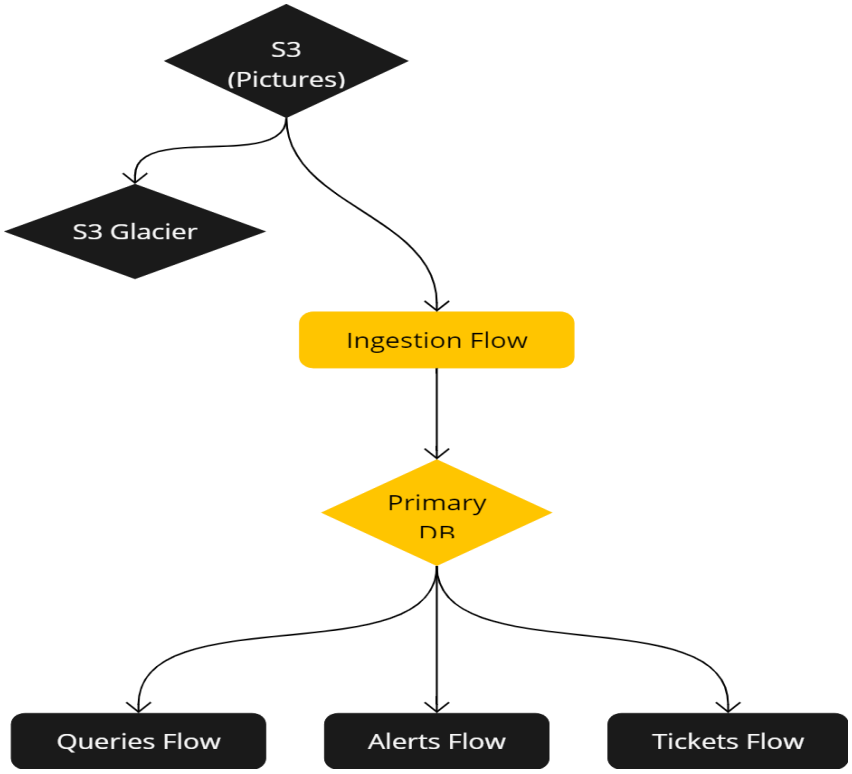
Cloud Computing Final Exercise

Overview

This design document outlines the system architecture and functionality, and key implementation choices for a cloud-based traffic control and ticketing system, within the project's timeline and budget constraints.

high level diagram of the system:



Detailed Design

Primary Database

The primary system database contains aggregated data about car traffic extracted from pictures taken by the cameras. Here are the use cases/queries the primary database needs to support:

1. Find the locations for car "123-32-123" on July 2, 2020.
2. Find all the cars that passed through a specific location.
3. Provide all red cars within 3 km of a location.
4. Count the number of cars passing through a particular location within a given time frame.
5. [For alerts] Find the locations where car "123-32-123" was witnessed after timestamp T.

The primary types of objects to store information about are cars and cameras (camera locations). All queries can be satisfied 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 metadata about all cars witnessed [car_id, color, model] and cameras [camera_id, lat, lng].

Estimating Database Load

Trial Run (First Stage):

- 150 cameras
- 12,500 pictures per camera per day
- 22 writes/second

Full Deployment (Second Stage):

- 5,000 cameras
- 17,500 pictures per camera per day
- 1,013 writes/second

Storage Size Estimation

- Data item size: ~60-80 bytes (using 128 bytes as an upper bound)
- Full deployment storage requirement: ~3.7 TB per year

Database Choice

We will use DynamoDB to implement the primary database due to its benefits:

- Fully managed by AWS with built-in support for backups and replications.
- Scales automatically as data grows.
- Optimized for high read/write rates and can store tables of any size.

Downside:

- Only supported on AWS platforms, making migration to a self-managed solution more challenging.

Considered Alternatives

1. **Relational Database (RDS):**
    - Data can fit into a schema and stored in a relational database.
    - Concerns with IOPS limitations and storage size limitations.
    - Cheaper and better for potential migration to a self-managed solution.
2. **Graph Database:**
    - Natural fit for the data model.
    - Does not scale well for queries filtering on timestamps.

