IoT Traffic Data Pipeline on AWS

Project Report - Group 12

Authors: Irfan Ullah, Sajid , Muhammad Zain ul Abideen

Github-Repo: https://github.com/bronglil/Aws-IoT-Data-Summarizer

Introduction:

The project aim is to create a project that will be suitable and reliable for large-scale IoT environments, in industry thousands of devices continuously exchange data over networks, generating massive volumes of traffic logs. Traditional batch processing introduces delays and leads to high storage costs when retaining all raw logs indefinitely.

This project addresses these challenges by designing an event-driven IoT traffic data pipeline on AWS. The system ingests traffic logs, processes them incrementally, and maintains only essential consolidated metrics such as daily totals, averages, and standard deviations for each source-destination IP pair.

Architecture and System Flow:

The pipeline follows an event-driven, loosely coupled design using AWS managed services to minimize operational complexity while ensuring scalability.

Core Components:
EC2 Upload Client: It is used to upload or place raw IoT traffic log files into Amazon S3 incoming directory.

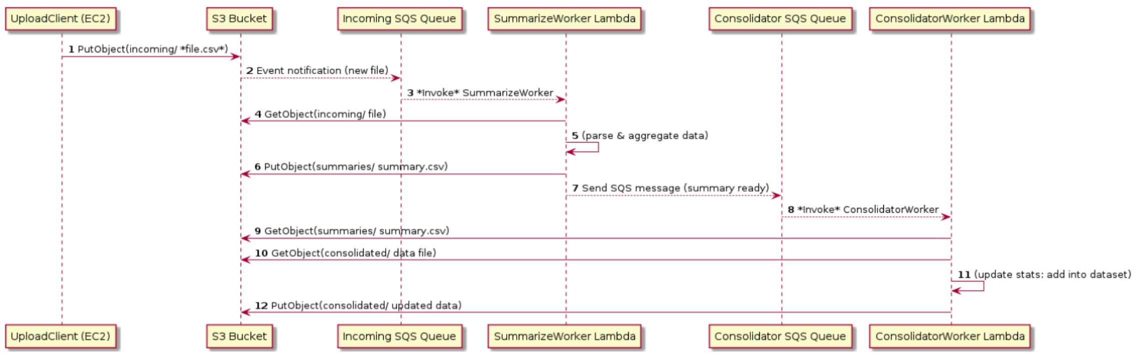
Amazon S3: Central storage with organized directories (incoming, summaries, consolidated, exports) that triggers processing events and also we used it to store jar files for consolidated and summarize so that it will handle repeated things.

Amazon SQS Queues: we use two SQS queues to manage how data moves between stages. The first queue receives messages from S3 whenever a new IoT CSV file is uploaded. This triggers the SummarizeWorker. After summarization is complete, a message is sent to the second queue, which then triggers the ConsolidatorWorker. This ensures that each processing step runs smoothly, even if one worker is temporarily unavailable.

AWS Lambda Functions: The SummarizeWorker Lambda reads raw IoT logs from the uploaded file, groups them by source IP, destination IP, and date, and writes a summary file to the S3 summaries/ folder. The ConsolidatorWorker then reads these summaries, updates the cumulative statistics like total durations and packet counts, and writes everything into a single, ongoing file in the consolidated/ folder

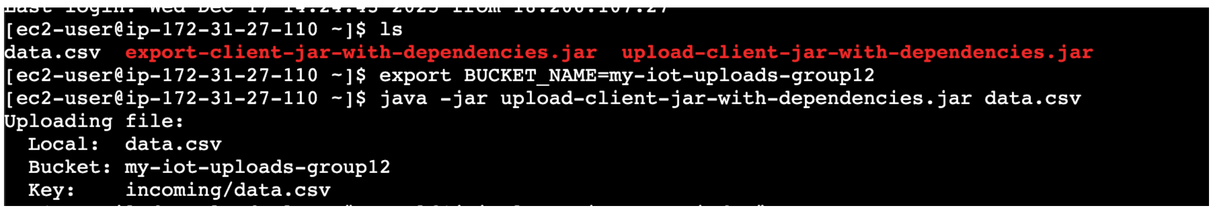
EC2 Export Client: Our Export Client runs on EC2 and fetches the latest consolidated traffic data from the S3 consolidated/ folder. It filters the data based on a specific source and destination IP pair, then saves the filtered report as a new CSV file in the exports/ folder in S3. This makes it easy to extract insights or generate reports for particular device communication patterns

System Flow:


UploadClient (EC2)
S3 Bucket
Incoming SQS Queue
SummarizeWorker Lambda
Consolidator SQS Queue
ConsolidatorWorker Lambda
1 PutObject(incoming/ *file.csv*)
2 Event notification (new file)
3 *Invoke* SummarizeWorker
4 GetObject(incoming/ file)
5 (parse & aggregate data)
6 PutObject(summaries/ summary.csv)
7 Send SQS message (summary ready)
8 *Invoke* ConsolidatorWorker
9 GetObject(summaries/ summary.csv)
10 GetObject(consolidated/ data file)
11 (update stats: add into dataset)
12 PutObject(consolidated/ updated data)
UploadClient (EC2)
S3 Bucket
Incoming SQS Queue
SummarizeWorker Lambda
Consolidator SQS Queue
ConsolidatorWorker Lambda

