Problem Statement

With millions of books available on Kindle, customers face challenges in discovering books that align with their interests. Unlike physical bookstores where recommendations come from store staff or curated displays, online bookstores **use recommendation systems** to assist users in book discovery through limited screen sizes. The **use case** is to deliver in real time, a ranked list of books to any Kindle user regardless of whether they are a returning customer with hundreds of purchases, a dormant user re-engaging after months of inactivity, or a brand-new user with no prior history. Further, empirical evidence show that recommendation systems play an **important** role in driving revenue and engagement in large-scale digital marketplaces. In particular, personalised recommendations are associated with a 29% increase in firm revenue, while highly relevant recommendations can further boost revenue by up to 30%, highlighting the economic importance of both recommendation presence and quality (Basu, 2021). Other large-scale empirical studies have shown that effective recommender systems significantly improve click-through rates, conversion rates, and overall revenue across global e-commerce platforms (Cilacı Tombuş et al., 2024). However, many existing recommendation models face key challenges, such as the cold start problem for new users, lack of personalisation for returning users, lack of granular customer segmentation, and lack of explainability and insights. From a business analytics perspective, improving recommendation quality is **useful** for optimising core performance metrics such as click-through rate (CTR), customer conversion rate. From a data engineering perspective, analytics driven optimisations are useful for allowing outcomes to be continuously measured, compared and refined. Addressing this problem therefore provides a practical testbed for designing end-to-end data pipelines that support reliable ingestion, transformation, aggregation, and storage of behavioural data at scale. From a machine learning system standpoint, recommendation systems are only useful if they can be retrained as user preferences evolve, evaluated via controlled experiments, monitored for performance degradation, and rolled back or updated safely when issues arise.

Taking a systems-level perspective, this project is designed to integrate stream ingestion, batch processing, feature engineering, model management, and continuous monitoring into a single closed-loop architecture. Rather than treating feature engineering and model training as isolated, one-off analytical exercise, I aim to embed machine learning within a continuously operating data platform that supports iteration, evaluation, and adaptation over time. By operationalising feature pipelines and models end-to-end, the system demonstrates how data-driven insights can be generated, validated, and deployed given constraints such as behavioural drift and evolving user preferences.

Dataset Information

I will be using the Amazon Reviews'23 dataset (filtered to Kindle products and reviews only), which includes features such as customer reviews (rating, title, review, product ID, customer_id, timestamp, verified_purchase, helpful_vote, images) and item metadata (main_category, title, average_rating, rating_number, features, description, price, store, categories, details, product IDs, bundles, images, videos) that are collected before September 2023. Customer review dataset and product dataset contains multiple millions of rows of data (customer: 5.6M, items: 1.6M, rating: 25.6M) Customer and Product dataset was collected from May. 1996 to Sep. 2023, ensuring the reviews and product data reflect the latest customer experiences and trends. I believe the dataset is well-suited to this project as it contains rich customer interaction data. More information on the suitability and EDA of the dataset can be found in Appendix B. The schemas for both sources are shown below.

Feature	Description	Type	Details
rating	User's rating for the eBook	Float	1-5 stars, reflecting the user's overall opinion
title	Title of the review	String	The summary provided by the reviewer
text	Full text of the review	String	The feedback or comments from the user
images	Images uploaded with the review	List	URLs or references to user-uploaded photos
asin	SQL of the reviewed item	String	Amazon Identification Number
user_id	Unique identifier for the reviewer	String	Anonymised ID to track user activity across reviews
timestamp	Date and time of the review	Datetime	Precise posting time, ranging from May 1996 to September 2023
helpful_vote	No. of votes from other users	Integer	Indicates how many found the review useful
verified_purchase	Whether the reviewer bought the eBook	Boolean	True/False flag confirming the user purchased it via Amazon