Data Storage Approaches

1. **By Car:**
    - Partition key: car_id
    - Sort key: timestamp
    - Values: camera_id
2. **By Camera:**
    - Partition key: camera_id
    - Sort key: timestamp
    - Values: car_id

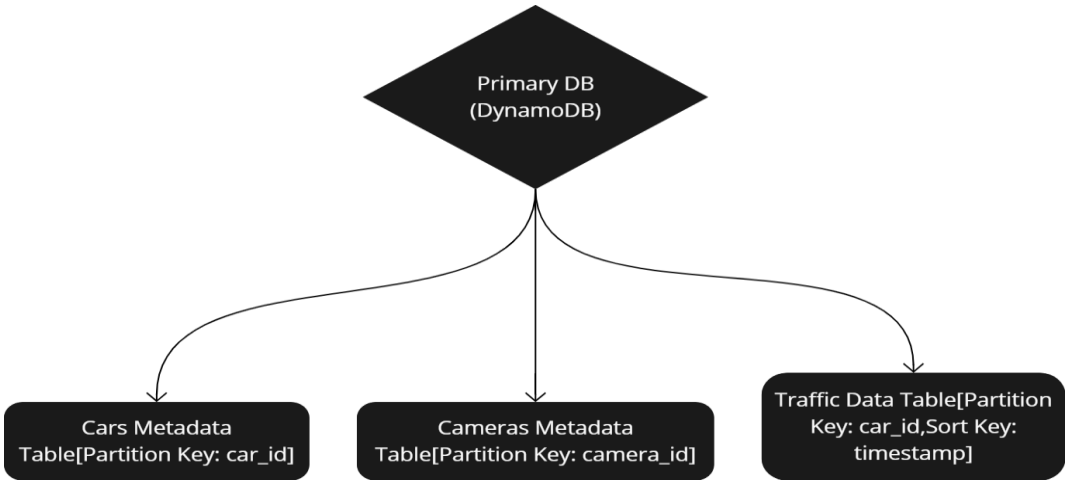
Each approach handles different queries with varying performance levels. A combined approach may be utilized in the future if needed.

Using the first approach appears to be the better option based on our limited knowledge, as it performs better on the queries we currently have. However, without more information on the database usage and data distribution patterns, we cannot be certain which approach will ultimately be more effective.

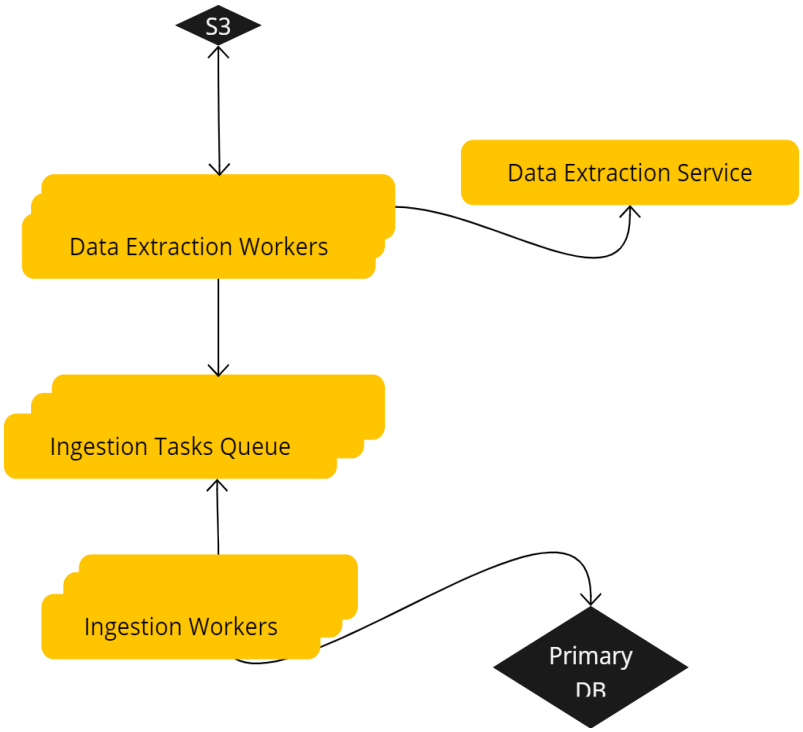
Implementing a combined approach without additional knowledge would be a premature optimization, increasing costs without necessarily providing a cost-effective benefit. Additionally, since data from each approach can be backfilled from the other, a combined approach can be adopted in the future if needed.

Final Decision

Using the first approach (by car) seems better based on limited knowledge. However, without more information on database usage and data distribution patterns, it's uncertain which approach will outperform the other. A combined approach may be used in the future if needed, and the data from each approach can be backfilled if necessary.



