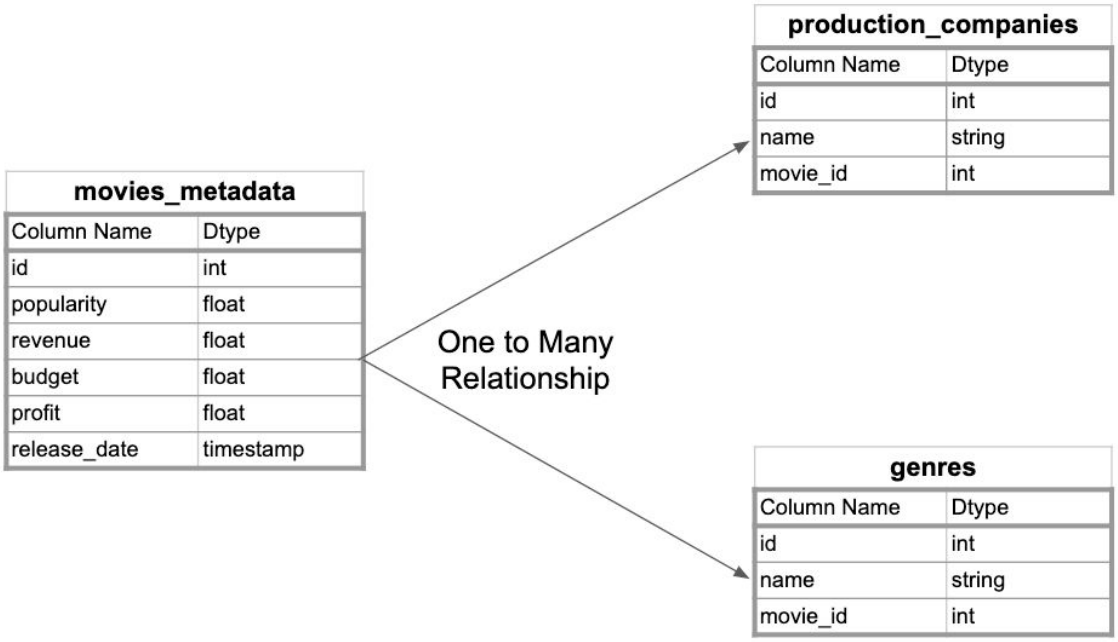
**Problem:**
Users need access to high level movie performance statistics aggregated by year, production company, and genre of movie via an API. The service must ingest incremental datasets monthly and recalculate all aggregate statistics.

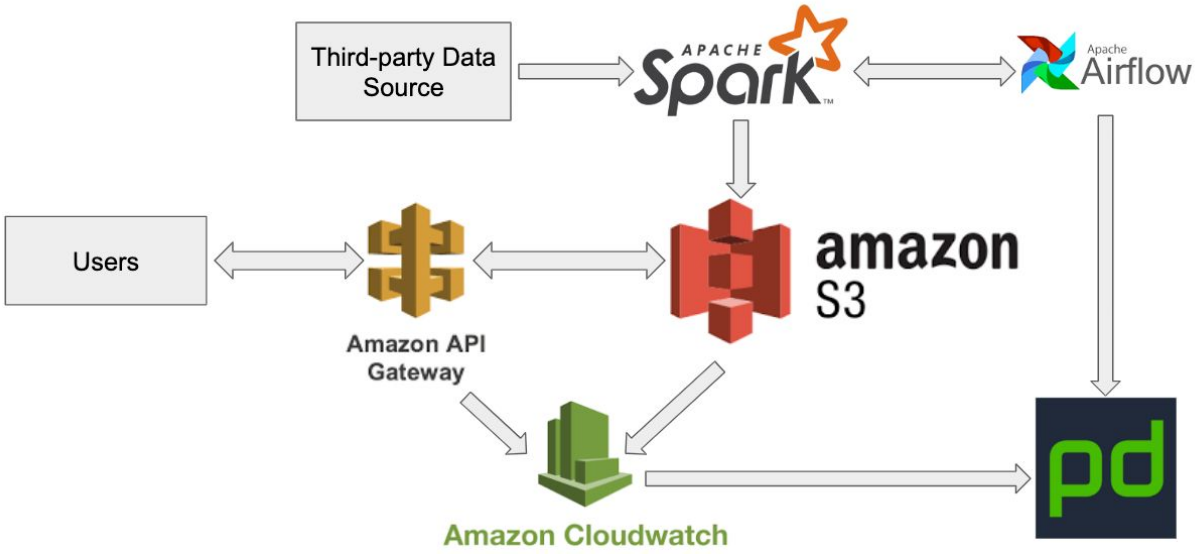
**Assumptions:**
- New data will be incremental.
- Subsequent datasets will be structured the same way as *the-movies-dataset.*
- Subsequent datasets could contain data from any given time period.
- *movies_metadata.revenue* less *movies_metadata.budget* is equivalent to movies' profits.
- Given the publicly-accessible nature of the data, authentication is not necessary.

**Data Model**




*movies_metadata* - This table represents individual movies and their metadata
*production_companies* - This table represents movie-to-production company combinations.
*genres* - This table represents movie-to-genre combinations.

**Proposed Software Design**




## Data Ingest and Storage

**Summary**

- Apache Spark jobs copy raw data to S3 or S3 glacier.
- Spark munges the data into the data model pictured above.
- Each result to be exposed is calculated once and stored in its own S3 key for retrieval.
- All scheduling and dependency management is handled by Airflow.
- Monitoring of ingest Spark jobs is implemented using Airflow's Pagerduty integration.
- Monitoring of S3 and S3 Glacier is implemented using AWS CloudWatch in conjunction with Pagerduty.