Figure 1: User Reviews Dataset

Feature	Description	Type	Details
main_category	Primary category of the eBook	String	Likely "Kindle eBooks" or a subcategory (e.g., Fiction, Nonfiction)
title	Title of the eBook	String	The full name of the book as listed on Amazon
average_rating	Avg user rating for the eBook	Float	1-5 scale, aggregated from all reviews
rating_number	Total no. of ratings received	Integer	Indicates how many users have rated eBook
features	Key features of the eBook	List	Could include bullet points.Sometimes can include the description of a book.
description	Detailed description of the eBook	String	The summary, often from the publisher or author
price	Price of the eBook	Float	In USD, may be 0.0
images	URLs to eBook cover images	List/String	Links to visual assets, typically the cover art
videos	URLs to related videos	List/String	book trailers or promotional content
store	Storefront identifier	String	Likely author or the store selling it.
categories	Subcategories or genres	List	E.g., "Mystery," "Romance," "Science Fiction"
details	Additional product details	Dictionary/JSON	Technical details of the product like Page count, publisher,etc.
parent_asin	Unique identifier for the eBook product	String	Amazon Standard Identification Number linking related items
bought_together	Items frequently bought with this eBook	List/String	ASINs (e.g., other books in a series)
subtitle	Subtitle of the eBook	String	Additional title info, mostly Kindle Edition
author	Author(s) of the eBook	String	Link to the author's page or image.

Figure 2: Product Details Dataset

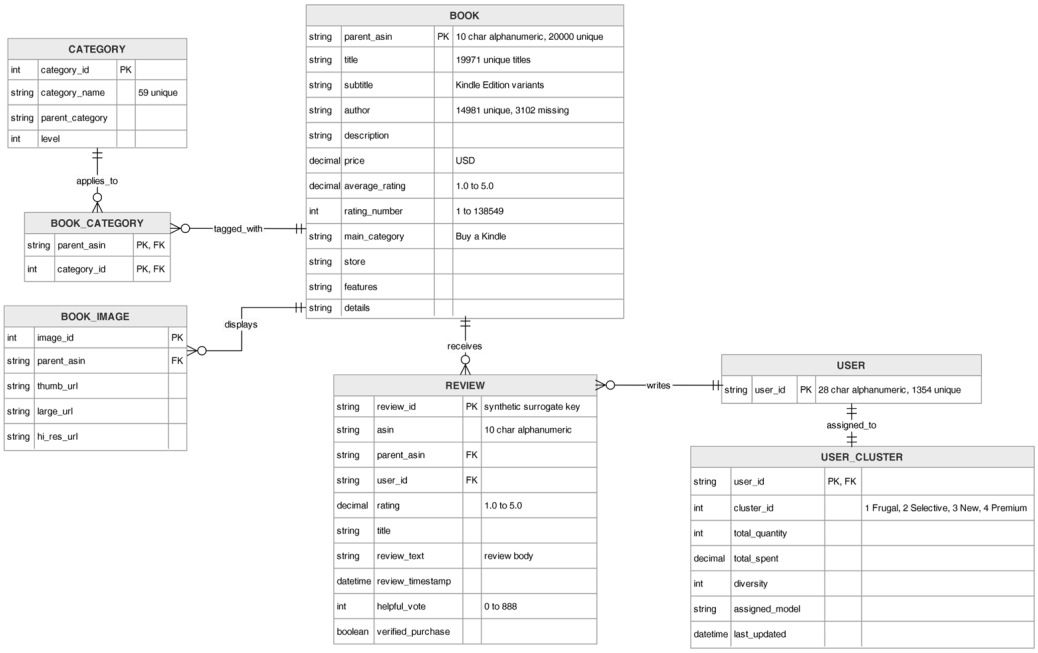


Figure 3: Entity Relationship Diagram that maps how Users relate to Books in BigQuery.

While I started with two datasets, some fields in both files are nested. Parsing them on every query would be expensive and would prevent us from indexing categories or computing co-purchase frequencies. Normalising them into separate tables (BOOK_IMAGE, BOOK_CATEGORY, BOUGHT_TOGETHER) makes the warehouse query-efficient and lets us reuse the BOUGHT_TOGETHER edges directly as the graph for the Random Walk recommender. The data warehouse is normalised to third normal form around the book and user entities. Many-to-many relationships between book-to-category and book-to-book co-purchase are resolved via junction tables (BOOK_CATEGORY, BOUGHT_TOGETHER) with composite primary keys. Two non-source tables bridge the warehouse to the ML pipeline. USER_CLUSTER stores the GMM-derived segment assignment per user, including the cluster ID, the four behavioural metrics that produced it, and the model the user is routed to.

