CS-GY 9223 Cloud Computing
MobyPick: Book Recommendation System

Anubhav Dinkar, Arpan Chatterjee, Shirley Berry, Shruti Garg
NYU Tandon School of Engineering

Code Link: https://github.com/abunav6/mobypick
Dataset Link: https://mengtingwan.github.io/data/goodreads.html#datasets

Abstract

We have designed and built MobyPick, a cloud-based web application that provides users with personalized book recommendations. The main goal of our application is to encourage users to broaden their horizons in their book-reading habits and discover new books they may have otherwise overlooked. To that end, our application first allows users to sign up and create a profile on the MobyPick website, where they can set up their preferences. Based on these preferences, we suggest new books to each user tailored to their taste by generating recommendations through the AWS Personalize service. These book recommendations are based on customized recommendation models created by Personalize that were trained on Goodreads data. To make the application robust, quick, and reliable, we have implemented caching in the API layer to retrieve user data quickly, load balancing for the web application to minimize downtime in the case of failures, and, wherever possible, organized our data in a way to ensure fast retrieval. Apart from the personalized recommendations, users are also provided with a Lex-based chatbot to get quick recommendations based on user interactions with the bot.

Introduction

Receiving customized book recommendations can provide an easy way to broaden one's reading tastes, encourage an appetite for a variety of genres, and find hidden gems. Goodreads currently has 3.5 billion books. The average college student can read 250 words per minute (Brysbaert 2019); given that the average novel is about 90,000 words (Streichler), it would take most people 6 hours to read a single novel. Even narrowing down by things like language and genre, there simply isn't enough time to sift through that volume of data by hand to find new books, let alone waste time and money reading books that don't end up aligning with your preferences. To continue with the nautical theme, recommendation systems can act as a captain as you explore the vast literary ocean. Book recommendations are essential navigational aids in a world full of options, opening doorways to new concepts and realms.

Many people today get book recommendations by signing up for new book alerts for authors whose books they've enjoyed in the past, by following new releases in genres they enjoy, or by hearing about them via word-of-mouth social media sources like tik-tok. These discovery mechanisms aren't tailored to specific users and don't take into account user preferences across genres. They can also act as an echo chamber since they don't encourage users to branch out and potentially discover new or less popular books they may have otherwise overlooked.

Given these shortcomings, AI has the potential to play a key role in revolutionizing book recommendations. By combining analysis of vast datasets of books and user behavior, AI can generate user-focused recommendations in a way that would be impossible to replicate by mechanisms like crowd-sourcing. Recommendation algorithms, powered by AI, utilize machine learning models to understand individual preferences, reading patterns, and genre affinities. AI algorithms can identify intricate patterns and correlations, ensuring more accurate and diverse recommendations. For example, someone who primarily reads books in the fantasy genre might only follow authors in that genre and social-media accounts that recommend books in that genre. With AI, this user could discover books outside of that genre that still match their reading preferences. In this way, the use of AI in recommendations not only facilitates efficient discovery but also contributes to a more dynamic and engaging reading experience.

Goodreads itself does provide recommendations tailored to user based on their past reading behavior and items they've placed on their shelves (Chung 2011). It makes decent recommendations however, unlike our system, it allows authors and publishers to pay to push recommendations to users (Cerézo 2022). This muddies the waters of which books are being recommended to a user because they're good fits and which are being recommended simply because the author could afford it.

In this paper, we will discuss how we built MobyPick. We first describe our cloud architecture, including some tradeoffs that we made to reduce our overall cost of running the application, given that it is not currently in production. We elaborate on decisions that we would make if cost were not a factor and we required robust scalability. We then discuss the data that we used for generating recommendations. Finally, we discuss how we organized both the dataset and our application's data to ensure speed and efficiency.

Cloud Architecture

In this section we discuss the AWS services and tools that we made use of in order to build MobyPick, why we made those decisions, and how each service integrates into the overall system. We touch on the data storage systems we used but outline the details of them in further detail in the following section. We also discuss the tradeoffs that motivated some of our design decisions, and elaborate on the decisions we would have made if cost were not a consideration.

AWS Services

In building our application, we focused on a cloud-first architecture that could provide cost-effective scaling and high-availability to our users. Cloud computing and a cloud-first architecture are pivotal to meet users' expectations of responsiveness, availability, and feature sets, offering scalable and on-demand access to computing resources and easy-to-orchestrate application tools. Cloud computing, and in particular AWS's rich and integrated feature offerings leave room for developers to innovate and build cost-effective solutions.

