RhythmIQ High-Level Design

Ryan Hamby (hambyr@umich.edu)
January 20, 2025

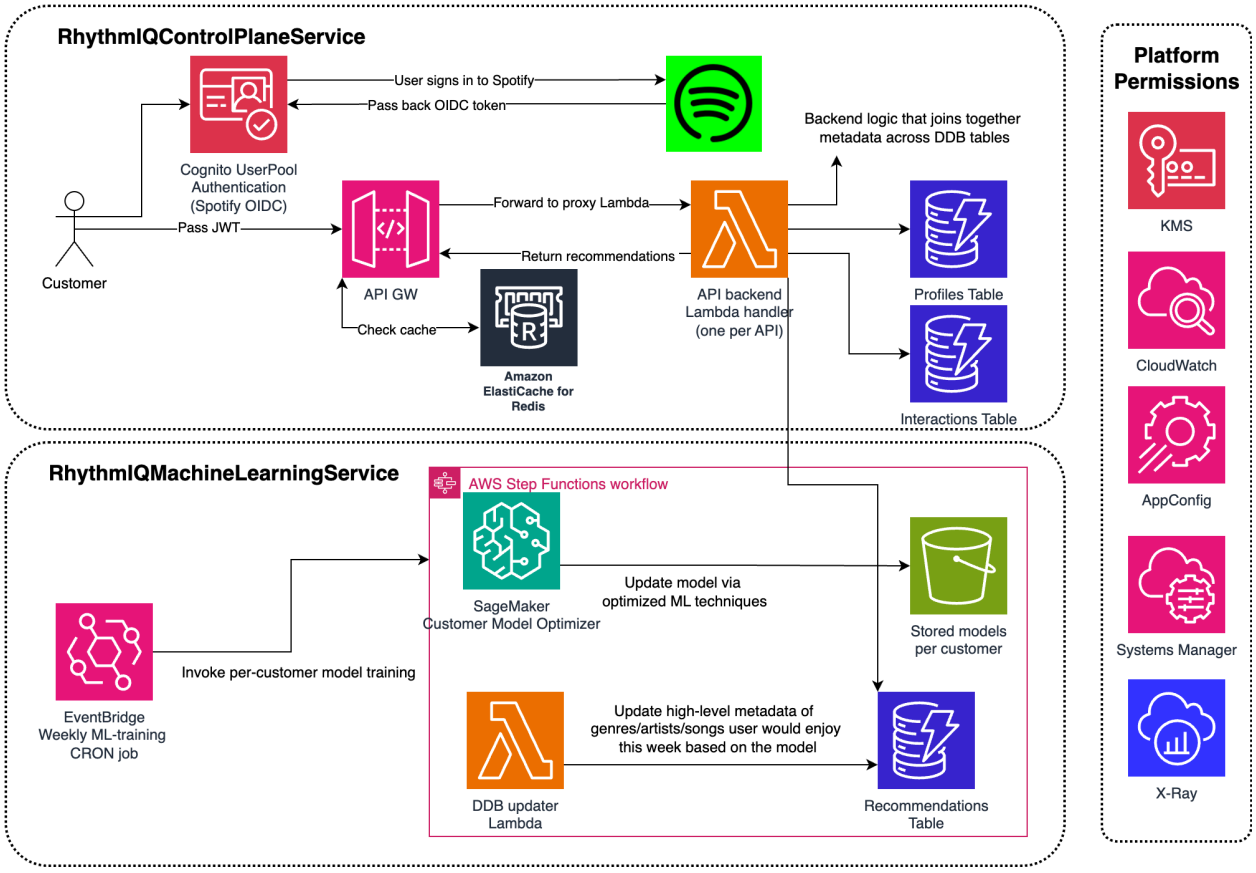
Purpose

This document follows from the high-level design [link] created for developing the environment-driven music recommendation system, RhythmIQ. Within this LLD, several technologies are explored with tradeoffs and considerations outlined to explain the thought process of choosing specific components.

Overview

This application includes a public control plane for handling high-TPS requests and an internal ML service responsible for updating and managing machine learning models.

Figure 1. RhythmIQ High-level Architecture Diagram



Resources

All resources will be created with a unique 36-char UUID to serve as their identifier. UUIDs are unique per resource and cannot be set by the user.