I transferred the transformed Kindle Store reviews and metadata into Google Cloud BigQuery for persistent storage and downstream use. This step ensures the processed data is structured and accessible in a cloud-based data warehouse. The pipeline authenticates with BigQuery using a service account key. The transformed datasets are then uploaded to two tables

in the kindle_store dataset. This streamlined loading process ensures the data is efficiently transferred, structured, and stored, fulfilling the pipeline's objective of delivering clean, transformed data to a cloud destination.

ETL Pipeline

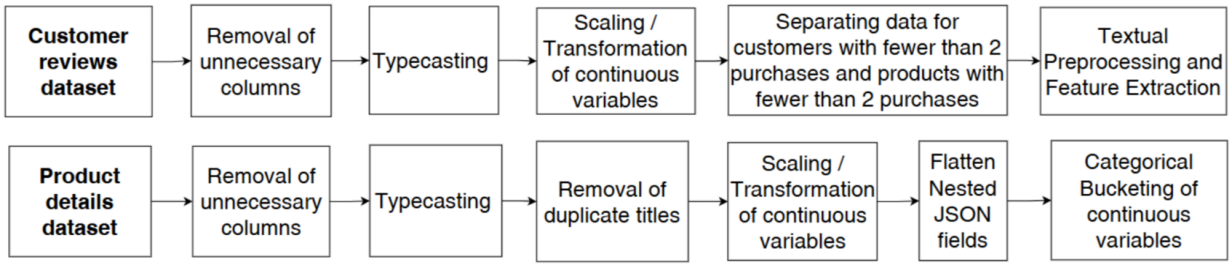


Figure 4: Transformation Component of ETL Pipeline. Additional details can be found in Appendix C.

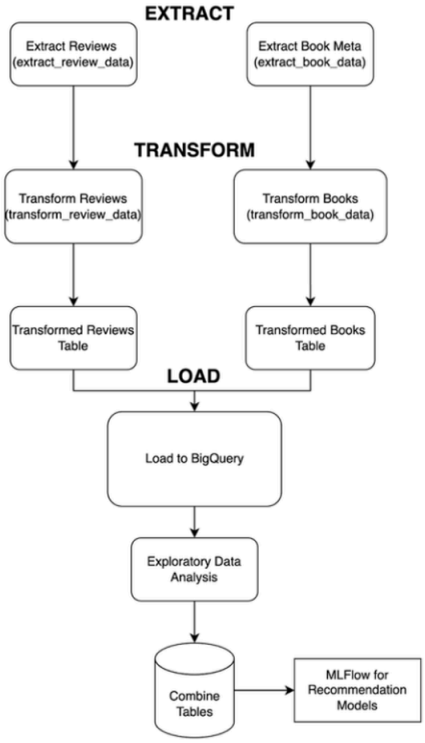


Figure 5: ETL Pipeline

The ETL pipeline is orchestrated by Apache Airflow and follows the classic extract-transform-load pattern. Reviews and book metadata are extracted in parallel from Hugging Face, transformed independently, then loaded into Google BigQuery.

Cleaning: Within the raw dataset, positive ratings dominate (60%+ five-star reviews), so I applied stratified sampling across the rating distribution to prevent positivity bias from leaking into model training. Missing values were handled per-field: helpful_vote had zero missing values; price contained 'None' strings which I imputed using the median price within each book's category, and I excluded rows where critical features (parent_asin, user_id, timestamp) were absent.

Column selection: I retained only columns relevant to behavioural and content modelling. From the Reviews table I kept user_id (customer identifier), parent_asin (book identifier), rating (interaction strength), timestamp (sequence order), text (raw review content for embeddings), and verified_purchase (credibility filter). From the Metadata table I kept parent_asin, title, categories, description, and features for content-based feature extraction.

Encoding: Categorical book categories were tokenized and embedded using a pre-trained BERT model to capture semantic relationships between genres (e.g. "Mystery" and "Thriller" sit closer in embedding space than "Mystery" and "Cookbooks"). The verified_purchase Boolean was one-hot encoded into a binary feature.

Loading: The transformed dataset is then loaded into Google BigQuery with BOOK and USER as the dominant tables of interest.