- **Cognito** is used for user authentication, access management, and access control. To view anything in the application, users must first sign up and log in to the system. To sign up, users must enter various details, such as email address, full name, date of birth, gender, username, and a secure password. The signup process triggers an email containing a verification code for added security. Cognito also generates a userID and stores this along with user data in its internal data-store.
- **Elastic Beanstalk** is used to host our web application built on Django. Elastic Beanstalk simplifies hosting by abstracting away configurational settings such as having to create your own virtual environments or load balancers, thus making the web application easy to run and scalable. Inside Beanstalk we configured the environment with EC2 instances (minimum of 1 but can scale up to 4) and load balancers to support a large number of users and requests. The Beanstalk environment was configured as a Python 3.11 environment, and using the requirements.txt file in the project, it installs the dependencies required to have the Django web server up and running.
- **S3** is used to store the processed data from the dataset. We also plan to use it for archiving data from other databases as they grow.
- **AWS Personalize** is used to build the recommendation engine. which is a fully managed machine learning service that enables developers to create tailored recommendations for their users. To create our solution in Personalize, we created a custom dataset using the book, user, and interaction data from the GoodReads dataset hosted in an S3 bucket. We picked the personalized user recommendation recipe to train the model. The trained model is then invoked via API calls to generate recommendations, The model also contains filters to create recommendations when certain parameters are missing, with the user ID being the only required parameter, The Personalize campaign is also set up to react to real-time events, such as the user liking a book. Personalize allows developers to build recommendation engines even without deep machine learning expertise.

  The training is entirely abstracted by Personalize, which means that as developers we were able to focus our efforts on building the site and improving the quality of our data rather than on machine learning models and algorithms. Once the model has learned a user's preferences it's able to generate new book recommendations. For each user, we kick start Personalize's learning process for that user by presenting them with a ”book picker” game when they first sign up. As users indicate that they've read a particular book, we send a notification of that event to Personalize which allows it to continue learning and improving its model.
- **RDS** stores the complete book information. Since it is a relational database, it enables easy and fast access. We chose to store book information here since it is utilized by various functions and services. Moreover, attributes of books such as title, and author can be considered constant.
- **ElastiCache** is a serverless caching service that sits in front of our RDS instance. It stores frequently accessed data, in our case popular books, for extremely fast response times.
- **DynamoDB**, a NoSQL database service is used to store all of the user details and their preferences. Currently, the application only provides parameters such as language and genre, thus, this data must be stored in a database that is robust to changes in the schema. DynamoDB fits this criteria perfectly.
- **Lambda** functions are used to implement interaction logic between different components. Our application has 4 major lambda functions -
  - getRecommendations: Interfaces with AWS Personalize and necessary databases to generate recommendations for the users based on userID, and optionally genre and language.
  - getRecommendationsLex: Provides slot validation for lex and generates recommendations.
  - periodicRecommendations: This is triggered periodically by Eventbridge to email users with new recommendations on a weekly basis.
  - processChatbot: Used to interface between the frontend and the chatbot and set required SessionAttributes.
- **Lex** helps provide chatbot functionality to the application. Lex uses NLP to recognize text that the user inputs and replies accordingly. For our purpose, we have defined two custom intents.
  - GreetingIntent: This is invoked when a user starts the conversation. We add sample utterances such as hi, what's up and others to train the bot and recognize this intent.

- BookRecommendIntent: This is invoked when a user asks for a book recommendation. We define four slots to achieve this. We have a slot to store the value of genre and another one to store language. However, since these parameters are optional we also have 2 additional slots to check if the user wants to provide these parameters. A lambda function is used to validate these slot values again a set of valid values in our data and provide recommendations.

- **API Gateway** acts as a layer in between our Django application and the lambda function that we're using to retrieve data Personalize and RDS. This allows us to decouple the backend of our application, which is concerned with retrieving and presenting data to the user, from the part of our application that is responsible for communicating with Personalize. We were able to iterate on and test the two pieces independently, and we could re-use logic for both the frontend integration with Personalize as well as the Lex integration.. Using API Gateway also allowed us to implement caching, so that repeated requests from a user for recommendations could be send back immediately. Requests can be cached on user ID, language, and genre.
- **SES** is used as the emailing service by the periodicRecommendations Lambda function, to send recommendations to a user.