Top-level Resources

1. **Profile**
    a. When a user onboards, he/she creates a profile
        i. In the response, it returns a profile-id (UUID) which is used for other APIs
        ii. This UUID will be used in most instances as keys for things like API keys and S3 object keys
            1. See _____ section to discuss UUID compression for optimizing cache memory utilization
    b. Captures basic details about a user
    c. Many components in the system rely on a unique email– there cannot be any two users with the same email
    d. A profile's UUID and email address are immutable once set. If a profile is deleted and recreated, a new random UUID will be assigned.
        i. Emails can be re-used only when the profile is deleted
        ii. In order to delete a profile, a user must also first delete all of its secondary resources. The system should be designed such that it could be possible for RhythmIQ to delete all second-level resources on behalf of the user automatically.
    e. See Appendix ___ to see the shape and attributes of this object

Second-level Resources

2. **Preference**
    a. Represents a user's favorability to choose certain types:
        i. Initial preferences will be a **`type`**: **`genre`**, **`tempo`**, **`instrument`**, or **`artist`**
        ii. Must fit into one of these types. Will be validated when created/updated
        iii. Can have up to 100 preferences
            1. Favorite preference will be at index 0
            2. Least favorite preference will be at index 99
    b. Just like a user's preferences change, a **`preference`** can alter over time as well
        i. A preference **`type`** cannot be changed once created. The preference must be deleted and then recreated with a different type
    c. Configurable such that the application can weight each preference
    d. Composed of two components
        i. User-set preferences → what the user sets/tells the application he/she enjoys listening to the most

        ii. Application-learned preferences → information that the application learns overtime by analyzing music listening patterns
            1. Will be kept internal for initial release. If this data is valuable enough, exposing this can be sold to customers as part of a premium subscription
    e. See Appendix ___ to see the shape and attributes of this object
3. **TuneVector**
    a. Represents playlists, podcasts, soundscapes, etc that are delivered to the application user.
    b. The vector can be re-sorted, adjusted to point in a new direction, or even curve a user into a zone of focus through a series of songs and sounds
        i. For example, imagine this is a 5D vector relying on a set of variables
        ii. The vector can start the music in a dopamine-inducing state involving something such as rock music, and slowly transition into slower, quieter songs that deliver soundscapes emulating a conducive study environment without the user noticing or managing the application.
            1. This feature can be referred to as “Ascent” or “Descent”
    c. TuneVectors should represent algorithms similar to short-form video algorithms in which each set of songs is tailored to the ML-based content that the user prefers.
    d. North star → support modes in which:
        i. The user can guide the vector’s progression via a time-constrained selection of genres to navigate through
        ii. An AI mode in which the vector self-adapts based off music listening patterns and other environment-related data
    e. See Appendix ___ to see the shape and attributes of this object
4. **Variable**
    a. Variables refer to the externally-exposed information about the environment of a user
    b. These are not an exhaustive list of parameters used to train a model
        i. For instance, **`averageTimeToSkip`** will be kept internal to the application
    c. A user must opt-in to each variable. Variables will not be opted into automatically.
    d. A user can opt-out of collecting variable data at any time.
    e. This can include, but is not limited to:
        i. Ambient noise level
        ii. Time of day
        iii. Outside temperature
        iv. Location and specifically patterns in location
        v. Ratings of previous vectors recommended for the user
        vi. Heart rate variability (HRV)
        vii. Accelerometer (helpful for runners and similar athletes)
    f. See Appendix ___ to see the shape and attributes of this object