Feature Engineering

Building on the preprocessed Kindle dataset, I engineered features for our recommendation models. I applied Gaussian Mixture Models for soft clustering on three user metrics: diversity (distinct book categories), total spending, and total books purchased, capping spending at $1,000 and purchases at 100 to address sparsity. I identified four segments in the GMM plot: Frugal Shoppers (high purchase count, low spending), Selective Spenders (high diversity, moderate spending), New/Dormant Users (low activity), and Premium and Loyal Shoppers (high engagement). I separately defined Totally New Users (zero activity) as a distinct segment, addressed by Hybrid ALS + CBF's cold-start solution via CBF. Using Word2Vec, I generated review_embedding (300 dimensions) from review text and product_embedding (300 dimensions) from book category, title, and description, combining these with scalar features (rating, normalized_timestamp, normalized_helpful_vote, rating_number_scaled, price_scaled, verified_purchase) into a 606-dimensional vector per interaction. For Random Walk, I constructed a user-item graph with contextual edge weights (e.g., inverse time delta, ratings), enabling recommendations across all segments.

Customer Segmentation

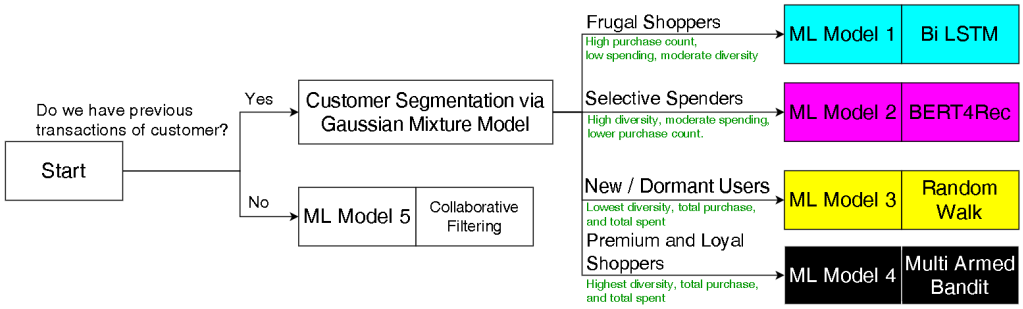
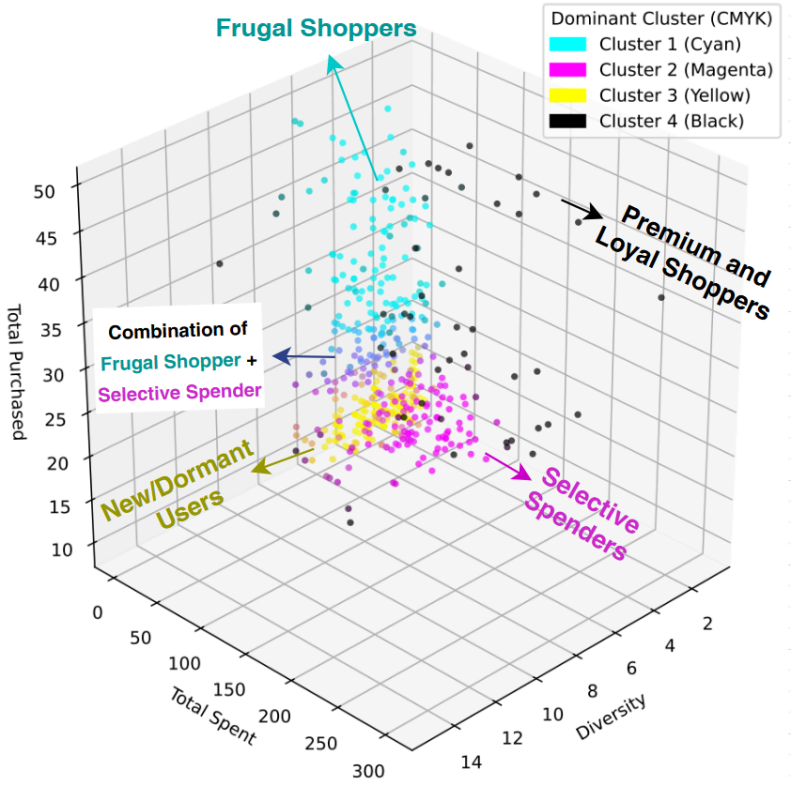