API Endpoints

This complex application utilizes different API endpoints (listed below) to ensure a smooth user experience.

- **Login**: This is called by the frontend to connect with Cognito and provide user authentication.
- **Sign Up**: Allows new users to create their profiles. User data is then stored in Cognito and DynamoDB.
- **Profile**: The profile endpoint allows users to update their preferences. This data is then stored in the relevant DynamoDB table.
- **Show Recommendations**: This connects the frontend to the getRecommendations lambda function to generate recommendations.
- **Chatbot**: Connects frontend to Lex to provide chat functionality.

Workflow

A user accesses the website through the Beanstalk-hosted URL. The first page is the login page. User has two options - sign up for new users or login for returning users.

First-time users - You land on the sign-up page where you are required to enter details such as name, email, preferred name, and password. Once you have validated your email, you get logged into the app. As a first-time user, the system has no information about you to tailor your recommendations. To counter this, we have come up with a book-picker game. To set a baseline for your preference, new users select certain books out of a given list of popular books across a variety of genres. Once you submit a list of your favorite books, you are taken to your profile page where you can select your preferred language and genre.

Returning users, upon login are redirected directly to this login page. Post this point, workflows converge for both new and returning users. On the login page, users have multiple options -

- Update Preferences: Users can change the language or genre they prefer.
- : Get Recommendations: Users can get up to 10 recommendations. The recommendations are displayed like a carousel with the following options for each book:
  - Go to previous/next picture
  - Add it to 'read book' shelf ('check')
  - Add it to 'want to read book' shelf ('plus')
- Talk to Ahab: Users can get their recommendations through a conversation with a Ahab, MobyPick's chatbot, without having to enter details manually using the chatbot. The chatbot provides 3 recommendations, with each book title hyperlinked with its corresponding GoodReads URL.
- View 'read book' shelf: You can look at books you have already read.
- View 'want to read book' shelf: You can look at the books on your reading-list. You also have the option to move books from the want to read shelf to the already read shelf. This would also trigger an event to Personalize so that it can continue to learn about the signed in user.

Tradeoffs

We had to make several decisions to limit the cost of using AWS services. The GoodReads dataset is huge and we wanted to work with the entire dataset to be able to train our model with more data. However, due to the storage limit on the free tier of AWS, we were only able to work with data from about 2000 books.

Setting up Personalize requires huge amounts of data processing and cleaning which could have been easily done using Amazon Sagemaker Data Wrangler, however, this is a paid service. As a result, we ended up using pyspark locally to complete this task.

Ideally, we would also want to index partial book data such as language and genres in Elasticsearch for fast search. On a small scale, this might not be of concern but will impact performance as MobyPick gains popularity.

We want to employ AWS Glue (another paid service) to provide data analysis and predict customer trends to be able to expand our database accordingly. AWS Glue would also provide us with data cataloging capabilities.

SES can send emails only to email IDs added to it in the development environment, which we are using. In production, we would need our own DNS records (which isn't possible on an Amazon hosted website), so as to allow any user that signs up to be able to receive emails.

