Contents

System design

I've chosen to implement the solution using API Gateway, Lambda, SQS, and DynamoDB (along with an S3 bucket for a dead letter queue to deliver to).

I'm using terraform to provision and deploy the service.

I wrote the command line tools in pythnon 3.8.

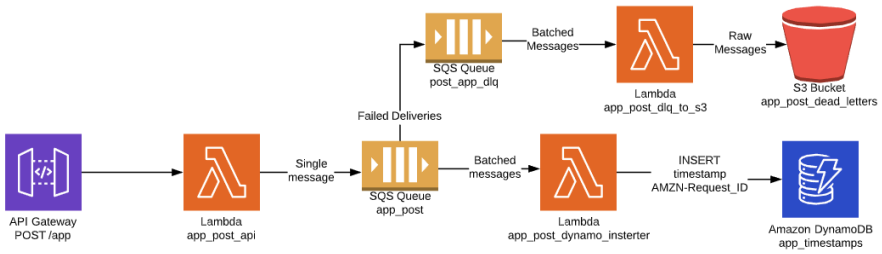

Figure 1: Proof of concept high level design

Considerations

Choice of a decoupled event driven architecture

I chose to make the requests submit events into a queue (SQS) which would be consumed separately and entered into the database in batches. This has the following benefits:

- Client requests do not depend on the speed of the database to ingest
- Requests get buffered before going into the database and are entered in batches which is a significantly more efficient process for most databases
- If the database is down for any reason (undergoing maintainence, during a disaster scenario etc) requests will still be accepted and buffered and retried for up to 24 hours giving much higher availablility to clients and easier more stress-free maintaininces to the engineers.

- If the database comes unexpected high load requests will buffer while the autoscaling scales up the database meaning congestion during the scaling activity will not affect clients or ingestion of raw data.
- SQS will give us automatic retries so if the insert fails for any reason it will get retried for up to 24 hours.
- Any requests which fail to insert after this time will be delivered to a dead letter queue (configured to store the files on s3), this allows for analysis to be performed later on failed requests to understand what's happening, it also means in the event of a multi-day outage you have lost no raw data and can catch up by replaying the requests which did not make it into the database/
- It's easy to change where the data is going by changing the consumer of the queue, so if DynamoDB turned out to be a poor choice it would be easy to swap it out with another data store.

Choice of database

I've chosen to use DynamoDB here for this proof of concept. I do not know how the data will be used later or accessed, depending how the data needs to be accessed Dynamo could be a poor choice, if we need to read out all of the entries, order them, or aggregate them Dynamo is not a good choice, although we could alleviate that by using an AWS DynamoDB to S3 datapipeline task. More likely however a different choice of database would be more suitable, possible a multi-az AWS Aurora (Postgres Compatible would be my choice) database. Another alternative (depending on how the data is used) is to batch up the requests and write them as parquet (for example) files to s3, then query with AWS Athena/Elastic Map Reduce.

I chose DynamoDB since this is a proof of concept and Dynamo gives us the following benefits:

- Lowest overhead to get the POC running
- Full point in time recovery
- Fully managed and automated replication over 3 availability zones in a single AWS Region (with the possibil to expand to multiple regions using a DynamoDB Global table)
- Automated (and manual) snapshots with easy recovery
- Easy monitoring with cloudwatch metrics and alarms (more comprehensive monitoring is easy with an external provider such as SignalFX or Datadog)
- Trivial autoscaling
- Extremely low maintainence requirements and low burden on operational/infrastructure teams

Since the architecture is decoupled from the api request by an event queue it would be easy to replace DynamoDB with a different database, or to replace the simple SQS queue with something more fully featured (Kinesis, Kafka, maybe Firehose depending on the volume) which would make delivery to multiple end

points easier, all of these solutions impart higher complexity though and so carry additional maintainence overheads and higher cognitive load to understand the system.

Choice of api

I chose to use API gateway to provide the api layer with each request triggering an AWS lambda.

Using API gateway with lamdbda has the following benefits

- Automatically highly available
- Easy to integrate with cloudfront and AWS Web Application Firewall, or an external provider such as cloudflare for DDOS prevention etc.
- Scales to extremely large volumes when required

There are some negatives

- Depending on the existing skills and deployment pipelines of the developments teams this approach may require additional training and engineering of delivery pipelines (it's more common in my experience to have developers already comfortable delivering a service via a docker container into ECS/Kubernetes)
- Lambda cold starts can mean some requests may take extra time (on the order of a few seconds in bad cases), but this can be mitigated in a production deployment by either setting the lambda to have provisioned capacity so it's always warm (at additional cost) or by artificially hitting the endpoint to keep the lambda in use (monitoring systems may provide this as a side effect of monitoring the endpoint).

Monitoring, alerting, and observability

We can get an excellent insight into this system using AWS X-ray giving us distributed tracing which covers every element of the system.

We can also monitor the following cloudwatch metrics with a preferred monitoring tool (Cloudwatch Alerts configured to look at Cloudwatch metrics would suffice at a minimum, or a more sophisticated tool that integrates with AWS would be better still (such as SignalFx or DataDog)).

- API Gateway
  - Error rates (4xx, 5xx HTTP statuses), initial configured on static values but when real traffic trends are established in production you can use historical analysis, and possibly even trend analysis to alert more dynamicly, these can be difficult to get right and cause alert fatigue if they fire a lot).
  - Request Counts: Depending on the level of production traffic you would monitor for either a large sudden increase, or too few requests.

As with error rates historical trend analysis can help here, but you need an established baseline over a statistically significant period.
  - Duration of requests
- AWS Lambdas (all 3)
  - Errors - The number of invocations that resulted in an error
  - Throttles - The number of times AWS throttled your invocations (which do not show up as errors in the Errors count)
  - Duration - How long the lambda executions took
- SQS - Primary queue
  - NumberOfMessagesSent - This is the number of messages sent into the queue. In a sophisticaed monitoring tool w could compare this with the number of successful invocations of the api gateway lambda. Otherwise we can use static/historical analysis.
  - NumberOfMessagesDeleted - The number of messages that have been deleted by a client receiving the message (which acknowledges the message has been processed)
- SQS - Dead letter queue
  - NumberOfMessagesSent - This will tell us that there is a complete failure to deliver these messages to DynamoDB, if there is a significant rate of these it could indicate a serious system problem.
- DynamoDB
  - There are numerous things to monitor for with DynamoDB, so rather than recount them all here I will link to the AWS blog post which lists everything you should be monitoring for DynamoDB https://aws.amazon.com/blogs/database/monitoring-amazon-dynamodb-for-operational-awareness/
  - The most important in my eyes though (having any is better than none, and you have to start somewhere) are:
    * UserErrors - Errors caused by bad requests
    * SystemErrors - Errors internal to DynamoDB
    * ThrottledRequests - How many requests have been throttled (which would hopefully be transient while the autoscaling kicks in, until you reach your maximum capacity, even then with the buffering from SQS unless this is sustained over a long period this may not be a problem)
    * ConsumedWriteCapacityUnits - How much write capacity you are using

We could potentially also monitor the cost of the solution by setting up an AWS Budget for a well known tag applied to every resource and creating a budget alert.

We would probably not want to alert an on call engineer for all of these (or even most), but for select metrics, and even for those it would be better to have a low priority and high priority alerts. Low priority alerts only being received during business hours and ideally configured to alert in advance of an incident becoming serious.