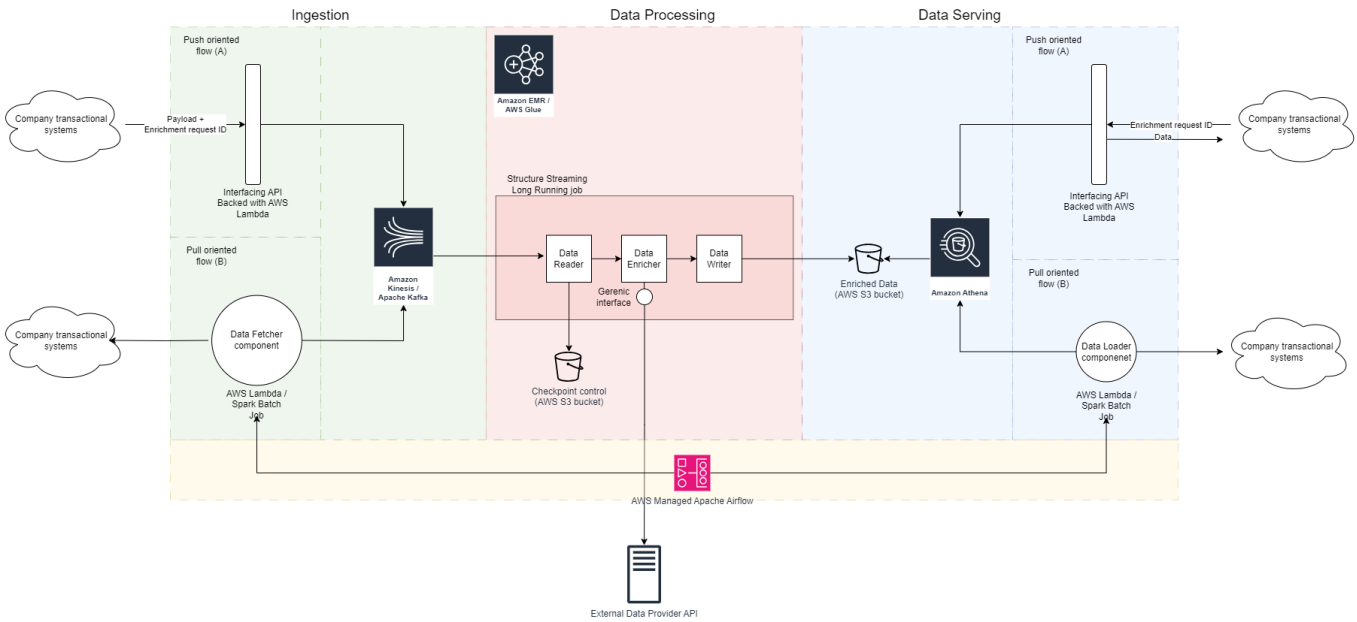
Data Enrichment Service Achitecture

This document aims to demonstrate the proposed design for the Data Enrichment Service and its components. In order to facilitate the reading, let us name the service as DES.

Solution design

The proposed long-term/ideal architecture diagram for the service described in the Take Home Assesment requirements is defined below:



Now let's dive in more details in each component of the proposed solution, in order to understand what role each one plays on it and how they interact with each other.

Ingestion layer

In order to start the DES ingestion layer, let us break it in two ways for inputting data into the service: (1) Push-oriented and (2) Pull-oriented methods. This definition takes into account that each company transaction service might have it's own level of maturity/complexity, therefore we need to provide a flexible integration as possible for others take advantage of our services provided.

Push-oriented

This approach is triggered by the transactional service that wants to enrich its data. For that, it was defined a interfacing API component on which this service can talk with the DES starting the enrichment request; it needs to send a JSON containing both the payload (contact list and a list of fields to be enriched), a **enrichment resquest ID (ERID)** and the total **number_of_records**. Here is an example:

```
POST https://des.internal.sayprimer.com/v2/enriched_data
{
    "erid": "3c3f4b80-f728-4d7c-8567-1bc01df5f0a1",
    "contact_list": [
```

```
        {
            "first_name": "John",
            "last_name": "Doe",
            "company_name": "sayprimer.com"
        },
        {
            "first_name": "Foo",
            "last_name": "Bar",
            "company_name": "sayprimer.com"
        },
        ...
    ],
    "fields_to_be_enriched": [
        "professional_email",
        "phone_number"
    ],
    "number_of_records": 100000
}
```

The `erid` is an important part of the flow because it defines a single enrichment transaction and as long as the requests made to the DES interfacing API have the same `erid`, they will be grouped at the output, in a single data object that can be referred to later.

