecure network such as the Internet*

TDD *Test Driven Development, software development approach where tests are written before the actual code*

UI *User Interface, space where interactions between a human and a system happens*

URL *Uniform Resource Locator, address on the Web for a resource*

VPC *Virtual Private Cloud, on-demand configurable pool of shared resources allocated within a public cloud environment, like AWS, providing a certain level of isolation between the different organizations*

XML *Extensible Markup Language, markup language for transmitting arbitrary data*

References

[1] NCBI Search Home Page, [online]. Available: https://www.ncbi.nlm.nih.gov/, June 2024.

[2] RNA-seq Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/RNA-Seq, June 2024.

[3] Phred Quality Score Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Phred_quality_score, June 2024.

[4] Elizabeth Carter and Robert C. Hubrecht. The 3Rs and Humane Experimental Technique: Implementing Change, [online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6826930/, June 2024.

[5] L. Collado-Torres, A. Nellore, and K. et al. Kammers. Reproducible RNA-seq analysis using recount2 [online]. Available: https://www.nature.com/articles/nbt.3838, June 2024.

[6] A. Lachmann, D. Torre, and A.B. et al. Keenan. Massive mining of publicly available RNA-seq data from human and mouse [online]. Available: https://www.nature.com/articles/s41467-018-03751-6, June 2024.

[7] The Human Protein Atlas, [online]. Available: https://www.proteinatlas.org/, June 2024.

[8] Mouse Allen Brain Atlas, [online]. Available: https://mouse.brain-map.org/static/atlas, June 2024.

[9] Eric Sayers. NCBI E-Utilities Handbook, [online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK25499/, June 2024.

[10] Saket Choudhary. Pysradb Open Source Documentation, [online]. Available: https://saket-choudhary.me/pysradb/, June 2024.

[11] Rob Edwards. SRA Metadata GitHub Project, [online]. Available: https://github.com/linsalrob/SRA_Metadata, June 2024.

[12] Saket Choudhary. Saket Choudhary Personal Website, [online]. Available: https://saket-choudhary.me/about/, June 2024.

[13] NCBI Traces Service, [online]. Available: https://trace.ncbi.nlm.nih.gov/Traces/sra-db-be/, June 2024.

[14] Giedrius Statkevičius. Push vs Pull In Monitoring Systems, [online]. Available: https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems/, June 2024.

[15] Marta Arcones Rodríguez. Github Project Repositiory, [online]. Available: https://github.com/arcones/sra-collector, June 2024.

[16] Github Flow, [online]. Available: https://docs.github.com/en/get-started/using-github/github-flow, June 2024.

[17] Alex Mullans. Dependabot, [online]. Available: https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/, June 2024.

[18] Alex Birsan. Dependency Confusion Attack, [online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, June 2024.

[19] Github Projects, [online]. Available: https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects, June 2024.

[20] Marta Arcones Rodríguez. SRA-Collector Project Management, [online]. Available: https://github.com/users/arcones/projects/1, June 2024.

[21] Marta Arcones Rodríguez. GitHub Milestones, [online]. Available: https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/about-milestones, June 2024.

[22] Amazon Web Services, [online]. Available: https://aws.amazon.com/, June 2024.

[23] Laura Bernheim. Cloud Provider Comparison, [online]. Available: https://www.hostingadvice.com/how-to/aws-azure-google-cloud-alternatives/, June 2024.

[24] Amazon Web Services official documentation, [online]. Available: https://docs.aws.amazon.com/, June 2024.

[25] Amazon Web Services in Stackoverflow, [online]. Available: https://stackoverflow.com/questions/tagged/amazon-web-services, June 2024.

[26] Amazon Web Services re:Post blog, [online]. Available: https://repost.aws/, June 2024.

[27] Types of Cloud Computing, [online]. Available: https://aws.amazon.com/types-of-cloud-computing/, June 2024.

[28] AWS VPC Documentation, [online]. Available: https://aws.amazon.com/vpc/, June 2024.

[29] Software-Defined Networking Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Software-defined_networking, June 2024.

[30] AWS IAM Documentation, [online]. Available: https://aws.amazon.com/iam/, June 2024.

[31] Principle Of Least Privilege Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Principle_of_least_privilege, June 2024.

[32] AWS Route 53 Documentation, [online]. Available: https://aws.amazon.com/route53/, June 2024.

