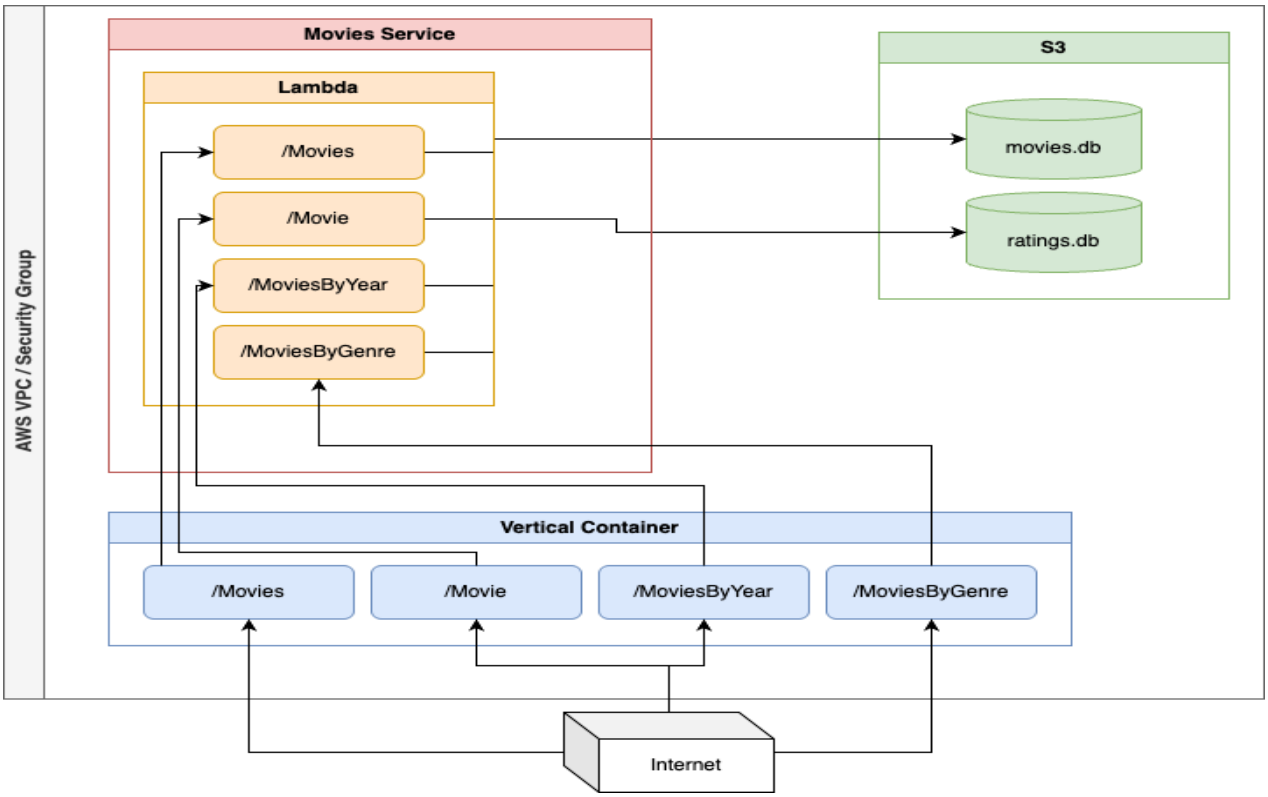
ARCHITECTURE



- For a simple API as described, I recommend a serverless architecture.
    - **AWS Lambda** – These provide the compute for the system. A function will be equal to one endpoint. Any updates to an endpoint will simply update the single Lambda for that endpoint.
    - **AWS API Gateway** – provides routing / mapping of the lambda functions to the URL for the endpoint.
    - **Database** – given the size and scope of the current databases, I recommend the use of **AWS S3** as storage for the database files and to execute the SQLite commands against those files. Updates to the files can be made at will and versioning can be used to maintain old versions of the files as they are updated. **AWS Aurora** could be used to host a MySQL or Postgres version of the database should the scale and business see a need for the additional functionality and costs associated with that upgrade.
    - **AWS SAM** – Serverless Architecture Manager allows for the easy configuration, deployment, and maintenance of this system. Changes are made to the configuration files and those control the roll-out and deployment of the services and database. **Serverless** is another possible framework that is not in the AWS family but provides similar features.

