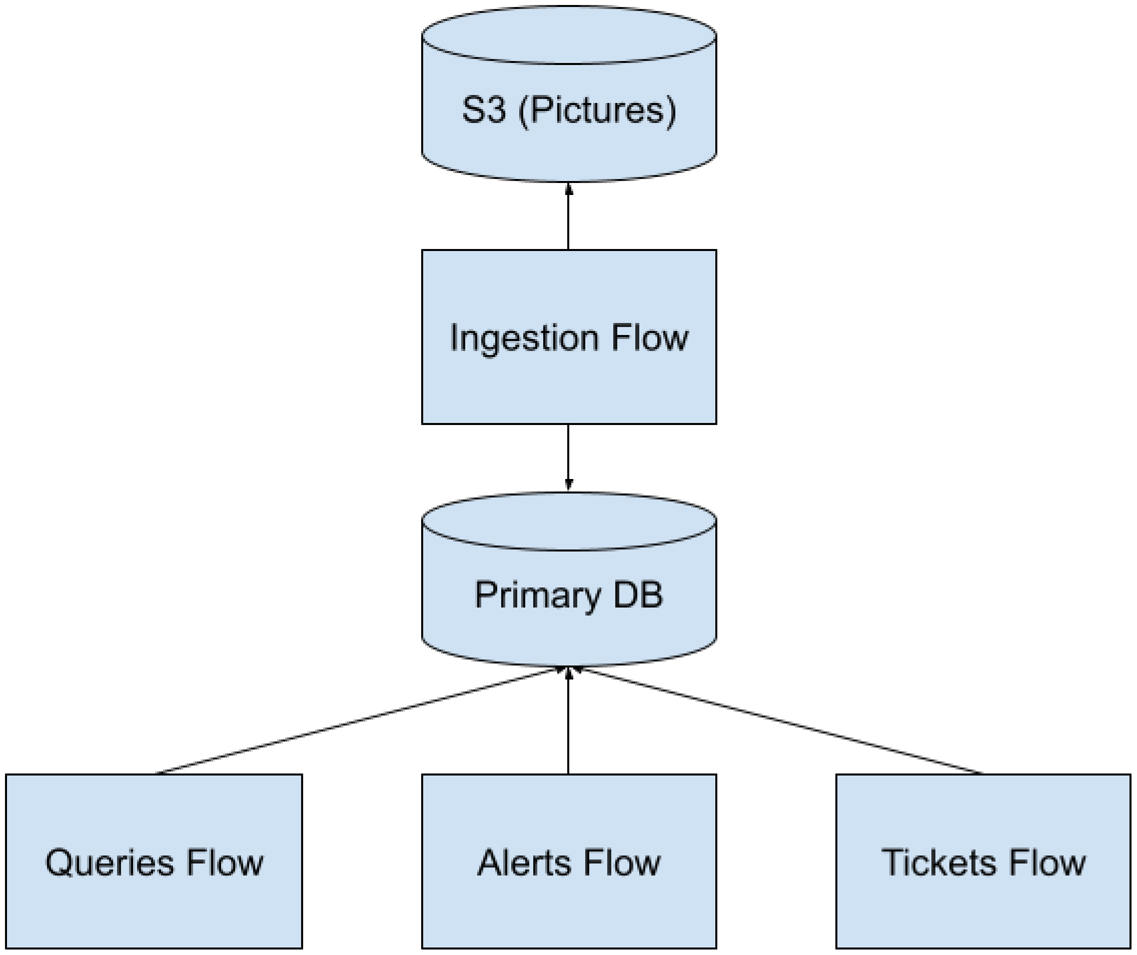
Cloud Computing Final Exercise

Overview

This design doc describes the system structure and functionality as well as the primary implementation options under specific project timeline/budget.

The following is a high level diagram of the system:



Detailed Design

Primary DB

The primary system db contains the aggregated data about cars traffic extracted from the pictures taken by the cameras.

Listed here are the use cases / queries the primary db has to support:

1. Find me the locations for car “123-32-123” on the 2nd of July, 2020.
2. Find me all the cars that passed through this particular location.
3. Give me all the red cars that were within 3km of a location.
4. How many cars went through a particular location at a given time frame.
5. [For alerts] Find me the locations where car “123-32-123” was witnessed after timestamp T.