The IoT data processing pipeline starts when the Upload Client, a Java program running on an Amazon EC2 instance, uploads a CSV file containing IoT traffic logs to an S3 bucket named my-iot-uploads-group12. Each CSV file is placed in the incoming/ directory within the bucket. These logs contain communication records between devices, including fields like source IP, destination IP, date, flow duration, and packet counts. Once uploaded, the file is durably stored in S3 and remains available for processing.

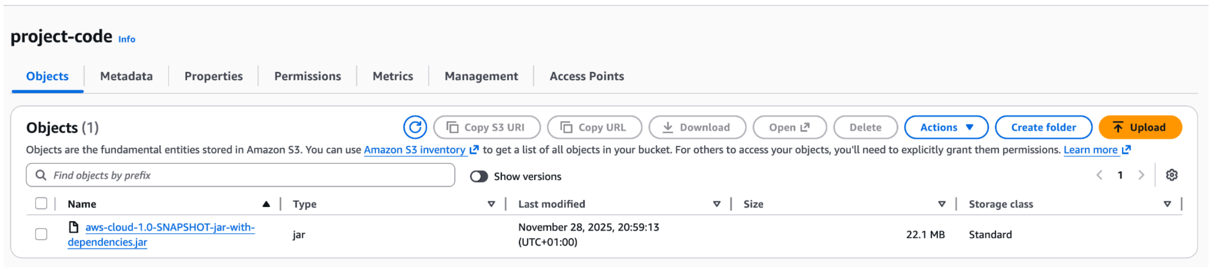
This upload is not performed manually. The EC2 instance runs a JAR executable file upload-client-jar-with-dependencies.jar, which was first built locally using Maven and then transferred to the EC2 instance using SFTP. The bucket name is passed as an environment variable, and the client uses the AWS SDK to upload files directly to the incoming/ prefix.



```
[ec2-user@ip-172-31-27-110 ~]$ ls
data.csv  export-client-jar-with-dependencies.jar  upload-client-jar-with-dependencies.jar
[ec2-user@ip-172-31-27-110 ~]$ export BUCKET_NAME=my-iot-uploads-group12
[ec2-user@ip-172-31-27-110 ~]$ java -jar upload-client-jar-with-dependencies.jar data.csv
Uploading file:
  Local:   data.csv
  Bucket: my-iot-uploads-group12
  Key:     incoming/data.csv
```

Once the file reaches S3, an event notification is triggered. Amazon S3 automatically sends a message to the first Amazon SQS queue named incoming-file-queue. This message includes metadata about the uploaded object (such as bucket name and key), which allows downstream processing to begin.

The next stage of the pipeline is handled by the SummarizeWorker, an AWS Lambda function. However, instead of uploading code directly through the Lambda console, the project JAR file aws-cloud-1.0-SNAPSHOT-jar-with-dependencies.jar was first uploaded to a dedicated S3 code bucket folder. Then, within the Lambda configuration, this S3 path url was referenced as the source for the function code. This setup ensures the Lambda can fetch the most recent version of the code during deployment without manual re-upload.



Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

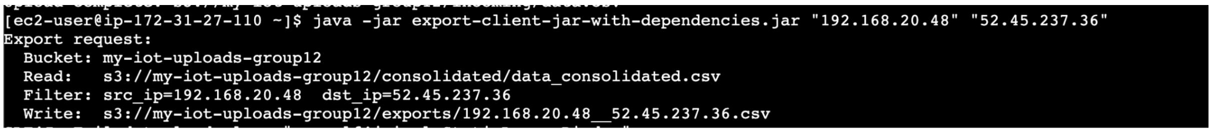
Name	Type	Last modified	Size	Storage class
aws-cloud-1.0-SNAPSHOT-jar-with-dependencies.jar	jar	November 28, 2025, 20:59:13 (UTC+01:00)	22.1 MB	Standard

Triggered by messages from the incoming-file-queue, the SummarizeWorker fetches the uploaded CSV file from the S3 incoming/ directory. It processes the file by grouping rows by source IP, destination IP, and date. For each group, it computes the total flow duration and total forward packets. The result is a summarized version of the data, which significantly reduces the file size while retaining key metrics.

Once summarization is complete, the output is written back to S3 in the summaries/ directory as a new CSV file. The function then sends a new message to the second SQS queue, consolidator-queue, signaling that a new summary file is ready to be consolidated.

ConsolidatorWorker also runs on code loaded from the same S3-hosted JAR. When it gets triggered, it grabs the new summary and the existing consolidated dataset from the consolidated/ folder in S3. It doesn't just pile on more data. Instead, it updates the stats recalculating averages and standard deviations for each device pair and then overwrites the old consolidated file. This keeps the storage clean and prevents files from endlessly growing.