Ingestion Flow



The ingestion flow is responsible for extracting data from pictures and storing it in the primary database. Here's an outline of the two-stage process and alternative approaches:

Stages

1. **First Stage**: Data Extraction
    - **Workers**: Data extraction workers retrieve raw data from S3.
    - **Service Call**: Perform a synchronous call to a data extraction service to extract the vehicle plate number, color, and model.
    - **Queue**: Write the extracted data to an ingestion tasks queue.
2. **Second Stage**: Data Ingestion
    - **Workers**: Ingestion workers retrieve data from the ingestion tasks queue.
    - **Database**: Ingest the extracted data into the primary database.

Monolithic Approach

An alternative approach involves creating a single worker type that both extracts data from pictures and stores it in the primary database.

Pros:

- Increased productivity in the short term.
- Reduced compute and storage resource usage.

Cons:

- High maintenance costs typical of monolithic applications.
- Potential long-term scalability and flexibility issues.

Depending on the exact timeline constraints, adopting a microservices architecture from the outset may be a better choice.

Data Extraction Service

The data extraction service is a separate unit that extracts the vehicle plate number, color, and model, returning this data to the system.

First Stage (Early Days)

- **Service Utilization**: Use an existing third-party service to focus on core business needs.
- **AWS Rekognition**: AWS Rekognition doesn't fully meet the system's needs as its API doesn't detect object colors.
- **Alternative Service**: Consider using a third-party service like Carnet.ai.

Decision Factors

- **Timeline**: If the project timeline is tight, a monolithic approach may be beneficial initially but should be refactored to micro services for long-term sustainability.
- **Scalability**: Micro services architecture provides better scalability and flexibility, suitable for long-term growth.
- **Maintenance**: Micro services reduce maintenance complexity compared to monolithic architectures.

Cost Considerations

- **Monolithic Approach**: Lower initial costs but higher long-term maintenance costs.
- **Micro services Approach**: Higher initial costs due to complexity but lower long-term maintenance costs.

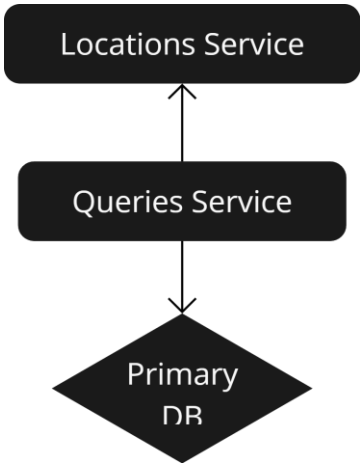
Final Recommendation

While the monolithic approach can provide short-term benefits, a micro services design is recommended for long-term sustainability, scalability, and maintenance efficiency. Using a third-party data extraction service like Carnet.ai in the initial stages allows focusing on core business requirements without investing heavily in developing custom extraction solutions.

By adopting a structured and phased approach, the ingestion flow can effectively manage data extraction and ingestion, ensuring the system remains scalable and maintainable as it grows.

Queries

We need to ensure the system can perform general data queries. The following diagram illustrates the flow of queries that enables this functionality.



Queries Service

The Queries Service is an endpoint we manage, providing an API/UI for operators to query the system. It retrieves data directly from the primary database, as detailed extensively in the Primary DB section.

Location Service

The Location Service is an external service, such as AWS Locations Service, which enables us to perform various tasks, including:

- Displaying a map on the UI and drawing custom polygons or bounding boxes.
- Mapping location names or geocodes to geometries.

When executing a query like “Find all the cars that passed through this particular location,” the Queries Service retrieves the location polygon from the Location Service and then determines the relevant cameras by matching their locations to the location polygon.