[33] AWS API Gateway Documentation, [online]. Available: https://aws.amazon.com/api-gateway/, June 2024.

[34] Cross-Origin Resource Sharing Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Cross-origin_resource_sharing, June 2024.

[35] API Gateway Concurrency Limits, [online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html#api-gateway-execution-service-limits-table, June 2024.

[36] AWS Cognito Documentation, [online]. Available: https://aws.amazon.com/cognito/, June 2024.

[37] AWS Lambda Documentation, [online]. Available: https://aws.amazon.com/lambda/, June 2024.

[38] Serverless Computing Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Serverless_computing, June 2024.

[39] AWS SQS Documentation, [online]. Available: https://aws.amazon.com/sqs/, June 2024.

[40] AWS RDS Documentation, [online]. Available: https://aws.amazon.com/rds/, June 2024.

[41] AWS Secrets Manager Documentation, [online]. Available: https://aws.amazon.com/secrets-manager/, June 2024.

[42] AWS S3 Documentation, [online]. Available: https://aws.amazon.com/s3/, June 2024.

[43] AWS SES Documentation, [online]. Available: https://aws.amazon.com/ses/, June 2024.

[44] AWS SES Documentation, [online]. Available: https://aws.amazon.com/sns/, June 2024.

[45] AWS Cloudwatch Documentation, [online]. Available: https://aws.amazon.com/cloudwatch/, June 2024.

[46] AWS OpenSearch Documentation, [online]. Available: https://aws.amazon.com/opensearch-service/, June 2024.

[47] Terraform Documentation, [online]. Available: https://developer.hashicorp.com/terraform?product_intent=terraform, June 2024.

[48] Terraform AWS Modules, [online]. Available: https://registry.terraform.io/browse/modules?provider=aws, June 2024.

[49] Anton Babenko. Terraform Registry VPC Module, [online]. Available: https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest, June 2024.

[50] Marta Arcones Rodríguez. Sra-Collector VPC Module Usage, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/network.tf#L3-L13, June 2024.

[51] Marta Arcones Rodríguez. Custom Data Pipeline Processor Module, [online]. Available: https://github.com/arcones/sra-collector/tree/main/infra/lambdas/infra, June 2024.

[52] Marta Arcones Rodríguez. Custom Data Pipeline Processor Module Usages, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/lambdas/main.tf#L12-L179, June 2024.

[53] Docker Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Docker_(software), June 2024.

[54] GitHub-Hosted Runners, [online]. Available: https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners, June 2024.

[55] Container Images In Lambda Functions, [online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/python-image.html, June 2024.

[56] Marta Arcones Rodríguez. SRA-Collector Docker Dependencies Build, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/lambdas/docker/Dockerfile, June 2024.

[57] Python Docker Official Images, [online]. Available: https://hub.docker.com/_/python, June 2024.

[58] Python Developer Guide, [online]. Available: https://devguide.python.org/, June 2024.

[59] AWS Lambda Runtimes, [online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html, June 2024.

[60] Python in Stackoverflow, [online]. Available: https://stackoverflow.com/questions/tagged/python, June 2024.

[61] Python Package Index, [online]. Available: https://pypi.org/, June 2024.

[62] Python For BioInformatics, [online]. Available: https://omicstutorials.com/top-ten-programming-languages-for-bioinformatics-in-2023/, June 2024.

[63] AWS Example Cloudwatch To Opensearch Lambda Exporter, [online]. Available: https://github.com/awslabs/hids-cloudwatchlogs-opensearch-template/blob/master/lambda/index.js, June 2024.

[64] GNU Make Documentation, [online]. Available: https://www.gnu.org/software/make/, June 2024.

[65] PostgreSQL, [online]. Available: https://www.postgresql.org/, June 2024.

[66] Sara A. Metwalli. Python Databases 101, [online]. Available: https://builtin.com/data-science/python-database, June 2024.

[67] Psycopg Library, [online]. Available: https://www.psycopg.org/, June 2024.

[68] Psycopg Repository Pulse, [online]. Available: https://github.com/psycopg/psycopg/pulse/monthly, June 2024.

[69] RDS Pricing, [online]. Available: https://aws.amazon.com/rds/pricing/, June 2024.

[70] Flyway Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Flyway_(software), June 2024.

[71] Marta Arcones Rodríguez. Database Migrations At Sra-Collector, [online]. Available: https://github.com/arcones/sra-collector/tree/main/db/migrations, June 2024.