The first observation about the set of queries is that the primary types of objects we want to store information about are cars and cameras (camera locations). All the queries can be satisfied (only satisfied, not performance level guaranteed) if for every appearance of a car in a picture we store a data item that consists of [camera_id, timestamp, car_id, picture_url] (“traffic data”) along with a set of metadata about all cars witnessed [car_id, color, model] and cameras [camera_id, lat, lng].

We would like to estimate the load on the db. We can only know the amount of writes since the read operations usage patterns are unknown to us. In the first stage (trial run) the system will perform (cameras * pictures * avg_cars_in_picture / day) / seconds in day = 150 * 12,500 * avg_cars_in_picture / 24 * 60 * 60 >= 22 writes/s assuming there’s at least one car in a picture. In the second stage, the system will produce at least 1013 writes/s (using the same calculation on 5000 cameras and 17k p/d).

We would also like to know the size of the storage required to store all the data in the system. The size of the cars and cameras metadata sets can be big but it’s almost static and in the long term is negligible in size compared to the traffic data size so we exclude it from the calculation. The size of a single data item should be approximately 60-80 bytes so we use 128 bytes as an upper bound. In the second stage the system will produce cameras * pictures / day * days in year * item size / bytes in TB = 5000 * 17,500 * 365 * 2^7 / 2^40 ~= 3.7 TB per year.

We will use DynamoDB to implement the primary db. DynamoDB is fully managed by AWS and as such it has built in support for backups and replications (availability and durability) and it scales operations to more machines automatically as the amount of the data grows. It is optimized for high read/writes rate and can store tables of any size which allows us to store the traffic data indefinitely as per the requirements.

The primary downside of using DynamoDB is that it is only supported on AWS platforms and if and when we would like to migrate our platform to be self managed we would be required to migrate the db solution first. A potential migration solution is storing all the data in a relational db in shards (yearly shards for example), in order to comply with the db size limits.

We considered the following alternatives:

1. Relational DB

    a. From the data model perspective, the data we deal with can be fit into a schema and stored in a relational db.
    b. Primary concerns are with IOPS limitations[1] and storage size limitations[2] RDS dbs have. Without having more data on the system usage patterns we can't know it will hold the load of read operations.
    c. RDS is cheaper[3]. It is also a better fit from the perspective of enabling a migration to a self managed solution.
2. Graph DB
    a. On first look it seems like a natural fit for the data model. We can use cars, cameras and reports (car seen by camera) entity types and point from a car and camera entities to a report entity that stores the timestamp of that report and a link to the picture. This approach doesn't scale well for queries that filter on timestamps because with time there will be a lot of entities of report type and it will slow down queries that filter on time ranges which are a major use case of the system.

Storing cars and cameras metadata is straightforward (car_id and camera_id are partition keys and the rest of the data is in the document body). However we can store the traffic data in two primary manners:

1. List all pictures of a specific car (car_id is the partition key, timestamp is the sort key and the camera ids are in the values).
2. List all pictures taken by a specific camera (camera_id is the partition key, timestamp is the sort key and the cars ids are in the values).

Every variant of the above is able to handle different queries with different performance levels. There's a third variant which is storing both of the data sets which will result in better performance but will cost more due to higher writes rate and using more storage space and may raise compatibility issues.

We would like to know what's the better alternative in terms of cost and performance to make a design decision regarding the database structure.

It's cheaper to fulfill queries 1 and 5 with the first approach because we need to do a single query operation using the car id and the timestamps range. If we wanted to fulfill this query with the second approach, we would have needed to use a scan operation and go over all the cameras partitions and the records on this date. On the other hand, it's cheaper to fulfill query 4 in the second approach for exactly the same reason.

Intuitively, if the data distribution is not unique, query 3 should work faster with the second approach over time assuming the amount of red cars becomes bigger than the amount of cameras for some radius (even if the radius is very big the number of cameras is very limited).

[1] https://stackoverflow.com/questions/13966368/aws-mysql-rds-vs-aws-dynamodb
[2] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html#USER_PIOPS
[3] https://dev.to/napicella/dynamodb-or-aurora-rds-should-we-always-use-dynamodb-51d5