Alerts

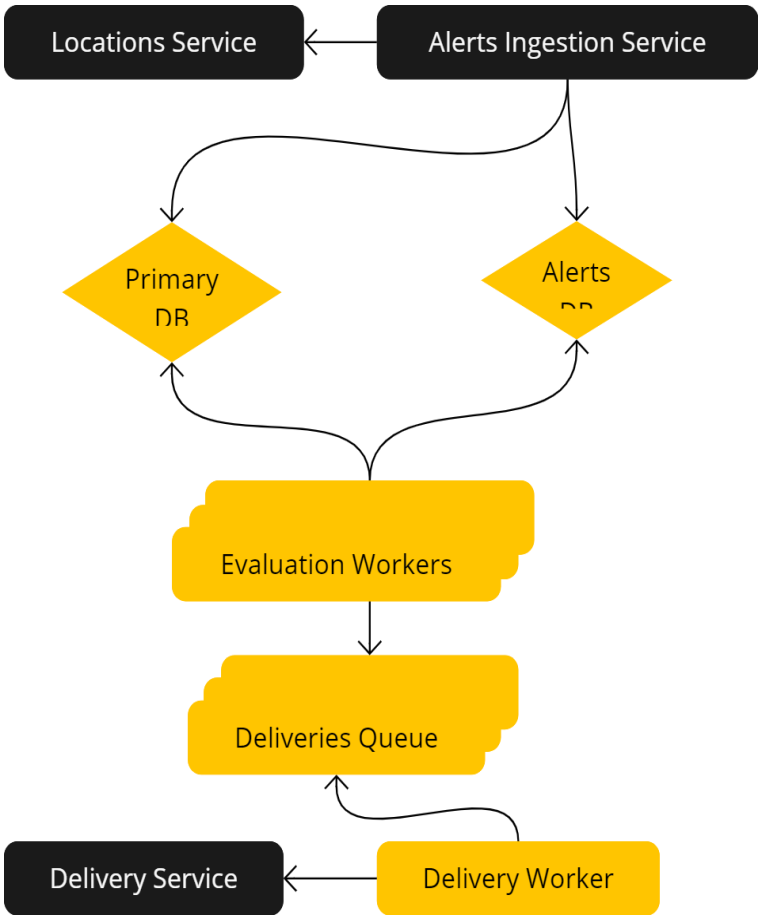
There are multiple alternatives for supporting this use case:

- **Push-based architecture:** Every time a record is changed or about to be changed, we check the live alerts and evaluate their alert conditions with the processed change.
    - Using DynamoDB Streams and Lambda Triggers.
    - During ingestion by an ingestion worker (see “A Monolithic Approach”).
- **Pull-based architecture:** This approach involves an evaluation component that polls the primary database at a fixed rate and evaluates the changes in the data against the alert conditions.

The main issue with a push-based architecture is that it potentially consumes a significant amount of compute resources to evaluate records that do not trigger an alert, as all record changes are being evaluated. Therefore, in the long term, we prefer a pull-based architecture.

The primary benefit of a push-based architecture is its simplicity in implementation. If there’s a pressing deadline for this system flow to be operational, a push architecture might be preferred.

In this section, we focus solely on a pull-based architecture.



The alerts are stored in the Alerts Database, which can be implemented using a relational database with the following tables:

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

UserAlerts Table

The **UserAlerts** table stores the alert data as provided by the user, allowing users to view and modify existing alerts. Each row in the **Alerts** table represents a valid combination of CarId, CarModel, CarColor, and CameraId. For example, a row with only CameraId is invalid. An alert should be fired if the conditions represented by a row are met.

Alerts Ingestion Service

The Alerts Ingestion Service provides an API/UI for system users to create alerts, translating them into the structured form supported by the **AlertsDB**. For instance, for an alert like “Raise an alert if a blue Toyota is seen in the South Region for the next 6 hours,” the ingestion service looks up cameras in the South Region via the Locations Service and creates corresponding alerts in the **AlertsDB** for each camera.

Evaluation Workers

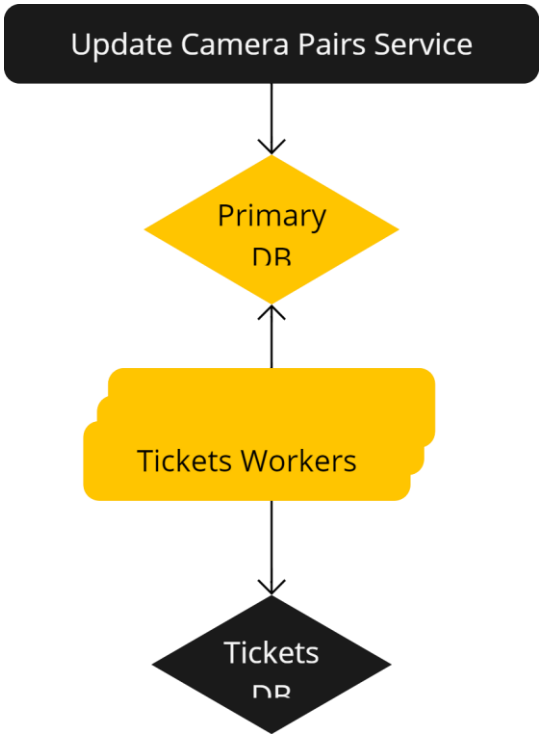
The job of evaluation workers is to read alerts from the **AlertsDB** and evaluate their conditions based on the most recent data ingested since the last pull cycle. Each alert condition can be checked with a single scan operation. Depending on the system's load (number of alerts), more evaluation workers can be instantiated to handle separate sets of alerts. When an alert condition is met, the evaluation worker files a delivery request to the deliveries queue, and a separate component manages the delivery.