Components: Considerations and Tradeoffs

1. **AWS API GW**
    a. Will use a REST API format, as opposed to HTTP or WebSocket
        i. HTTP APIs have downsides such as no caching, no rate limiting, and no support for AWS WAF and backend authentication.
        ii. WebSocket APIs are useful for when the client and server need to send bidirectional messages to each other independently.
            1. This is useful for streaming and real-time services, but not here.
    b. This component will not use ACM at P0. ACM is useful for custom domain names, but is not necessary for a P0 of this project.
        i. Additionally, this project will be bootstrapped to avoid costs to manage and maintain over time, and custom domain names add costs.
    c. All API GW REST APIs have TLS 1.2 and SSL/TLS certificates by default that are owned and managed by AWS
    d. CORS will be applied at the API resource level
        i. Structure is API GW → API → Resource → Method. An API is deployed to a stage via a deployment in order to “publish” it for use
        ii. REST APIs
    e. Usage plans will key by the <IdP_name>_<Cognito User Pool sub claim>
        ********** Should use the UUID of the user
2. **AWS Cognito**
    a. This will use Cognito User Pools to handle authentication
    b. Spotify uses OIDC Federation to allow users to sign in with Spotify and pass credentials back to the AWS-hosted application
    c. Users will call the API GW endpoints with a Cognito JWT
3. **AWS Lambda**
    a. Lambda is the most cost-effective technology for this application at its initial launch, until the application scales up to require lower-latency and higher-TPS constraints
        i. ECS or EC2 would be a better choice for a backend infrastructure platform when containerization and/or custom workloads need to run (especially if the processing time for specific requests exceed 15 minutes)
    b. To reduce cold start latency, this application can use either SnapStart or provisioned concurrency.
        i. SnapStart is free for Java applications, but costs money for others. This can improve cold start times by up to 90%
        ii. Provisioned concurrency (on an alias) costs more money but is best for latency-sensitive applications since it eliminates cold starts when the demand for concurrency <= provisioned concurrency
    c. Each function will be assigned a reserved concurrency of 10 by default in order to prevent DDoS attacks from consuming all Lambda concurrency within an account
    d. Memory will be scaled up per function, but will start at the minimum of 128 MB.

e. For any jobs that require loading ML models in to ephemeral memory, the limit on ephemeral memory will be scaled accordingly (with a CW alarm)
f. Lambda timeouts will be set about 50% higher than their p99 execution time after observing traffic for about 1 week on each function. The default will be 1 minute to help scale up operations

4. **Amazon S3**
    a. S3 will house a pre-processed format of data before it is sent to SageMaker.
    b. S3 customer training data will be stored in the **song-recommendation-data-<account-id>-<region>** bucket
    c. Artifacts of SageMaker are also stored in S3
    d. An application can store thousands of models in S3. Scalability for a 1-1 mapping per user is not a major concern
    e. Objects will be versioned such that the user can have their most recent model be stored as the most recent version
    f. S3 buckets will be encrypted with S3-SSE
    g. S3 objects will be stored in One-zone Infrequent Access (IA) storage tier. If a model is lost due to natural disaster or outage, the most up-to-date data is stored in the DDB table used to create a new model. If the S3 object is lost due to a zonal outage, the application will throw an error and retry training on impacted users.
5. **AWS SageMaker**
    a. This is purely an ML-based use-case. There is no clear use-case for using GenAI, so we leave Bedrock out of scope for now
    b. Will make use of Multi-model endpoints to load more than one model dynamically from S3 at a time
    c. Will use serverless inference in order to only pay when the model is invoked. This will also utilize spot instances to save costs up to 70%
    d. Will use FP16, INT8 quantization, or knowledge distillation in order to compress models for cheaper storage and faster inference.
    e. It's likely the best models will be built with Matrix Factorization or Neural Collaborative Filtering, but for initial training this application can use AutoML with SageMaker Autopilot to find the best model for the recommendation system
    f. Once a model is selected to show the best results (with graphs of how this is determined), we can select hyperparameters to optimize for custom tuning.
6. **Amazon DynamoDB**
    a. Will be used as the primary database for storing user interactions, metadata, and model recommendations.
    b. Table Structure:
        i. **Users Table**: Stores user profiles and preferences.
        ii. **Interactions Table**: Logs user interactions with recommended songs for feedback loops.
        iii. **Models Table**: Stores metadata of trained models and their versions.
    c. Indexing:

        i. GSI (Global Secondary Index) for querying user interactions efficiently. Up to 20 GSIs can be created on a table, but can be done after table creation.
        ii. There is no base case to search the sort key efficiently, so no LSI will be created. This must be done at creation time.
    d. Capacity Mode:
        i. On-demand capacity mode initially for cost savings and automatic scaling.
        ii. Provisioned mode can be enabled with Auto Scaling once traffic patterns are predictable.
    e. Data Retention:
        i. TTL (Time to Live) will be used to delete stale interaction data after 90 days.
    f. Backups and Security:
        i. Point-in-time recovery will be enabled for critical tables.
        ii. Encryption at rest with AWS KMS.
7. **AWS StepFunctions**
    a. will orchestrate complex workflows in a serverless manner
        b. Use Cases:
            a. Orchestrating model training and deployment lifecycle in SageMaker.
            b. Managing ETL jobs for processing interaction data.
    2. Error Handling & Retries:
        a. Implement exponential backoff for transient failures.
        b. Dead-letter queue (DLQ) for capturing failed executions.