[72] Marta Arcones Rodríguez. SRA-Collector Database Migration Execution For Local Development, [online]. Available: https://github.com/arcones/sra-collector/blob/main/Makefile#L9-L17, June 2024.

[73] Marta Arcones Rodríguez. SRA-Collector Database Migration Execution For CI/CD, [online]. Available: https://github.com/arcones/sra-collector/blob/main/.github/workflows/db.yml#L11-L16, June 2024.

[74] Martin Fowler. Test Pyramid, [online]. Available: https://martinfowler.com/bliki/TestPyramid.html, June 2024.

[75] Marta Arcones Rodríguez. SRA-Collector Unit Tests Suite, [online]. Available: https://github.com/arcones/sra-collector/blob/main/tests/unit_tests/lambdas_unit_test.py, June 2024.

[76] H2 Database Engine Documentation, [online]. Available: http://h2database.com/html/main.html, June 2024.

[77] Abraham Enyo one Musa. What Is Mocking In Unit Testing, [online]. Available: https://abraham-musa.medium.com/what-is-mocking-in-unit-testing-a-data-scientists-perspective-explained-with-practical-use-cases-c2ba2e75cdb7, June 2024.

[78] Marta Arcones Rodríguez. SRA-Collector Unit Tests CI/CD Workflow, [online]. Available: https://github.com/arcones/sra-collector/blob/main/.github/workflows/deploy.yml#L11-L20, June 2024.

[79] AWS Serverless Application Model Documentation, [online]. Available: https://docs.aws.amazon.com/serverless-application-model/, June 2024.

[80] Marta Arcones Rodríguez. SRA-Collector Integration Tests Suite, [online]. Available: https://github.com/arcones/sra-collector/blob/main/tests/integration_tests/lambdas_integration_test.py, June 2024.

[81] Marta Arcones Rodríguez. SRA-Collector Integration Tests CI/CD Workflow, [online]. Available: https://github.com/arcones/sra-collector/blob/main/.github/workflows/integration-test.yml, June 2024.

[82] Reinventing The Wheel Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Reinventing_the_wheel, June 2024.

[83] Pysradb Bug Tracker, [online]. Available: https://github.com/saketkc/pysradb/issues, June 2024.

[84] Cloudwatch Live Tail, [online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CloudWatchLogs_LiveTail.html, June 2024.

[85] Cloudwatch Insights, [online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/appinsights-what-is.html, June 2024.

[86] About Opensearch, [online]. Available: https://opensearch.org/about.html, June 2024.

[87] Elastic Stack: ElasticSearch & Kibana, [online]. Available: https://www.elastic.co/elastic-stack, June 2024.

[88] Opensearch Index Automatic Creation, [online]. Available: https://opensearch.org/docs/latest/api-reference/index-apis/create-index/, June 2024.

[89] AWS Free Tier, [online]. Available: https://aws.amazon.com/free/, June 2024.

[90] Marta Arcones Rodríguez. Makefile OpenSearch Purge Action, [online]. Available: https://github.com/arcones/sra-collector/blob/main/Makefile#L34-L37, June 2024.

[91] Eric Sayers. NCBI E-Utilities Handbook, [online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK25497/#chapter2.The_Nine_Eutilities_in_Brief, June 2024.

[92] Pycharm IDE, [online]. Available: https://www.jetbrains.com/pycharm/, June 2024.

[93] AWS Toolkit For Jetbrains, [online]. Available: https://docs.aws.amazon.com/toolkit-for-jetbrains/latest/userguide/setup-toolkit.html, June 2024.

[94] Pre-commit Hooks Documentation, [online]. Available: https://pre-commit.com/, June 2024.

[95] Pre-commit Supported Hooks, [online]. Available: https://pre-commit.com/hooks.html, June 2024.

[96] Marta Arcones Rodríguez. SRA-Collector Pre-commit Hooks, [online]. Available: https://github.com/arcones/sra-collector/blob/main/.pre-commit-config.yaml, June 2024.

[97] cURL Client Wikipedia Entry, [online]. Available: https://es.wikipedia.org/wiki/CURL, June 2024.

[98] Swagger Wikipedia Entry, [online]. Available: https://es.wikipedia.org/wiki/Swagger_(software), June 2024.

[99] OpenAPI Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/OpenAPI_Specification, June 2024.