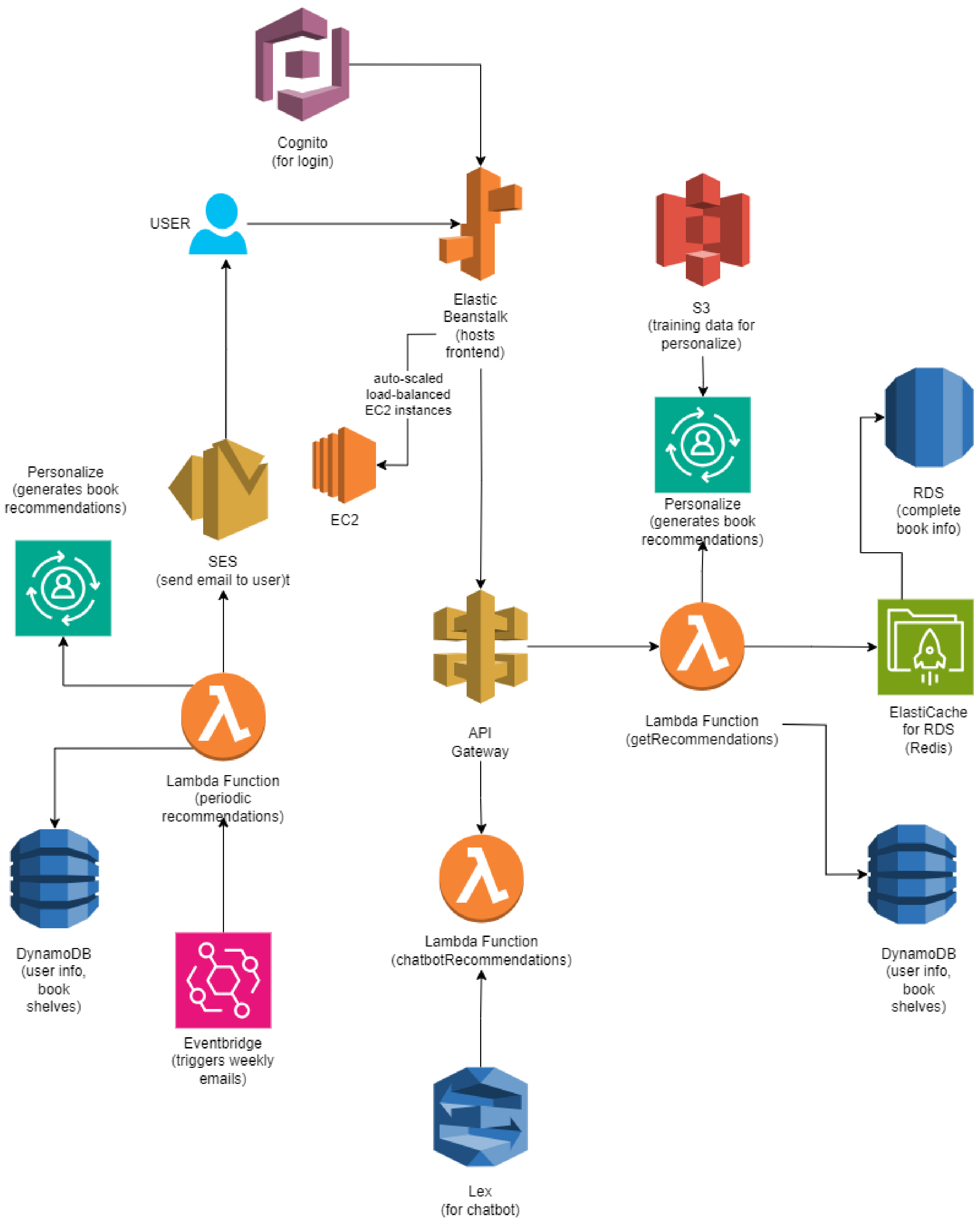

Figure 1: Architecture Diagram

Data

Goodreads Data

The datasets we used were collected in late 2017 from goodreads.com(Wan and McAuley 2018)(Wan et al. 2019). It contains scraped users' public shelves, (public meaning that everyone can see it on the web without login). The User IDs and review IDs have been anonymized. The dataset contains three groups: (1) meta-data of the books, (2) user-book interactions (users' public shelves) and (3) users' detailed book reviews. These datasets can be merged together by joining on book/user/review ids.

Data Cleaning

The raw Goodreads data needed to be cleaned before it could be fed into Personalize and loaded into mySQL. Many of the rows had key fields that were blank, like title and genre, or a media type that wasn't book. We started by filtering out those rows since our concentration is solely on book recommendations. There were also columns that contained primarily null fields, like language. These fields were removed. Apart from these, there were also unnecessary columns for our case like the book version and series which we removed. Finally, we truncated the number of books down to 2,000 by first sorting the data set on the number of ratings each book received, and then limiting our dataset to 2,000 rows.

Once we had a clean version of the dataset we used a user-defined functions to add language using a python package called langdetect which inferred the language based on the book's title. We then joined in the genre and author datasets, and used UDFs to format each of them to the format required by Personalize. In order for Personalize to be trained on a field, it must be a flat, non-textual field (i.e. not a long string that might represent an array of values or a description). Although UDFs are generally less efficient than built-in PySpark transformations, the manipulations we needed to do to extract the data from the Goodreads fields and into our format couldn't be done using out-of-the-box operations. UDFs still take advantage of parallelization however, given that they are sent to the worker nodes to operate on the data in parallel.

Pyspark was run on a macbook M1 pro with 16GB of memory, 1TB disk, and 10 cores. Even cut down to 2,000 rows, the notebook to generate the data for Personalize from the books took roughly 16 minutes and the notebook to correctly format the interactions data took roughly 13 seconds. Since this was a one-time operation we were using to kick-start Personalize training and test our application, this was sufficient and free.