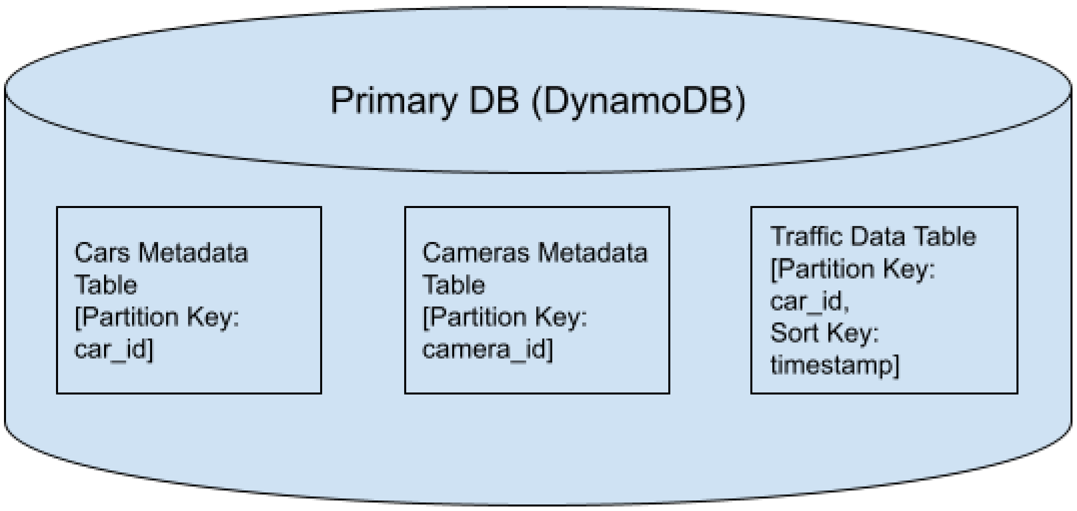
Query 2 is a classic use case for the second approach because then we only need to scan all the records attached to a single camera partition. Otherwise we would have needed to scan all the car partitions searching for records with the required camera.

Query 1 is particularly interesting because it's used heavily by the Tickets flow, while we don't have any information on the usage patterns for other queries.

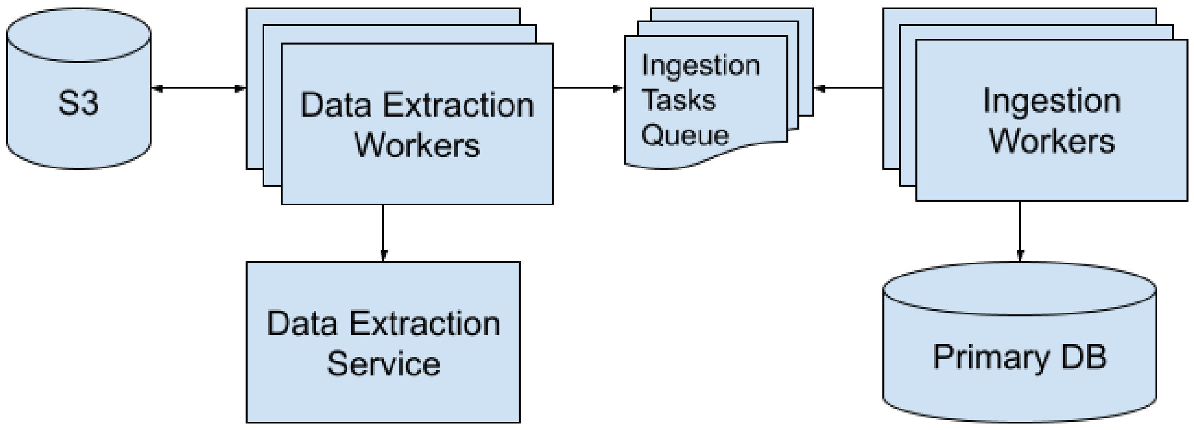
Using the first approach seems allegedly like a better option since it performs better on queries based on the limited knowledge we have. However, without having more knowledge on the database usage and data distribution patterns we can't know for sure which approach will outperform the other.

Using a combined approach w/o having more knowledge like mentioned above seems like a premature optimization that will cost more and will not necessarily produce a cost effective benefit. On top of that, since the data of each approach can be backfilled from the other, the combined approach can be utilized in the future if needed.

The final DB structure diagram:


Primary DB (DynamoDB)
Cars Metadata Table [Partition Key: car_id]
Cameras Metadata Table [Partition Key: camera_id]
Traffic Data Table [Partition Key: car_id, Sort Key: timestamp]

Ingestion Flow



The ingestion flow is responsible for extracting data from the pictures and storing it in the primary db. In the first stage, data extraction workers get the raw data from S3, perform a sync call to a data extraction service to extract the vehicle plate number, color and model and write this data to a queue of ingestion tasks. In the second stage, an ingestion worker ingests the extracted data to the primary db.