Tech Stack

1. **Frontend**
    a. Application will use a React frontend with `react-native-web` in order to allow an easier transition to React Native if the application eventually requires a robust mobile platform
    b. React offers extensive library support, component reusability, browser support, and testing libraries
    c. Alternatives below were considered:
        i. Vue.js → too simple and only useful for simple apps, especially single-page ones
        ii. Express.js → this also defines a backend, which should instead be handled by Java directly
        iii. Angular.js → higher learning curve, less support for mobile applications
2. **Backend**
    a. Code that handles client-facing requests will be written in Java (JDK 21). In general, backends written in Java are strongly-typed and can be maintained well.

        i. Java also currently is the only language that AWS Lambda supports for free when using Lambda SnapStart. The alternative is to use a vast amount of provisioned concurrency on Lambda functions for every API, which will grow costly.
        ii. If the backend does migrate to EC2 or ECS, the stack is more portable on a JVM-based application stack. This enables efficient memory management, multithreading, and language support.
        iii. Note: If AWS had more support for Go, then Go would be considered more since it has better memory footprint and runtime performance.
    b. Code that handles ML model training and recommendations will use Python. This will leverage CRON jobs and Python-based ML libraries such as Pytorch and Tensorflow, which primarily build on Python.
3. **Infrastructure**
    a. Will use AWS CDK. CDK is a clean interface to be able to organize AWS resources and stacks programmatically.
    b. Another option is Terraform, but this does not support nearly as many features as AWS CDK since it is owned by a separate company. It is unlikely that the vendor will be switched to a different cloud platform.
4. **Dependency Injection**
    a. Dagger is generally the gold standard if not using SpringBoot
        i. SpringBoot is useful to define the entire server, and generally is built for long-running applications, not those confined to 15 minute runtimes
        ii. Dagger also benefits from lighter memory footprint and runs its DI checks at compile time, making it easier to use for serverless frameworks
    b. Google Guice is also a powerful library but tends to suffer memory constraints
5. **Pipeline and CI/CD**
    a. Will operate out of the us-east-1 region on AWS. This region supports the highest quotas and is generally easiest to work with services ACM, IAM, and CloudFront.
    b. Will host PRs in GitHub, and merge on UI
    c. Will use AWS CodeBuild, AWS CodePipeline, and AWS CodeDeploy to manage CI/CD
    d. Utilizes CW alarms to be able to rollback and block deployments
    e. Will be split into the following stages:
        i. Devo
            1. Allows for test deployments in local environment
        ii. Beta
            1. Allows for first-stage deployments. There should be at least two stages between a newly-merged commit and production.
            2. 1 hour bake time
        iii. Gamma
            1. Runs any integration tests and some canaries before the code reaches prod.
            2. Leaves a 3 hour bake time such that any issues can be caught
        iv. Prod

        1. Deploys the new code in gradual rollout.
        2. 4 hour baketime at the end in case there is a need to rollback.
6. CRM
7. Project Management
8. Monitoring
9. Logging
10. Deployment
11. Testing
    a. Unit testing → 90% line and branch coverage enforced

API Design

This section serves to enumerate a few options available for clients to call the APIs.