**Technical Implementation**

- Using Airflow and Boto3, all data is copied from the source location (S3, SFTP, etc.) to S3 or S3 Glacier partitioned by ingest year and month under the *the-movies-dataset-internal* S3 key.
    - All relevant information needed for the data model to generate the desired aggregate statistics is found in the *movies_metadata* files, which are stored in an S3 bucket.
    - Additional collected data is not necessary at this time, but is nonetheless stored in Amazon S3 Glacier.
- Using Spark, *movies_metadata* is munged into three tables: *production_companies*, *genres*, and *movies_metadata* (pictured above).
    - Resulting dataframes are saved to the *raw_munged* as parquet files partitioned by both ingest year and month without duplicates.
- Spark reads the files in the *raw_munged* key and performs aggregate statistics to generate a new set of results, which are stored under the *api_endepoints* S3 key.

**Considerations**

Scaling

- S3 and S3 Glacier can scale to whatever size necessary without performance impacts. (See AWS S3 FAQs)
- If traffic to S3 spikes, AWS will spread load evenly (across all S3)
    - This service's performance should not be affected by spikes in traffic. (See AWS S3 FAQs)
- Spark can also scale if the size of the data grows by adjusting job configs/cluster sizes.

Cost

- By using S3, we will only have to pay for storage used.
- Using S3 glacier saves on storage costs for data not needed readily available but potentially beneficial in the future.
- Airflow provides retry logic, dependency management, and scheduling at the cost of managing an airflow cluster.
- Given the static nature of the data, there is no need to maintain a RDMS. Rather, the service can generate the results after new data is ingested monthly. That way, the only compute costs are munging data monthly and calculating aggregate statistics.

Monitoring

- By using Airflow, developers can check on running ingest and munging jobs to determine if anything is awry.
- If any ingest or munging jobs fail, Airflow can alert developers using Pagerduty to determine the root cause of the issue.

Failure Recovery

- Airflow's built in retry logic will handle any transient failures.
- Partitioning all data by ingest year and month ensures standard partition sizes, keeps track of which data was ingested when, and provides the ability to rerun any previous data ingests from the raw CSV files and/or *raw_munged* to generate results.
- Manual retry can be performed via the Airflow UI. All tasks are idempotent, so there will be no need to clear out data in existing partitions to rerun a task or DAG.

Planning for the Future

- If it is beneficial to expose information stored in tables other than *movies_metadata*, that data is preserved in S3 Glacier and can be added to the existing data model.
- If it is beneficial to expose information stored in the other columns of *movies_metadata*, parquet files support schema evolution, and can be added to the existing data model.

Notes

- In the python script provided, copying files from third party data source to the internal directory will be implemented using the os library. In production, file replication will be performed using the AWS Boto3 library to provide S3 functionality.
- In production Spark will not run locally. Instead, it will run on a multi-node cluster via AWS monitored by Cloudwatch.
- In production all 'lit(year/month)' would be replaced by airflow Macros representing the run date.

# API

**Summary**

- The API will be implemented via AWS S3 Proxy implemented using Amazon S3 and Amazon API Gateway (in a very similar architecture as outlined here).
- All infrastructure will be monitored using AWS Cloudwatch in conjunction with Pagerduty for alerting.

**Technical Implementation**

- Users will have access to HTTP endpoints to run GET requests handled by Amazon API Gateway.
- Amazon API Gateway will have the permissions to get and list the contents of the specific S3 Keys where the aggregated results are stored in S3 via an IAM role.
- Method responses will be defined from the API Gateway console (200, 404, 500, etc.)
- Configuring API Gateway Child Resources as GET methods to each S3 key containing a specific result creates our HTTP endpoints where they will get the data from S3.
    - Users can send GET requests to specified HTTP endpoints, configured API Gateway Resources, to retrieve data stored under the *api_endpoints* key.
- If a website is needed to provide documentation or general information for the API, could a static S3 website can be set up as described here.

**Considerations**

Authentication

- A variety of options will be available for different use cases if users need to be authenticated.
    - If for internal use at Guild, IAM roles and corresponding policies could be used as described here.
    - If for external use, authentication via an AWS Lambda and bearer tokens could be used as described here.

Monitoring Considerations

- All monitoring will be implemented via AWS Cloudwatch in conjunction with Pagerduty for alerting. Since the entire API integration will be implemented using AWS infrastructure, Cloudwatch will provide one source of truth for the entire service.

Scaling

- The Amazon API Gateway is only a proxy for S3. The API will scale as well S3 can.

**Trade-offs**

One big trade-off with this architecture is the serverless nature of the API (not including Airflow). In the future, if dynamic queries on the data model were allowed as a request parameter, this architecture, which only serves static files, does not currently support that. As this not a requirement at this time, the cost savings of only calculating the results once are preferable. Furthermore, this architecture would not be viable option for receiving data in real-time. The choice for a serverless S3 proxy approach is based around the cadence at which files are received and the static nature of the results to be exposed.

An additional trade-off is to break out the data model into multiple tables when one would suffice. I chose to opt for a more traditional relational database structure (primary key → foreign key), rather than using one table that can be read once to generate all results. I made this choice because as a former analyst, I found that a traditional data model is easier to understand and far easier to work with. The cost of joining tables is not worth the necessary overhead of teaching new contributors how an overloaded data model is structured.