A Monolithic Approach
Another approach to this flow design is creating a single worker type that both extracts the data from the pictures and stores it in the primary db. This approach will increase productivity in the short term and spare compute and storage resources but will eventually suffer from the usual disadvantages of monolithic apps[4] s.a. high maintenance costs. Depending on the exact timelines restrictions, it may be a better idea to choose a microservices design in the first place.

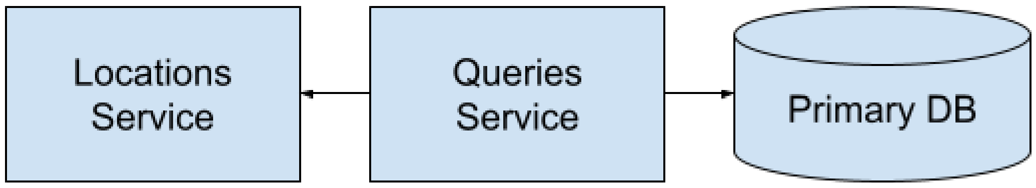
Data Extraction Service
The data extraction service is a separate unit that extracts the plate number, color and model of the car and returns this data to the system. In the early days of the system (first stage), we should use an existing service and focus on our business needs.

AWS rekognition doesn't seem to provide the full needs of our system since it's API doesn't mention it can detect object colors. We can use a 3rd party service s.a. Carnet.ai[5].

[4] https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
[5] https://carnet.ai/

Queries

We're required to provide the functionality to query the data in the system for general use. The following diagram describes the queries flow that provides this functionality.



Queries service is an endpoint that we manage that provides an API/UI that allows operators to query the system. It gets the data directly from the primary db in the manner described in many details in the Primary DB section.

The location service is an external service s.a. AWS Locations Service that provides us the ability to do things like:

- Present a map on the UI and draw a custom polygon / bounding box on the map.
- Map location names / geocodes to geometries.

When a query s.a. "Find me all the cars that passed through this particular location" is executed, the queries service gets the location polygon from the locations service and then generates the set of required cameras by matching their location to the location polygon.

Alerts

There are multiple alternatives for supporting this use case:

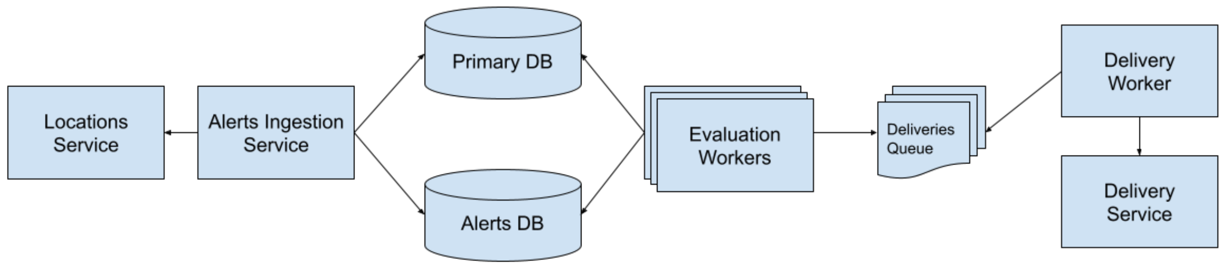
- A push based architecture - every time a record is changed or about to be changed we check the live alerts and evaluate their alerting conditions with the processed change.
    - Using DynamoDB Streams and Lambda Triggers[6].
    - During ingestion by an ingestion worker (see "A Monolithic Approach").
- A pull based architecture which consists of an evaluation component that polls the primary db at a fixed rate and evaluates the changes in the data against the alerts conditions.

The main problem with a push based architecture is that potentially a lot of compute resources are used to evaluate records that should not trigger an alert because all of the record changes are being evaluated. Therefore, in the long term, we prefer a pull based architecture.

[6] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.html

The primary benefit of a push based architecture is that it's simpler to implement. We may prefer a push architecture if there's a pressing deadline for this flow of the system to be operational.

In this section we only focus on a pull based architecture.



The alerts are stored in the Alerts DB which can be implemented using a relational db with the following tables:

- UserAlerts[UserAlertId, CarId, CarModel, CarColor, Location, DeliveryConfig]
- Alerts[UserAlertId, AlertId, CarId, CarModel, CarColor, CameraId]

UserAlerts stores the alerts data as it was provided by the user in order to allow users to view and modify existing alerts. The semantics of a row in Alerts is that as long as it exists for the condition it represents given by legal combinations of CarId, CarModel, CarColor, CameraId (a row with CameraId only is an example for an illegal row) there should be an alert fired if the conditions are met.