[100] Marta Arcones Rodríguez. SRA-Collector Swagger Interface, [online]. Available: https://arcones.github.io/sra-collector, June 2024.

[101] SchemaSpy Docs, [online]. Available: https://schemaspy.org/, June 2024.

[102] Marta Arcones Rodríguez. SRA-Collector SchemaSpy Configuration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/schemaspy.properties, June 2024.

[103] Marta Arcones Rodríguez. SRA-Collector SchemaSpy Makefile Integration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/Makefile#L19-L22, June 2024.

[104] Marta Arcones Rodríguez. SRA-Collector SchemaSpy GitHub Actions Integration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/.github/workflows/db.yml#L17-L32, June 2024.

[105] SchemaSpy Docker Image, [online]. Available: https://hub.docker.com/r/schemaspy/schemaspy/, June 2024.

[106] Ubuntu Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Ubuntu, June 2024.

[107] Ubuntu GitHub Actions Runner, [online]. Available: https://github.com/actions/runner-images/blob/main/images/ubuntu/Ubuntu2204-Readme.md, June 2024.

[108] Fail-fast Systems Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Fail-fast_system, June 2024.

[109] Fail-safe Systems Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Fail-safe, June 2024.

[110] Exponential Backoff Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Exponential_backoff, June 2024.

[111] Single-responsibility Principle Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Single-responsibility_principle, June 2024.

[112] Unix Philosophy Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Unix_philosophy#Do_One_Thing_and_Do_It_Well, June 2024.

[113] Best-effort Delivery Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Best-effort_delivery, June 2024.

[114] James D. McCabe. Best-Effort Service, [online]. Available: https://www.sciencedirect.com/topics/computer-science/best-effort-service, June 2024.

[115] Marta Arcones Rodríguez. Kilombo Application, [online]. Available: https://github.com/arcones/kilombo, June 2024.

[116] Sebastián Ramírez Montaño. FastAPI Framework, [online]. Available: https://fastapi.tiangolo.com/, June 2024.

[117] Asynchronous System Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Asynchronous_system, June 2024.

[118] Event-driven Architecture Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Event-driven_architecture, June 2024.

[119] RabbitMQ Documentation, [online]. Available: https://www.rabbitmq.com/, June 2024.

[120] Azure Queue Storage, [online]. Available: https://azure.microsoft.com/en-us/products/storage/queues/, June 2024.

[121] Cloud-native Computing Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Cloud-native_computing, June 2024.

[122] Test-driven Development Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Test-driven_development, June 2024.

[123] Security By Design Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Secure_by_design, June 2024.

[124] Gui Alvarenga. Shift Left Security, [online]. Available: https://www.crowdstrike.com/cybersecurity-101/shift-left-security/, June 2024.

[125] Infrastructure As Code Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Infrastructure_as_code, June 2024.

[126] Infrastructure As Code Recommendations, [online]. Available: https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/infrastructure-as-code-design, June 2024.

[127] Declarative Programming Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Declarative_programming, June 2024.

[128] Don't Repeat Yourself Code Principle Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Don't_repeat_yourself, June 2024.

[129] Marta Arcones Rodríguez. SRA-Collector Libraries, [online]. Available: https://github.com/arcones/sra-collector/tree/main/infra/lambdas/docker, June 2024.

[130] Networking Control Plane Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Control_plane, June 2024.

[131] Networking Data Plane Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Data_plane, June 2024.

[132] HTTP Response Codes Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes, June 2024.

[133] Marta Arcones Rodríguez. A_get_user_query Lambda Code, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/lambdas/code/A_get_user_query.py, June 2024.

[134] AWS Standard SQS Queue Characteristics, [online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues.html, June 2024.

[135] AWS SQS FIFO Queues Documentation, [online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-fifo-queues.html, June 2024.

[136] AWS SQS Pricing, [online]. Available: https://aws.amazon.com/sqs/pricing/, June 2024.

[137] Idempotence In Computer Science Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Idempotence, June 2024.

[138] Marta Arcones Rodríguez. B_get_query_pages Lambda Code, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/lambdas/code/B_get_query_pages.py, June 2024.

[139] Marta Arcones Rodríguez. C_get_study_ids Lambda Code, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/lambdas/code/C_get_study_ids.py, June 2024.

[140] NCBI Registration Web, [online]. Available: https://account.ncbi.nlm.nih.gov/, June 2024.

[141] Observability Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Observability, June 2024.