The `number_of_recrds` should be used in the subsequent layers in order to control whether a particular Data Enrichment Transaction has finished or not.

It's important to highlight that it's an asynchronous request, therefore no enriched response will be returned at this point. Each request containing thousands of data objects will be forwarded to a Streaming Broker/Messaging System, so they can be processed and written in a near real-time fashion. Please check for more details in Processing Layer and Processing Layer.

P.S: Even though the diagram doesn't show the authentication on the API side, we can leverage some functionalities on AWS API Gateway for that purpose keeping our API secure and compliance.

Pull-oriented

In this method, the service can't, for any reason, integrate with our Asynchronous Ingest API but still, it needs to enrich data. The proposed architecture contemplates this situation as well, where instead of receiving the data enrichment requests, the DES service will start the process.

To accomplish this requirement, there are two key components of the architecture:

- A workflow orchestrator, AWS MWAA (Managed Workflow with Apache Airflow), accountable for managing the flows, job schedules, and dependencies for the data-pulling process
- An Data Fetcher component in a form of an ETL (read more on Config-based ETL framework), built on top of AWS Lambda or Spark batch jobs, that can handle multiple data sources using generic interfaces and specialized mechanism source-oriented implemented using object inheritance (circles on the architecture).

Either the service has to query the transactional data directly in their database, or APIs or has to read from an intermediate storage layer (s3 buckets, FTP serves, and others), by using the pull-oriented method, the service will also be able to bring the data to the service and process it in a scalable fashion. After querying the data on the source, it will forward it to the Broker/Messaging system that keeps the boundaries between the ingestion and processing layer.

One important aspect to highlight is that, even though there is no external service starting the data enrichment transaction, the Data Fetcher component must also generate a `erid` field so we can keep consistent on the data serving layer. In order to mediate the conversation between the Data Fetcher and the Data Loader components (see Serving data in a Pull-oriented flow), we can leverage AWS Managed Airflow in order to hold the `erid` for a particular ETL execution.

Broker/Messaging system

It will be one of the key components of the DES architecture that will provide us high scalability, as well as a fault-tolerant service which linked to the processing layer will form the high performatic core of the solution.

The broker will act as a data storage engine for real-time data reading, allowing streaming processes to read from it in a performant way, at the same time ensuring the data consistency with an *end-to-end exactly-once* semantics in case of failure.

Processing Layer

This layer is the DES functionality core, accountable for performing the data enrichment in fact. The approach chosen was streaming given the requirements for high performance and fault-tolerance, which means, a single job can scale from 1 to N distributed executors, increasing the parallel computation power ensuring the fault-tolerance at the same time. The suggested streaming job can be built on top of AWS EMR or AWS Glue framework and its logic will consist of three main internal "controllers":

- Data reader: process in charge of reading the data from the Broker as well as managing the checkpointing of the solution, so in case of failures, the solution won't either generate data duplication or data holes. It will start exactly from where it was stopped (using the offset concept)
- Data Enricher: it's the process that will perform the enrichment of the data, using external APIs for this purpose. Therefore it needs to control the connection to the N Data Source APIs implemented, the data rate that will be sent in each request, the number of requests, and other configurations (see more details below)
- Data Writer: it's a process in charge of writing the data in micro-batches to the output storage (s3 bucket), allowing the enriched data to be served.

Data Enricher

Interactions with APIs such as authentication, pagination, and endpoint convention might employ different techniques/frameworks, but the overall workflow tends to follow the same pattern. Thinking on that, the Data Enricher must be well developed mainly because it needs to provide a unique interface for any Data Provider API, making it easy to extend and not demand too much operational cost at each new integration.

Also, under the Data Enricher umbrella, we also need to define a good balance for the number of parallel requests performed to the external Data Source API that will allow us to read more data in a low-latency basis but at the same time won't compromise our Data Provider partners performance/availability.

Data Writer

When writting the data, we want to make sure we control the partitions of the When writing the data, we want to make sure we control the partitions of the data (see in Storage Structure). Not only that, but we also want to control the Data Enrichment Transactions that are still in progress. To keep track of the progress of the data, the Data writer must hold a counter that will be incremented each time it writes data to the output. A mechanism can be built on top of it to make sure the data consumer will only see the data when it's fully processed, see more on the Incomplete Enrichment Transactions section.