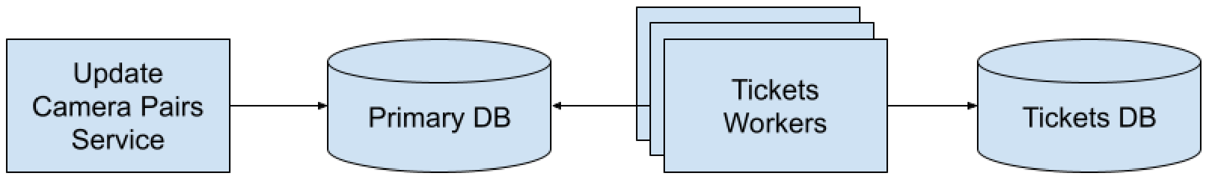
The alerts ingestion service provides some API/UI for the system users to create alerts and translates them to the structured form supported by the AlertsDB. For example, for alerts like "Raise alert if a blue Toyota is seen in the South Region for the next 6 hours" the ingestion service is looking up for the cameras that are present in the South Region via the locations service and creates appropriate alerts in Alerts DB for every camera.

The evaluation workers job is to read the alerts from Alerts DB and evaluate their conditions on the most recent data that was ingested since the last pull cycle. Every alert condition can be checked via a single scan operation. Depending on the exact load on the system (number of alerts), more evaluation workers can be instantiated and handle disjoint sets of alerts. When an alert condition is matched, the evaluation worker files a delivery request to the deliveries queue and a separate component handles the delivery.

The evaluation worker can be broken to two worker types and a queue - one worker that reads alerts and files evaluation tasks and the other that performs the evaluation task for a single alert. This seems unnecessary at this point w/o knowing the size estimations of the AlertsDB.

For the same reasons discussed in the "Data Extraction Service" section we would want to use some existing delivery service s.a. AWS SES for mails and AWS SNS for SMS at the trial period.

Tickets Flow



We're required to develop a flow that automatically files tickets to vehicle owners that performed a speed limit violation (filing a ticket in the flow is represented as writing a ticket row to the Tickets DB, the action of processing the ticket further is out of this design's scope). The speed limit violation is determined by applying a speed-distance-time calculation based on a car's appearance in pictures taken by two subsequent cameras, their timestamps, the distance between the cameras and the speed limit in the area between the cameras.

Operators will be able to create/change such camera pairs via the update camera pairs service which provides an API that supports that. The service will write the pairs to the primary db cameras metadata table as a list of subsequent cameras in the document of each camera. Every such item will contain the distance between the cameras and the speed limit in this part of the road.

The tickets flow is primarily based on ticket workers which are cron jobs that perform batch processing on accumulated traffic data. Since there's no requirement to file tickets in real time, we can process the data once a day on a fixed schedule on the entire data set that was accumulated the day before. For example, if today is June 24th, we can calculate all the tickets that should be given for traffic rules violation events that happened on June 23rd.

The tickets job looks up for all the traffic data accumulated for a set of cars (some partitions) in the last day and runs the following logic:

```
cams_graph = get_cameras_graph()
for car in cars:
    reports = car.get_reports_from_yesterday()
    for (prev, curr) in reports:
        if not cams_graph.are_subsequent_cameras(prev.camera, curr.camera):
            continue
        distance = cams_graph.get_distance(prev.camera, curr.camera)
        speed_limit = cams_graph.get_speed_limit(prev.camera, curr.camera)
                 if  above_speed_limit(prev.timestamp,  curr.timestamp,  distance,
speed_limit):
            file_ticket()
```

`get_cameras_graph` is run once per job and it builds in the job memory a data structure that holds the cameras graph. It should not be a painful task in terms of resource usage due to the relatively small number of cameras in the system.

Caching

There are no plans to support caching layers atm. When we have more data on usage patterns (reads) and potential bottlenecks in the system, caching layers can be added on demand.

Launch Plan

External Cloud Services

We used multiple external services throughout the system. We define external service as services that don't run logic provided by us and don't store data from our system and that it'll take a significant amount of time and money to develop a replacement inhouse service with the same level of quality/accuracy.

Such services are the cars recognition service used in the ingestion flow that is provided by Carnet.ai and the Locations and Mailing/SMS services provided by AWS.