[142] Marta Arcones. SRA-Collector Alarms Configuration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/cloudwatch.tf, June 2024.

[143] Marta Arcones. SRA-Collector Lambdas Error Ratio Alarms Configuration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/cloudwatch.tf#L37-L83, June 2024.

[144] Marta Arcones. SRA-Collector Lambdas Error Ratio Alarms Thresholds, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/cloudwatch.tf#L14-L24, June 2024.

[145] Structured Logging, [online]. Available: https://sematext.com/glossary/structured-logging/, June 2024.

[146] Marta Arcones. SRA-Collector OpenSearch Dashboards Versioning, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/opensearch/export.ndjson, June 2024.

[147] Marta Arcones. SRA-Collector OpenSearch Instance Configuration, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/opensearch/domain.tf#L9, June 2024.

[148] Marta Arcones. SRA-Collector OpenSearch Clean Indices Functionality, [online]. Available: https://github.com/arcones/sra-collector/blob/main/Makefile#L34-L37, June 2024.

[149] Horizontal Scaling Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Scalability#Horizontal_or_scale_out, June 2024.

[150] AWS Aurora RDS Documentation, [online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html, June 2024.

[151] Marta Arcones. SRA-Collector Message Processing Timeouts, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/sqs.tf#L56, June 2024.

[152] AWS Lambda Documentation, [online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html, June 2024.

[153] Rohit Akiwatkar. AWS Lambda vs EC2: Comparison of AWS Compute Resources, [online]. Available: https://www.simform.com/blog/aws-lambda-vs-ec2/, June 2024.

[154] Quotas and constraints for Amazon RDS, [online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html, June 2024.

[155] Aleksandra Duło. Mail Servers Attachment Limits, [online]. Available: https://emaillabs.io/en/max-size-of-email-attachment-and-size-limit-explained/, June 2024.

[156] Amazon S3: Allows Amazon Cognito users to access objects in their bucket, [online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_s3_cognito-bucket.html, June 2024.

[157] AWS Savings Plans Documentation, [online]. Available: https://aws.amazon.com/savingsplans/, June 2024.

[158] Vendor Lock-In Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Vendor_lock-in, June 2024.

[159] AWS rePost On SQS Message Prioritization, [online]. Available: https://repost.aws/questions/QUo4Ij8z8LR4Og66aT6hPP8g/can-sqs-standard-queue-give-priority-to-some-messages-in-the-queue, June 2024.

[160] AWS Cognito Sign-up Methods, [online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html, June 2024.

[161] Marta Arcones. SRA-Collector OpenSearch IP Filtering, [online]. Available: https://github.com/arcones/sra-collector/blob/main/infra/opensearch/domain.tf#L37-L39, June 2024.

[162] OpenSearch Authentication Usign Cognito Documentation, [online]. Available: https://docs.aws.amazon.com/opensearch-service/latest/developerguide/cognito-auth.html, June 2024.

[163] Minimum Viable Product Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Minimum_viable_product, June 2024.

[164] Tetiana Stoyko. Angular vs React vs Vue: The Main Differences and Use Cases, [online]. Available: https://incora.software/insights/react-vs-angular-vs-vue, June 2024.

[165] MUI Component Library, [online]. Available: https://mui.com/, June 2024.

[166] ProsperousPlus Application, [online]. Available: http://prosperousplus.unimelb-biotools.cloud.edu.au/index.php, June 2024.

[167] Crow's Foot Notation Wikipedia Entry, [online]. Available: https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model#Crow's_foot_notation, June 2024.

[168] Cloudcraft Tool, [online]. Available: https://www.cloudcraft.co/, June 2024.

[169] ABET Accreditation, [online]. Available: https://www.abet.org/accreditation/, June 2024.

[170] EUR-ACE Accreditation, [online]. Available: https://www.enaee.eu/eur-ace-system/, June 2024.

[171] 2030 Agenda, [online]. Available: https://sdgs.un.org/goals, June 2024.

[172] Nuno Henrique Franco. Animal Experiments in Biomedical Research: A Historical Perspective, [online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4495509/, June 2024.

[173] Ghanshyam Das Gupta & Balak Das Kurmi Abhinav Vashishat, Preeti Patel. Alternatives of Animal Models for Biomedical Research: a Comprehensive Review of Modern Approaches, [online]. Available: https://link.springer.com/article/10.1007/s12015-024-10701-x, June 2024.