In the Data Writer process, we need to evaluate the rate at which we are going to output data, taking into account that the more frequently the data is written, more closer to real-time it is, but also more granular files it generates increasing the I/O time on the consumer side. It's a trade-off.

Serving Layer

In the final layer of the DES, there are the output data storage (AWS S3 bucket) and the query engine to access the data (AWS Athena). Also, since in the service input, it was offered two methods for ingesting data, it will be also offered the same methods so we can cover most company transactional services capabilities.

Storage strucure

The processing layer should write data in a structure that will allow them to be read by the consumers in a performatic and consistent way. Therefore, it was chosen the following partitioning schema for the data:

```
s3://output-bucket/enriched_data/
    request_date=2024-02-09/
        erid=3c3f4b80-f728-4d7c-8567-1bc01df5f0a1/
            <parquet files>
        erid=ef36c16b-735b-4e14-8f94-a70c33fa50c7/
            <parquet files>
        ...
    request_date=2024-02-10/
        erid=4bb24762-80f0-4f78-99a6-1f02d1efe4c2/
            <parquet files>
        erid=4e3441d9-9d6d-4da5-ad55-289442d20d96/
            <parquet files>
        erid=107e45b1-190e-48b5-acb1-e78f8678ed38/
            <parquet files>
        ...
    ...
```

We want to make sure that whenever a query is executed, it won't scan the full table, so the 1st partition defined for the `enriched_data` table is the `request_date` (date when the data enrichment request was performed). As long as the consumer includes the date range in the query, the engine will only scan the related partitions.

Since we also want to group each Data Enrichment Transaction in a single data entity, the 2nd partition defined for the table is the `erid` field. With these two partitions defined, the executed queries will be able to retrieve the exact enriched data they will look for.

Incomplete Enrichment Transactions

The Data Writer phase is accountable for making sure the consumers will only see the data when it's fully processed and it can be done at the partition level. Essentially, while the row counter on the Data Writer is lower than `number_of_records` (parameter passed from the ingesting phase up to the processing job) the data will be saved in a partition with a suffix `_incomplete` at the `erid` partition name. Whenever the records counter has reached the `number_of_records` from the request, it removes the suffix from the partition. Below is an example:

```
s3://output-bucket/enriched_data/
    request_date=2024-02-09/
        erid=3c3f4b80-f728-4d7c-8567-1bc01df5f0a1/ # <= completed data enrichment
transaction
            <parquet files>
        erid=ef36c16b-735b-4e14-8f94-a70c33fa50c7_incomplete/ # <= incomplete data
enrichment transaction
            part-0000.parquet
            ....
        ...
```

Serving data in a Push-oriented flow

For the services that have push capabilities, it was offered also a push-oriented approach for retrieving the data, build on top of AWS API Gateway and AWS Lambda. This component will be in charge of transforming HTTP requests in Athena queries executed over the output S3 Bucket data.

Each request will be composed by a date range (start_date, end_date parameters) and the `erid` field. For example, below is the HTTP request and the formed query:

```
# HTTP request
GET https://des.internal.sayprimer.com/v2/enriched_data?start_date=2024-02-
08&end_date=2024-02-11&erid=3c3f4b80-f728-4d7c-8567-1bc01df5f0a1


# Athena query
SELECT
    *
FROM
    enriched_data
WHERE
    request_date >= '2024-02-08' AND request_date <= '2024-02-11'
    AND erid = '3c3f4b80-f728-4d7c-8567-1bc01df5f0a1'
```

This reponse will be composed by the payload (input data + enriched fields) as well as the fields to control the pagination for large datasets (number of pages and the next page, number of items per page, etc). Here is an example of a response data:

```
{
    "erid": "3c3f4b80-f728-4d7c-8567-1bc01df5f0a1",
    "data": [
        {
            "first_name": "John",
            "last_name": "Doe",
            "company_name": "sayprimer.com",
            "professional_email": "john.doe@sayprimer.com",
            "phone_number": "123456789"
        },
        {
            "first_name": "Foo",
            "last_name": "Bar",
            "company_name": "sayprimer.com",
            "professional_email": "foo.bar@sayprimer.com",
            "phone_number": ""
        },
        ...
    ],
    "number_of_pages": 200,
    "next_page": 2,
    "status": "complete"
}
```

Status request and status check

The field `status` will play an important role in the Response, it has three possible values essentially:

- "inexistent": this status represents the absence of data. It might happen for two reasons: (1) a company service submits a data enrichment request to DES, but when requesting a GET on the same `erid` on the output API, the streaming process wasn't ready yet (resource allocation, temporary outage, etc) or (2) the requested `erid` don't exist, meaning the data enrichment request was never performed.
- "incomplete": this status represents an in-progress data enrichment transaction
- "completed": this status represents a completed data enrichment transaction

In the query of the previous section, only a single query statement was executed because it was enough to return more than zero rows, marked then as `completed`, however in a case where it returns zero records, the component must look at the incomplete transaction partitions (with `_incomplete` suffix) to know if the status of a transaction is either `incomplete` or `inexistent`. Here is an example of the same request, but now, zero records were found in the first query:

```
# HTTP request
GET https://des.internal.sayprimer.com/v2/enriched_data?start_date=2024-02-
08&end_date=2024-02-11&erid=3c3f4b80-f728-4d7c-8567-1bc01df5f0a1


# Athena query - zero records found
SELECT
    *
```

```
FROM
    enriched_data
WHERE
    request_date >= '2024-02-08' AND request_date <= '2024-02-11'
    AND erid = '3c3f4b80-f728-4d7c-8567-1bc01df5f0a1'

# Athena query - more than zero records found
SELECT
    *
FROM
    enriched_data
WHERE
    request_date >= '2024-02-08' AND request_date <= '2024-02-11'
    AND erid = '3c3f4b80-f728-4d7c-8567-1bc01df5f0a1_incomplete'
```

In this particular case, the status will be considered `incomplete`. In case the 2nd query also doesnt't return rows, it will be marked as `inexistent`.

Since the Data Enrichment Transaction is asyncronous, the service that push the request, may also query the output endpoint from time to time in order to know whether the data is already available. Using the status mentioned above, the consumer may want to implement some rules in order to properly query the status of the request, so please consider the following python code:

```
# Data Enrichment Transaction request
post_data_enrichment_transaction(input_data, enrich_fields)

# ...
# Get Data Enriched request
response = get_enriched_data(start_date, end_date, erid)
attempt_inexistent = 0

# Check for inexistent data
while(attempt_inexistent < N_MAX_ATTEMPTS_INEXISTENT and response.status ==
'inexistent'):
    wait_exponential_rate_seconds()

    response = get_enriched_data(start_date, end_date, erid)
    attempt_inexistent++

if(response.status == 'inexistent'):
    print("Data not found")

    # re-do the Data Enrichment Transaction request using another erid

else:
    # check for incomplete data
    attempt_incomplete = 0
    while(attempt_incomplete < N_MAX_ATTEMPTS_INCOMPLETE and response.status ==
'incomplete'):
        wait_some_seconds()
```

```
        response = get_enriched_data(start_date, end_date, erid)
        if(response.status == 'completed'):
            break

        attempt_incomplete++

    # do the logic with the retrieved data.
```

It's important to set values for `N_MAX_ATTEMPTS_INEXISTENT` and `N_MAX_ATTEMPTS_INCOMPLETE`, so we don't keep the company service requesting the DES APIs forever. For `N_MAX_ATTEMPTS_INCOMPLETE`, it's important to recognize that depending on the dataset, we might set this to a higher value.

Serving data in a Pull-oriented flow

Both Data Fetcher (see in the Pull-oriented ingestion section) and Data Loader components were designed to be part of an ETL framework built to read/write data from/to many source systems. The orchestrator tool, AWS MWAA, is in charge of mediating the communication between the two jobs, essentially passing the `erid` from Data Fetcher to Data Loader.

With the `erid`, the job can start retrieving the data using Athena and writing data on the destination. Since it's built on top of an ETL framework with generic interfaces for both sources and sinks, the Data Loader will interact with the interfaces thal will talk to the inherited-specialized objects that will interact with the destination source. This will make the whole system more extensible and whenever new sources/destinations are added, it's only a matter of extending the interfaces and implement needed methods.