When the system becomes operational and more mature it might be beneficial to use existing data to benchmark with other solutions to improve precision and costs. We can even consider developing inhouse solutions to spare the costs of using external services.

Let's use the cars recognition service as an example and explore it deeper.

In the short term we produce 150 * 12,500 = 1.8m pictures a day. Carnet.ai don't provide cost estimates for such loads. However, if 100k requests per day costs 325 EUR/month then processing 1.8m pictures a day should cost 18 * 325 * 0.9 (estimated discount factor) = 5265 EUR/month which is a half of an engineer's salary (Assuming engineers salary is 10k EUR/month).

However, in the long term we will produce 5,000 * 17,500 = 87.5m pictures per day the cost will increase to 256k EUR/month which is more than the salary of 20 engineers (for simplicity we assume engineers salaries are the only expense of the company).

Assuming that building a cars recognition service that reaches a precision rate that is close enough to the Carnet.ai precision rate (97%) takes 5 engineering years, we may hire a team of 5-6 engineers (Assuming for simplicity engineers are the only type of workers required for completing such a project) that will complete building the service within a year. This will cost the company a year of salaries of engineers (720k EUR) but will save the company 256k EUR/month from the second year, which means the ROI will happen after 3 months only.

Obviously, all the assumptions we made in the calculation (Only engineers are required and they are all paid equally and that's the only company expense) are incorrect. When taking the additional considerations into account the cost grows linearly. This means the ROI will be delayed, but will still happen at some near point in the future which makes this investment worth considering.

Serverless vs Self Managed

Except for the external services discussed above which were defined as services that don't run our logic and store our data the system consists of services, worker nodes, queues and databases that do run our logic and store our data.

It's our responsibility to develop, maintain and pay for these components. Throughout the design we did not mention the implementation details of every component but rather focused on the function it fulfills. This is because there are multiple options to consider, each has different benefits in terms of development velocity and costs.

The primary three options are using serverless solutions (DynamoDB, RDS, SQS, Lambda, API Gateway), using self managed cloud hosted solutions (EC2) and migrating from the cloud (this option is discussed shortly in the next section).

Choosing between a serverless implementation and a self managed cloud hosted solution is a trade off between development velocity and costs. While a serverless implementation can usually guarantee a shorter time to market period, it will cost more in the long term for certain load patterns. A serverless solution is usually more cost effective if our machines are not used in full capacity, which is not our case.

It does not necessarily mean a self managed solution is always better. Different flows have different loads and some of them may be cheaper to maintain on serverless platforms. Also, different flows and even different components on the same flow may be implemented in different approaches, depending on the load on them.

To make this point clearer, we'll explore in detail the option to implement the workers in the ingestion flow (either the data extraction or the ingestion workers fit this example) in the two approaches. We can do this because we have information about the load patterns in this part of the system in advance (at least 22 writes/s in the trial period and 1013 writes/s later).

All the workers in the ingestion flow may be implemented as Lambda workers that start processing based on events (new picture added to S3, via S3 triggers) or as EC2 instances that run the worker logic and are auto scaled per the system load.

In the trial period, if we use a serverless solution that supports 22 qps, where each request lasts 250ms and requires 256Mb of memory we will pay a total of 65$ per month. If we use an EC2 solution we would need a single instance with 22 / 4 = 6 cores and 2GB (upper bound) of memory which will cost 96.81$ per month.

In the longer term, if we use a serverless solution that supports 1013 qps, each request lasts 250ms and requires 256Mb of memory we will pay a total of 3300$ per month. If we use an EC2 solution we would need 22 instances with (1013 / 4) / 22 = 12 cores and 4GB (upper bound) of memory which will cost 2900$ per month **if the reservation term is for 3 years** (This can also get cheaper if paid upfront but then we would need another service that predicts inflation rates).

Using a serverless solution in the trial period for the ingestion flow components may make sense if the deadline for having a working system is very pressing, however it may cost more to migrate to an EC2 based solution afterwards than the total price delta between serverless and EC2.

This conclusion may not apply to other flows in the system which have different loads. If we want to boil down all the above information into a single general conclusion then it would be starting with a serverless solution, and when enough information is available about load patterns in the system, consider replacing the serverless to a self managed solution if the ROI is in the near future.

Migrate From Cloud Services

In the very long term, when the system is mature/maintenance mode it may make sense to migrate from cloud services (not the external services but rather all the components managed by us). Calculating the ROI period for this option is out of the scope of this design.