Worker Types

The evaluation worker can be divided into two types and a queue: one worker reads alerts and files evaluation tasks, while the other performs the evaluation task for a single alert. However, this division might be unnecessary without size estimations of the **AlertsDB**.

For the same reasons discussed in the “Data Extraction Service” section, we should consider using existing solutions. AWS SES for mails and AWS SNS for SMS

Tickets Flow



Automatic Ticket Filing for Speed Limit Violations

We need to develop a flow that automatically files tickets for vehicle owners who exceed speed limits. Filing a ticket in this flow means writing a ticket row to the **Tickets DB**; further processing of the ticket is outside the scope of this design. Speed limit violations are determined by calculating speed using the distance-time formula 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.

Operators can create or update such camera pairs via the **Update Camera Pairs Service**, which provides an API for this purpose. The service writes the pairs to the primary database's **Cameras Metadata** table as a list of subsequent cameras in each camera's document. Each item will include the distance between the cameras and the speed limit in that section of the road.

The Tickets Flow

The tickets flow relies 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 for the entire dataset accumulated the previous day. For example, if today is June 24th, we can calculate all the tickets for traffic rules violations that occurred on June 23rd.

Tickets Job Logic

The tickets job processes all the traffic data accumulated for a set of cars (some partitions) over the last day. It follows this logic:

1. **Initialize Cameras Graph**: The job starts by building an in-memory data structure representing the cameras graph, which contains information about the cameras' locations, distances between them, and speed limits for the respective road segments.
2. **Process Each Car's Reports**: For each car, the job retrieves traffic reports from the previous day.
3. **Evaluate Camera Pairs**: For each pair of subsequent reports (previous and current) for a car:
    - **Check Subsequent Cameras**: Determine if the cameras in the reports are subsequent.
    - **Get Distance and Speed Limit**: Retrieve the distance between the cameras and the speed limit for the road segment.
    - **Calculate Speed**: Apply a speed-distance-time calculation to check if the car exceeded the speed limit.
4. **File Tickets**: If a speed limit violation is detected, a ticket is filed in the **Tickets DB**.

The **"Initialize Cameras Graph"** function is run once per job, constructing an in-memory data structure that holds the cameras graph. This task is not resource-intensive due to the relatively small number of cameras in the system.

Caching

Currently, there are no plans to implement caching layers. Once we have more data on usage patterns (reads) and potential bottlenecks in the system, we can introduce caching layers as needed. Implementing caching layers can also help reduce costs by minimizing the load on our primary database and reducing the frequency of expensive read operations.

Launch Plan

External Cloud Services

We utilized multiple external services throughout the system. An external service is defined as one that does not run our provided logic, does not store data from our system, and would take a significant amount of time and money to develop an equivalent in-house service with the same level of quality and accuracy.

Examples of such services include the car recognition service used in the ingestion flow provided by Carnet.ai, as well as the Location and Mailing/SMS services provided by AWS.

As the system becomes operational and matures, it might be beneficial to benchmark our existing data against other solutions to improve precision and reduce costs. We can also consider developing in-house solutions to save on the costs associated with external services.

Example: Car Recognition Service

In the short term, we will produce 1.8 million pictures per day (150 cameras * 12,500 pictures per camera). Carnet.ai does not provide cost estimates for such loads, but if 100,000 requests per day cost 325 EUR/month, then processing 1.8 million pictures per day would cost approximately 5,265 EUR/month (calculated as 18 * 325 * 0.9, assuming a 10% discount), which is about half of an engineer's monthly salary (assuming an engineer's salary is 10,000 EUR/month).

In the long term, we will produce 87.5 million pictures per day (5,000 cameras * 17,500 pictures per camera), resulting in a cost of 256,000 EUR/month, which exceeds the salary of 20 engineers (assuming engineers' salaries are the only company expense).