Config-based ETL framework

Whenever is possible, we want to avoid write customized code, making the service generic as necessary to attend many different use cases. In the case of pull-based ingestion, this isn't different; the engineering teams don't want to spend time writing new jobs whenever a new company service requests an integration for data enrichment. Based on this statement, we can create config-based workflows on the ETL framework on which each company microservice team should be able to add their own config files and the DES should be able to deploy automatically a new ETL pipeline for it. A config file can be similar to the following:

```
pipelineName: enrich_marketing_data
dataSource:
    - type: mysql
    - extraOptions:
        - hostname: marketing-service.internal
        - port: 3306
        - password: ENV_PASSWORD
        ...
    - table: users
    - fields: [first_name, last_name, company]
    - fields_to_enrich: [company_email, phone_number]
destination:
    - type: mysql
    - extraOptions:
        - hostname: output-service.internal
```

```
        - port: 3306
        - password: ENV_OUT_PASSWORD
        ...
    - table: enriched_users
```

Monitorability

The DES components have native integration with AWS CloudWatch, so it has been defined as the main Monitoring/Observability tool. Additionally to the logs, it's possible to access Apache Spark UI to track the Streaming Application status, as well as some other monitoring UIs available on EMR (Ganglia) for job tunning. In the case of AWS Glue, it also has some built-in UIs for monitoring, integrated with CloudWatch.

In the case of metrics, each component can output its own set of metrics, so we can efficiently debug incidents:

- APIs for interfacing input/output data: we can monitor the traffic that has been flowing through. Number of requests, number of errors 4xx, number of errors 5xx, and latency. In the case of API Gateway, all those metrics should be available. For lambdas, we can also see the number of invocations/concurrent executions, throttles, and others. It's also important to expose a `/status` endpoint, so other external services will be able to know if DES is fully operant before sending data.
- Data Fetcher component / Data Loader components: the ETL framework built to accommodate these two pipelines must output metrics associated with the source data quality so we can make sure the ingestion/loading is working as expected when problems arise. Some examples are the number of distinct records, the number of rows with null values when reading and when writing, and others.
- AWS Kinesis Stream also is an important component and it offers some built-in metrics: the number of successful GetRecords operations and, the number of PutRecord.Bytes, number of PutRecords.FailedRecords and so on.
- Structure streaming enrichment job: this core component on the architecture needs to output some important metrics for each Data Enrichment Transaction, so the team knows it's operating properly: number of records fetched from AWS Kinesis Stream, number of records enriched, number of records that weren't found on the External Data Providers, number of records written, number of Data Enrichment Transactions being processed and others.
- On the Enriched Data s3 bucket, we might find it useful to watch some metrics for example number of PutRequests. If this number is high, the streaming engine might have been writing too small files on the bucket. The number of GetRequests will also be useful to track the number of queries that have been executed against it.
- AWS Athena: we might define some metrics on the Athena side to make sure the data downstream consumers are not facing any issues when reading the data, for example, QueryPlanningTime, QueryQueueTime, TotalExecutionTime. A higher value for these numbers might also indicate an issue on the storage side or resource allocation.

With those metrics established, we may leverage thresholds and alerts on AWS Cloudwatch, so we can be alerted whenever abnormal behavior has been found on DES

Deployment strategy

The DES components might be categorized into two types when it comes to deploying new changes: (1) the critical 24x7 and, (2) non-critical components.

The first group, the critical components, will require a more robust deployment process since the main goal is to keep the service running without interruptions when either deploying or rolling back changes. Will fall in this category the interfacing input/output APIs, AWS Lambda, AWS Kinesis, and AWS Athena, the streaming processing job, and the output S3 bucket. All AWS native services will offer strategies to keep versions and will change among them without interruptions.

The streaming processing job will require a more robust CI/CD pipeline and since we cannot run two streaming jobs at the same time, otherwise the processing engine might generate duplicated data on the output, the automated operations for the deployment will have to be defined as follows:

1. whenever there are changes on the streaming processing job, the code starts the artifact building and artifact copying to the right location.
2. when it's ready to be deployed, the CI pipeline will have to interact with the framework (AWS EMR/AWS Glue) to request the old streaming job to stop
3. the new streaming job will be requested to start, using the same checkpoint as the previous one (starting from where the previous one has stopped)
4. in case of rollback, after submitting the revert PR on the repository, it will repeat this process.

Even though it's an error-prove CI/CD approach, it will result in some minutes of data being accumulated on AWS Kinesis/Apache Kafka and data not available in the output bucket as a result of the streaming processing job being kept stopped.

For the non-critical components, mostly the pull-based approach components, the team might be able to deploy their changes normally, since a Data Fetcher/Load job won't be interrupted for a code change deployment. Also, it's important to keep the idempotency of the data pipelines, so they can be re-executed in case of rollback and still produce the same results.