Finally, when the data is consolidated, we have got another EC2 client ready for the Export Client. It pulls the data_consolidated.csv from S3, filters it based on a user specified source and destination IP, and then pushes the filtered result back to S3, inside the /exports folder. This version is perfect for analysis, reports, or feeding into dashboards



```
[ec2-user@ip-172-31-27-110 ~]$ java -jar export-client-jar-with-dependencies.jar "192.168.20.48" "52.45.237.36"
Export request:
  Bucket: my-iot-uploads-group12
  Read:   s3://my-iot-uploads-group12/consolidated/data_consolidated.csv
  Filter: src_ip=192.168.20.48  dst_ip=52.45.237.36
  Write:  s3://my-iot-uploads-group12/exports/192.168.20.48__52.45.237.36.csv
```

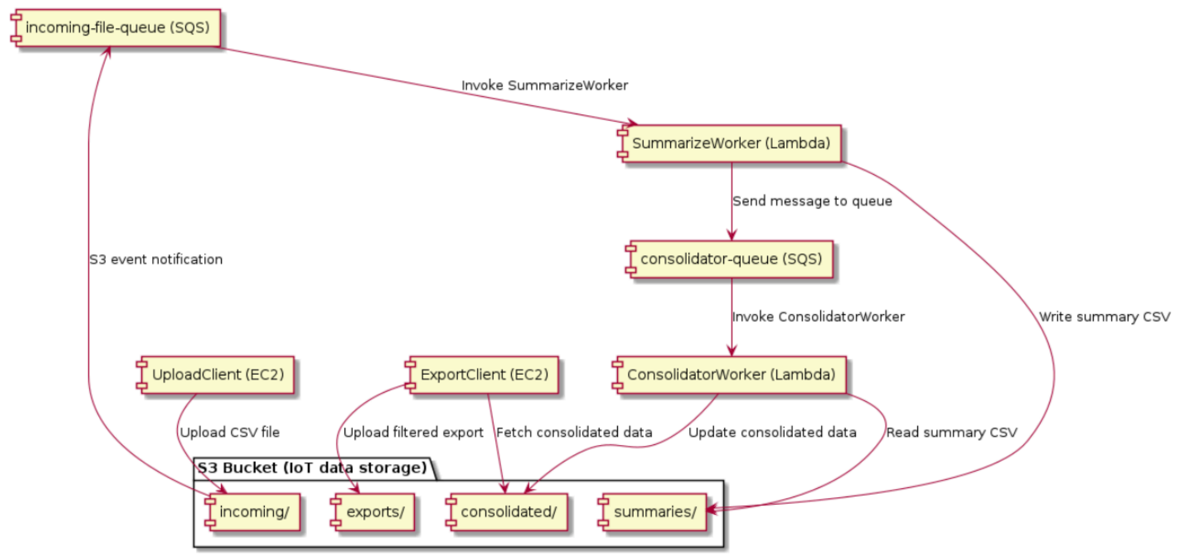
Implementation Design:

UploadClient (EC2): Encapsulates S3 client functionality for transferring raw traffic logs to the incoming directory, marking the pipeline's entry point.

SummarizeWorker (Lambda): Triggered by SQS messages, reads CSV files from S3, aggregates traffic records using internal dailyStats structure, and outputs compact summaries while signaling the next stage.

ConsolidatorWorker (Lambda): Maintains long-term metrics in consolidatedStats structure, overwrites a single consolidated dataset to prevent storage growth, and ensures statistics remain current.

ExportClient (EC2): Provides flexible reporting capabilities by applying custom filtering criteria and generating outputs for downstream analysis.



Key Benefits:

Reliability: SQS queues buffer workloads during traffic spikes and enable retry mechanisms, preventing data loss during processing errors.

Scalability: Lambda functions automatically scale based on queue depth, enabling parallel processing without manual intervention.

Storage Efficiency: Early summarization and single consolidated dataset design keeps storage usage bounded as data volumes increase.

Component Selection Rationale:

Amazon S3: Provides durable, scalable storage with native event triggering capabilities ideal for both storage and processing coordination.

Amazon SQS: Critical for decoupling components and ensuring fault tolerance through message buffering and retry support.

AWS Lambda: Optimal for short-lived, event-driven summarization tasks with automatic scaling and no server management overhead.

Amazon EC2: Offers flexibility at ingestion and export boundaries for custom scripts and controlled access patterns typical in IoT environments.

Conclusion:

This event-driven architecture efficiently transforms IoT traffic data into actionable insights through incremental processing. The design balances simplicity with robustness, following cloud best practices to create a practical, extensible solution that scales with data growth without major redesign requirements. The design avoids unnecessary complexity while following cloud best practices, making it a practical and extensible solution for IoT analytics. As data volumes grow, the pipeline can scale without major redesign, ensuring long-term usability and effectiveness.