    - **GitHub –** code repository for all system. While not a member of the AWS services, it is the most feature rich and popular code repository system today. **Github Actions** allow for automation and CI/CD to keep updates moving through the pipeline efficiently.
    - **Terraform –** Infrastructure as Code system to handle the build and teardown of the system as needed. This is minimal with the current requirements, but as the system grows and becomes more complex, these features become integral to management and maintenance of the system.
- This architecture provides several advantages:
    - Minimal up-front costs. All services used as managed services, so the expenses are proportional to what is used.
    - All systems are language agnostic and all developers to work in the languages with which they are the most familiar. Lambda supports Node.js, Java, and Ruby.
    - Infrastructure is handled by code (IaC) so it is written and committed to a repository like all other code to be reviewed and executed as needed.
- Some trade-offs are required to get these advantages:
    - Lambda function has a start-up time if they are not already in memory. Given the size and complexity of the endpoints in the requirements, this will only be minimal issue, but as usage of the endpoints increases, some features of Lambda can be used to mitigate this issue.
    - Costs are variable and will increase with use of the system. Since Lambda is billed per invocation, more use is more costs. Generally, since that also indicates higher adoption rates, this is not an issue.

GENERAL INFORMATION

- The api will be versioned starting with the value *v1*. Future iterations and releases will only increase the version if there are breaking changes to the endpoint. So different endpoints can exist and be used in different versions at the same time. Endpoints and versions can be deprecated and removed as business cases allow.
- All endpoints will handle paging via an optional '*page'* parameter. This value will default to 0 and thus will return the first page if not included. All endpoints that return a list of results will get a full result count for the query and will use that plus the page variable and the page size (50) to determine if other results are available. If there are more results that could be returned in subsequent pages, the system will return the *nextPage* property with the value of the next page. If there are no more results, then the *nextPage* property will contain *null* to signify that no more results

can be returned. With this, a complete list of the results can be obtained if needed from the api with minimal calls.

- To protect the services from DoS and DDoS attacks, a throttle will be implemented to only allow a certain number of requests per second to the endpoints. This allows us to control the usage and to deter bad actors from causing issues on the system. A given IP will only be allowed <Rate Limit> requests per second and then must wait <Back off Time> before other requests will be allowed. An appropriate error messae will be returned in this case.
- Other security measures can be implemented with this system like access tokens to control access. Current requirements did not mention sure measures at this time, so those are not detailed in this recommendation.

ENDPOINTS

GET /movies

https://<BASE_URL>/v1/movies

Query:

SELECT imdbId,title,genres,releaseDate,PRINTF('$%,d',budget) as fmtBudget

FROM movies

LIMIT(50)

OFFSET (<page>*50);

Query Parameters:

page - indicates the page to show in the results. Defaults to 0.

Response:

nextPage: number of the next page to be returned. is null if no further pages are available

count: number of results

currentPage: value of the page parameter

results: array of results of the query

GET /movie/<id>

https://<BASE_URL>/v1/movie/<id>

Query:

```
SELECT
   m.imdbId as imdbId,
   m.title as title,
   m.description as description,
   m.releaseData as releaseData,
   PRINTF('$%,d',m.budget) as fmtBudget,
   m.runtime as runtime,
   AVG(r.rating) as averageRating,
   m.genres as genres,
   m.language as language,
   m.productionCompanies as productionCompanies
FROM movies m JOIN ratings r ON m.movieId = r.movieId
WHERE m.movieId = <id>
```

Query Parameters:

Id – id of the movie to be returned

Response:

count: number of results

results: record for the given movie or null

GET /moviesByYear

https://<BASE_URL>/v1/movies/year/<year>

Query:

SELECT imdbId,title,genres,releaseDate,PRINTF('$%,d',budget) as fmtBudget

FROM movies

WHERE SUBSTR(releaseDate, 1, 4) = <year>

ORDER BY releaseDate asc|desc

LIMIT(50)

OFFSET (<page>*50)

Query Parameters:

Year – the year the movie was released

page - indicates the page to show in the results. Defaults to 0;

order – optional parameter to determine the order of the returned movies in the selected year. Default value is 'chrono'. Possible values are:

'chrono': chronological order

'reversed': reverse chronological order

Response:

nextPage: number of the next page to be returned. is null if no further pages are available

count: number of results

currentPage: value of the page parameter

results: array of results of the query

GET /moviesByGenre

https://<BASE_URL>/v1/movies/genre/<genre>

Query:

SELECT imdbId,title,genres,releaseDate,PRINTF('$%,d',budget) as fmtBudget

FROM movies m JOIN json_each(m.genres, '$') g

WHERE json_extract(g.value, '$.name') = 'Comedy'

LIMIT(50)

OFFSET (<page>*50)

Query Parameters:

genre– The genre the user wants to search for

page - indicates the page to show in the results. Defaults to 0;

Response:

nextPage: number of the next page to be returned. is null if no further pages are available

count: number of results

currentPage: value of the page parameter

results: array of results of the query