Assuming that developing a car recognition service with precision close to Carnet.ai's (97%) takes five engineering years, we could hire a team of 5-6 engineers to complete the service within a year. This would cost the company 720,000 EUR (a year's worth of engineer salaries), but it would save 256,000 EUR/month starting from the second year, achieving ROI after only three months.

Of course, the assumptions made in this calculation (engineers are the only workers required, all are paid equally, and salaries are the only company expense) are simplistic. Considering additional factors, the costs would increase linearly, delaying the ROI but still making the investment worthwhile in the near future.

Serverless vs. Self-Managed

In addition to the external services discussed earlier, the system consists of services, worker nodes, queues, and databases that run our logic and store our data. It is our responsibility to develop, maintain, and pay for these components. While the design focuses on the functions of each component rather than implementation details, it is important to consider the various options available, each with different benefits in terms of development velocity and costs.

The primary three options are:

1. **Serverless Solutions**: (DynamoDB, RDS, SQS, Lambda, API Gateway)
2. **Self-Managed Cloud-Hosted Solutions**: (EC2)
3. **Migrating from the Cloud**: (Discussed in the next section)

Choosing between a serverless implementation and a self-managed cloud-hosted solution involves a trade-off between development velocity and costs. A serverless implementation typically offers a shorter time to market but may cost more in the long term for certain load patterns. Serverless solutions are generally more cost-effective if our machines are not used at full capacity, which is not our case.

However, this does not necessarily mean a self-managed solution is always better. Different flows have different loads, and some may be cheaper to maintain on serverless platforms. Additionally, different flows and even components within the same flow may benefit from different approaches depending on their load.

To illustrate this, let's explore the option to implement the workers in the ingestion flow (data extraction or ingestion workers) using both approaches. We have advance information about the load patterns in this part of the system (at least 22 writes per second during the trial period and 1013 writes per second later).

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

Trial Period Costs

- **Serverless Solution**: Supports 22 queries per second (qps), each request lasting 250ms and requiring 256MB of memory. This will cost a total of $65 per month.
- **EC2 Solution**: Requires a single instance with 6 cores and 2GB of memory, costing $96.81 per month.

Long-Term Costs

- **Serverless Solution**: Supports 1013 qps, each request lasting 250ms and requiring 256MB of memory. This will cost a total of $3,300 per month.
- **EC2 Solution**: Requires 22 instances with 12 cores and 4GB of memory each, costing $2,900 per month if reserved for 3 years (potentially cheaper if paid upfront, though this requires inflation rate predictions).

Using a serverless solution during the trial period for the ingestion flow components may make sense if there is a pressing deadline for having a working system. However, migrating to an EC2-based solution afterward may cost more than the total price difference between serverless and EC2.

This conclusion might not apply to other flows in the system with different loads. A general approach would be to start with a serverless solution and, once enough information is available about the system's load patterns, consider replacing the serverless solution with a self-managed solution if the ROI is achievable in the near future.

Migrate From Cloud Services

In the very long term, when the system is mature and in maintenance mode, it may be sensible to consider migrating from cloud services (excluding external services, but including all components managed by us). Using infrastructure as code tools like Pulumi can facilitate this migration by allowing us to manage and transition our infrastructure efficiently. Calculating the ROI period for this option is beyond the scope of this design.

main costs

S3

GB = 595 imeges

Scenario 1:

150 cameras, 12,500 pictures per camera per day:

Total monthly storage = 92.4 TB, Total monthly cost $1,177.60 + $954.37 = $2,131.97/month

Scenario 2:

5,000 cameras, 17,500 pictures per camera per day:

Total monthly storage = 4,308 TB, Total monthly cost

$1,177.60 + $10,137.60 + $81,922.56 = $93,237.76/month

S3 Glacier

0.0036 per GB

Scenario 1: $339.46/month

Scenario 2: $15,884.42/month

DB

Scenario 1: 22 WCUs $27.50/month + 110 RCUs $27.50/month = $55/month

Scenario 2: 1,013WCU $1,266.25/month + 5,065 RCUs $1,266.25/month = $2,532.50/month

Rekognition

Scenario 1: $1,000+$3,200+$18,000+$5,312.50=$27,512/month

Scenario 2: $1,000+$3,200+$18,000+$647,500=$669,700/month

AWS SES 10,000 emails/month×0.10/1,000=$1.00/month

AWS SNS (SMS) 5,000 SMS/month×0.0065=$32.50/month