Figure 6: Pipeline for Customer Segmentation



user_id	cluster_1	cluster_2	cluster_3	cluster_4	dominant_cluster
AE22FPJQ6YSFA25LCQHUA4QLMNBA	0.10785496	0.84199289	0.00150594	0.04864621	2
AE22WTKMX7A2FMFIF36XBJ5YVENQ	0.05317554	0.13251528	0.81189809	0.00241109	3
AE22YNFWCWI4ENWO5I4OKG2SWT4Q	0.78225938	0.01422514	8.08E-13	0.20351548	1
AE2KDXED4Z7XVYL6TXZMO3GJ7QWQ	2.00E-25	6.16E-31	2.64E-205	1	4
AE2NFRAOKNX4NMRJPH3BHH7VZOUQ	0.17274956	0.36293257	0.45880119	0.00551668	3

Figure 7 (left): Users clustered through Gaussian Mixture Model with cluster size of four, with Diversity as the x-axis, Total Spending as the y-axis, and Total Books Purchased as the z-axis. Each point is coloured using CMYK, proportional to the probability of belonging to each of the four clusters, so lavender is a combination of clusters 0 and 1. The legend anchors each cluster to its dominant base colour.

Figure 8 (top): Example of each user id tagged to a probability for each cluster, where the sum of all four clusters is 1.

Using Gaussian Mixture Models, I did soft clustering with respect to three fields - Diversity (the number of distinct book categories), total spending per user and total number of books purchased per user. Unlike hard clusters such as K-Means, soft clusters provide elliptical, overlapping decision boundaries, that can handle natural overlapping to capture other hybrid variations of customers, and are well suited to handle right skewed, continuous data. Further, due to the sparsity of users spending above 1000 USD, and users purchasing above 100 books, I set a maximum cap to both parameters.

From the GMM output, four key behavioural segments were identified, each aligned with a CMYK-coded cluster to reflect distinct user groups. **Cluster 1** in Cyan consists of Frugal Shoppers. These users have high purchase count, low spending and moderate diversity compared to the other clusters. These users seem to be highly budget-conscious and aim to maximise utility per spending (even more than new or dormant users). Frugal Shoppers are more likely to purchase Kindle books that are free, on discount or of low price, and price seems to play a huge factor in their behaviour and decision making. To date, we have 22.2% of users with the dominant trait of a frugal shopper. **Cluster 2** in Maroon consists of Selective Spenders. These users have high diversity and moderate spending but a lower purchase count. Their selective purchasing behaviour reflects a quality-over-quantity mindset, and they may benefit more from curated and quality-focused consumption. To date, we have 28.9% of users with the dominant trait of a selective spender. **Cluster 3** in Yellow consists of New or Dormant Users. These users have the lowest purchase count, spending and diversity compared to the other clusters. These users have minimal interactions with Amazon's Kindle platform, and they are critical targets for activation or re-engagement strategies, and they may benefit more from selective exploration to find the Kindle categories that suit their taste most. To date, we have 36.1% of users with the dominant trait of a new or dormant user. **Cluster 4** in Black consists of Premium and Loyal Shoppers represented by the highest purchase count, spending and diversity. These users show high engagement, and are the most valuable customers to the Kindle Platform compared to the other cluster. To date, we have 12.8% of users with the dominant trait of a premium and loyal shopper.

Soft clustering via GMM also further reveals users with mixed memberships that may have some form of combined behaviour. One common hybrid is the intersection of Frugal Shoppers (cyan Cluster 1) and Selective Spenders (maroon Cluster 2) as shown in the figure above. These users exhibit a blend of high diversity and low to moderate spending. Compared to pure Frugal Maximisers, they may spend slightly more per item but purchase fewer items overall. Compared to Discerning Dabblers, they retain the high diversity but reduce the average cost or frequency of their purchases. Such a hybrid profile can indicate a user who is a combination of adventurous and value-driven yet selective, and such a user can benefit from the combination of personalised promotions that weighs affordability with novelty.

I also initially considered using some derivative of Latent Dirichlet Allocation (LDA), that was developed for discrete, count based data especially word frequencies, but I found GMM to be more aligned with right skewed, continuous variables such as total spending and diversity. This could extend conveniently into the future when more users have bought more items and I do not need to extend the axes to account for them.

Hyperparameters for GMM included setting covariance_type to `full` to allow for flexible modelling of cluster shape and orientation. Since there were no numerical instability and convergence issues, 1e-6 was used for regularisation (to prevent singular matrices) and 1e-3 was used for convergence threshold (to ensure a more precise convergence). Maximum iteration was increased to 1000 so the model can converge. The number of components was set to four.