Data Storage

After cleaning the data, we uploaded the CSV files to S3, DynamoDB, and mySQL. We uploaded the files to S3, as Personalize expects CSV files as inputs to generate recommendations. We also uploaded these CSV files to mySQL for fast rendering and low latency of data loading for requests from our Django application.

In our first implementation of the system we stored the table BookData in DynamoDB partitioned on ITEM_ID since ITEM_ID is the ID returned by Personalize that we would be using to scan the table to retrieve book details for recommended books. We chose this implementation at first in order to maximize our cost savings. However we were finding that book details were taking on the order of 16-17 seconds round-trip in the UI, clearly an unacceptable user experience. DynamoDB is not ideal for this kind of item retrieval since it's a key value store. We decided to switch to using RDS which, although it is not free like DynamoDB, could be configured to meet our needs while also being inexpensive.

We loaded the columns that the frontend would need to render book details into our RDS mySQL instance using mysqlimport and set its primary key as ITEM_ID. Again, this is the key that Personalize would be using to identify books so it was important that the table be indexed on that column. Because RDS builds an index on this field, retrieving the 10 records all at once is trivial. Retrieving the book details from RDS reduce the round-trip time to 1-2 seconds, a much better user experience.

We also added an ElastiCache Redis cache to sit in front of our RDS database. This ensures that the UI remains responsive even as our application scales to potentially billions of books. Especially given the traffic pattern that a small subset of books receive the vast majority of requests, having a cache will make sure these books can be retrieved and returned quickly and without overwhelming the database.

Cost Tradeoffs

Similarly to our architecture choices, there were some choices that we made in our data architecture due to the fact that this is intended to be a small proof-of-concept project and we were unwilling to rack up too large of a bill.

As mentioned, running PySpark locally would obviously not scale to the level a full productionalized application would require. If we needed to consume this data on a regular basis and were willing to bear the cost, we would want to use the DataWrangler feature of Sagemaker to dramatically simplify and speed up this ingestion process and allow us to load far more books into the system. DataWrangler automates many data cleaning processes like detecting null rows, renaming columns, and reformatting columns. Because it's integrated into the Personalize workflow, it automatically ensures that your data adheres to the Personalize specifications.

For the portion of our data cleaning and formatting jobs that DataWrangler is unable to handle, for example the language inference, we would load the data into Elastic File Storage and use Elastic Map Reduce to run the jobs.

With RDS, we disabled some of the more expensive features that make it highly-available, robust, and extremely scalable. We are not using an RDS proxy to connect to the database, but are instead establishing a normal connection. Using an RDS proxy would allow applications to pool together and share connection resources which avoids the overhead of opening a new database connection each time, lets the database handle unexpected surges in traffic, protects against oversubscription, and provides an additional layer of security. We are also currently only running in a single availability zone with a single replica. In a production environment, we would ensure that we had replicas in multiple AZs to ensure uptime and stability.

Conclusion

MobyPick makes use of a wide variety of AWS services, hand-picked to make it work for a large audience. As students, we understand the importance of reading and the value it brings, and we believe that anyone can find a love for reading if matched with the right books. Given the volume of data involved in cataloging books, and the vast number of book lovers, it was important to build this system in a way that could scale. We are happy to be the creators of MobyPick and we hope it enables people to develop a healthy reading habit and read a wider range of books.

References

Brysbaert, M. 2019. How many words do we read per minute? A review and meta-analysis of reading rate. *Journal of Memory and Language*, 109: 104047.

Cerézo, A. 2022. How Does Goodreads Make Money?

Chung, K. 2011. Announcing Goodreads Personalized Recommendations.

Streichler, R. ???? Reading Speed Charts. https://catalog.shepherd.edu/mime/media/10/911/SU+Credit+Hour+Policy+Appendix+B.pdf. [Accessed 10-12-2023].

Wan, M.; and McAuley, J. J. 2018. Item recommendation on monotonic behavior chains. In Pera, S.; Ekstrand, M. D.; Amatriain, X.; and O'Donovan, J., eds., *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*, 86–94. ACM.

Wan, M.; Misra, R.; Nakashole, N.; and McAuley, J. J. 2019. Fine-Grained Spoiler Detection from Large-Scale Review Corpora. In Korhonen, A.; Traum, D. R.; and Màrquez, L., eds., *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, 2605–2610. Association for Computational Linguistics.