A few considerations:

1. Versioning
    a. This set of APIs will not support versioning from the start. Versioning APIs can be costly, controversial, and be a side effect of improper upfront design
    b. See this reference for more details on why not to version
    c. Google also has a published public doc on this, and recommends content-negotiation or v2+ versioning if it does end up being necessary
2. OpenAPI Definition
    a. This combines concepts of both REST and gRPC into one (ie, it uses full path discoverability to operate on entities that are abstracted by procedures by the RPC convention)
    b. This application may not necessarily be able to enforce all client protocols, so gRPC is not the best choice in this application
    c. Here is a Google-written doc on this design choice
3. Protocol
    a. HTTP/2. New APIs on web-based servers experience faster performance and scaling with this version. HTTP/3 is slightly newer, and sticking to TCP transport layer protocol is still stable technology.
        i. HTTP/3 is not supported in some browsers yet such as Safari
4. Profile APIs
    a. There is no ListProfiles API since users can only have a single profile. If needed, ListProfiles will be created as an internal-only API
5. Replacement behavior
    a. Full replacement (PUT)
        i. Useful for overwriting an entire configuration/resource and resources are simple and/or infrequently updated.
        ii. Passes ownership of the resource lifecycle management to the client since the client has to maintain the previous and expected new state of the metadata.

        iii. Will be used for the **Profile APIs** that deal with simple structures and limited metadata
    b. Partial update (PATCH)
        i. Useful when the data is large, updated frequently, and clients only need to modify specific fields.
        ii. Can be more difficult to manage on the backend but helpful for clients such that they don't need to manage large payloads.
        iii. Also helps to prevent bugs and customer misunderstandings from accidentally overwriting data in the application
        iv. Will be used for APIs that deal with **frequently updating data** and **complex structures**, such as music preferences
6. Authentication & Authorization
    a. Will use OAuth 2.0 in order to login to downstream clients, such as Spotify (also uses OAuth 2.0).
        i. The application will have to request certain scopes to read the data such as `playlist-read-private` , `user-library-read` , and `user-top-read`
        ii. Spotify and other music services will pass back a JWT
        iii. The application will refresh the token when it expires using the refresh token
            1. When a user is created, create an EventBridge rule that will invoke a Lambda to refresh the token before it expires and write to DDB.
            2. The refresh Lambda handler will create another EventBridge rule to continuously fetch the refresh token, and store it back in DDB
        iv. Will not use RBAC (Role-based access control) or ABAC (Attribute-based access control) since OAuth covers permissions
        v. This app will not support multi-user login to a profile, nor will it support multi tenancy. There will be a 1-1 mapping of the user to profile.
    b. Spotify will redirect the user back to the app with an authorization code in order to access user data needed to train the model, create playlists, etc
    c. To implement this on API GW, this will use a Lambda Authorizer that does the customer authentication.
        i. Cognito User Pools cannot be used since they do not support Spotify directly. Many Spotify accounts do not use Social login, so passing the Google/Apple/Facebook/Amazon login info will not unlock data
7. Cache
    a. Many non-mutating APIs are introduced in this application and eventual consistency is tolerable (read-after-write consistency is preferred, but not required), so caching is
    b. The API GW will require canary traffic on the APIs in order to measure a baseline SLA on the APIs. Once baseline traffic is calculated, the performance improvement of adding the cache can be calculated
    c. The default TTL of 300 sec will be used to start, and can be right-sized as traffic grows

    d. API GW built-in cache is sufficient compared to Elasticache (Memcached and Redis).
        i. The built-in cache enables better pricing since it only needs to cache GET idempotent requests. Elasticache would be more effective for caching DB-layer data or for ultra-high traffic with more access to fine-tune the cache
        ii. To note: Memcached is designed for simplicity while Redis offers a rich set of features that make it effective for a wide range of use cases. Redis offers snapshots facility, replication, and supports transactions, which Memcached cannot.
8. CORS
    a. This requires CORS to call the APIs from a custom domain that the application will be hosted from
    b. Clients can call the API in preflight to assess routes available for the CORS protocol
9. Rate Limiting
    a. Use API keys based on a random UUID of each profile, and then perform a lookup in the DDB for that API key if it exists each time.
    b. Both Google and AWS have documentation on how to map this to a user account
10. Error Handling
    a. Will surfaced all client messages to frontend with descriptive error messages
    b. The following will not show an error on the API itself, but the frontend may help to guide users by catching/rethrowing the errors:
        i. 200 OK
        ii. 400 Bad Request
        iii. 401 Unauthorized (failed OAuth)
        iv. 403 Forbidden (Authenticated, but not authorized)
        v. 404 Not Found
        vi. 500 Internal Server Error
        vii. 429 Too Many Requests
11. Testing
    a. All PRs will require 90% unit test line and branch coverage, and this will be enforced at build time as well
    b. Integration tests will be added to the pipeline in Jenkins
12. Monitoring
    a. Via AWS CDK, this application will have CloudWatch alarms based on the p99 latency, error rates, and availability of the API.
    b. If the application grows to need a containerized backend, Prometheus or Grafana should be considered to test endpoint availability without locking into the AWS ecosystem
13. Security
    a. APIs will adhere to sigV4 signing and require a generated SSL certificate rotated at least quarterly

b. AWS Web Application Firewall will be enabled to minimize chances of SQL injection and cross-site scripting attacks.

This application will have the following APIs:

API Name	Route	Method	Priority
CreateProfile*	/profiles	POST	P0
UpdateProfile*	/profiles/<profile-id>	PUT	P0
GetProfile	/profiles/<profile-id>	GET	P0
DeleteProfile	/profiles/<profile-id>	DELETE	P0
CreatePreference	/profiles/<profile-id>/preferences	CREATE	P0
UpdatePreference	/profiles/<profile-id>/preferences/<preference-id>	PATCH	P0
GetPreference	/profiles/<profile-id>/preferences/<preference-id>	GET	P0
DeletePreference	/profiles/<profile-id>/preferences/<preference-id>	DELETE	P0
ListPreferences	/profiles/<profile-id>/preferences	GET	P0
CreateTuneVector	/profiles/<profile-id>/tune-vectors	CREATE	P0
UpdateTuneVector	/profiles/<profile-id>/tune-vectors/<tune-vector-id>	PATCH	P0
GetTuneVector	/profiles/<profile-id>/tune-vectors/<tune-vector-id>	GET	P0
DeleteTuneVector	/profiles/<profile-id>/tune-vectors/<tune-vector-id>	DELETE	P0
ListVectors	/profiles/<profile-id>/tune-vectors/<tune-vector-id>	GET	P0
CreateVariable	/profiles/<profile-id>/variables	CREATE	P1
UpdateVariable	/profiles/<profile-id>/variables/<variable-id>	PATCH	P1
GetVariable	/profiles/<profile-id>/variables/<variable-id>	GET	P1
DeleteVariable	/profiles/<profile-id>/variables/<variable-id>	DELETE	P1
ListVariables	/profiles/<profile-id>/variables	GET	P1

Components for P1 and Beyond

Web Application Firewall

Appendix

CreateProfile

CreateProfileRequest

HTTP/2 POST /profiles
Content-type: application/json

```
{
  "username": "string", // required
  "firstName": "string", // required
  "lastName": "string", // required
  "email": "string", // required
  "phoneNumber": "string", // required
  "password": "string", // required
}
```

CreateProfileResponse

```
201 Created
{
  "message": "Profile created successfully",
  "profileId": "unique-profile-id"
}

400 Bad Request
{
  "message": "Validation failed",
  "errors": {
    "email": "Invalid email format",
    "password": "Password must be at least 8 characters long"
  }
}

409 Conflict
{
  "message": "Username or email already exists"
}
```

UpdateProfile

```
UpdateProfileRequest
```

HTTP/2 PUT /profiles/<profile-id>
Content-type: application/json

Note: omits email since email is immutable for an account

```
{
  "username": "string",       // Optional
  "firstName": "string",      // Optional
  "lastName": "string",       // Optional
  "phoneNumber": "string",    // Optional
  "password": "string"        // Optional, if changing passwords
}
```

```
UpdateProfileResponse

200 OK
{
  "message": "Profile updated successfully",
  "profileId": "<profile-id>",
  "updatedFields":
  {
    "firstName": "NewFirstName",
    "lastName": "NewLastName"
  }
}

400 Bad Request
{ "message": "Validation failed", "errors": { "email": "Invalid email
format", "phoneNumber": "Phone number must include a country code" } }

401 Unauthorized
{ "message": "Unauthorized access. Please provide a valid token." }

404 Not Found
{ "message": "Profile with the specified ID does not exist" }
```

```
409 Conflict
{
  "message": "Email or username already in use"
}
```

References

1. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-known-issues.html