Machine Learning

Cluster 1: LSTM (Frugal Shoppers)

Frugal Shoppers exhibit price-driven sequential behaviour: they wait for discounts, batch-purchase during promotions, and return to similar low-price genres. An LSTM is a natural fit because it maintains a hidden state that evolves across the user's purchase sequence, letting price react contextually rather than as a static feature.

There are three layers: Embedding layer thatr converts categorical fields (user_id, parent_asin, categories, author) into dense 64-dimensional vectors. Title text is separately TF-IDF encoded. LSTM layer with hidden_size=128, single layer, processes the concatenated feature sequence and maintains temporal state. Fully connected layer that projects the final hidden state to a scalar predicted rating.

Numerical features (rating, average_rating, rating_number, price, days_since_publication, print_length) feed in alongside the embeddings. Crucially, price is included as an explicit feature rather than being abstracted away — this is what lets the model learn frugal-specific price-elasticity signals. Timestamps are used for sequence ordering but excluded from training to avoid leakage. A single LSTM layer with hidden_size=128 was chosen over deeper or bidirectional variants for two reasons. First, frugal users' purchase patterns are predominantly forward-causal (a discount triggers a purchase; later behaviour does not retroactively explain earlier choices), so bidirectional context offers little gain. Second, deeper stacks overfit on this segment's relatively short median sequence length.

Cluster 2: BERT4Rec (Selective Spenders)

Selective Spenders explore broadly but buy carefully, and their sequences look like [SciFi_A, Thriller_X, SciFi_B, Cookbook_Y, SciFi_C] rather than the genre-locked patterns Frugal users show. Unidirectional models (GRU4Rec, SASRec) can only condition each book on prior context, missing the bidirectional taste signal a Selective user actually exhibits. BERT4Rec's deep bidirectional self-attention captures these patterns by letting every book in the sequence attend to every other book.

The architecture includes stacked transformer blocks with three embedding sources combined at the input. First, the Embedding layer book embeddings + category embeddings + position embeddings, summed before entering the Transformer stack. Next, Transformer layers (L stacked) where each block applies multi-head bidirectional self-attention with residual connections and feed-forward sublayers. And last, Output projection predicts the masked item via a softmax over the book vocabulary.

Rather than next-item prediction (which would leak future context in a bidirectional model), BERT4Rec uses BERT's masking strategy: random books in the sequence are replaced with [MASK] and the model is trained to predict them from surrounding context. This is the architectural innovation that makes bidirectionality tractable for sequential recommendation. I used a chronological 80/20 train-test split rather than k-fold cross-validation. Selective Spenders' preferences evolve over time, so shuffled folds would leak future taste into the past and inflate metrics unrealistically. The temporal split mirrors deployment: train on history, predict on the future.

Cluster 3: Random Walk (New / Dormant Users)

New and Dormant Users have too few interactions for matrix factorisation or neural models to learn personal embeddings, but they typically have *some* purchases, enough to seed a graph traversal. Random Walk fits naturally because the recommendation problem becomes "given this one anchor book, what are 24 contextually related books past users have walked toward?"

Each book is a node. Edges are weighted by a contextual payload combining four signals: genre similarity, price proximity, average user rating, and inverse time delta (recent co-occurrences weighted higher). This is a directed temporal graph rather than a static co-purchase graph. However, pure random walks are surprisingly *exploitative*, given enough iterations they converge to high-degree nodes. To prevent recommendation echo chambers for users who are actively exploring, I layered a Contextual Multi-Armed Bandit on top. Each node's outgoing edges are treated as bandit arms; arms are scored by their context (inverse time delta + rating + price), and a global noise constant injects stochasticity. This lets dormant users discover "rabbit holes" that pure walks would never surface.

Cluster 4: Greedy-ε MAB (Premium / Loyal Shoppers)

Premium users have rich, predictable interaction histories — exactly the regime where exploitation pays off. They've already revealed strong preferences, and aggressive exploration (which would help a sparse-history user) just degrades their experience by surfacing irrelevant items. The Multi-Armed Bandit framing lets us tune the exploration rate explicitly via ε.

Using greedy ε-bandit with ε = 0.05, I do 95% exploitation and 5% exploration. Each book node's outgoing edges are arms. The reward function is item price (proxy for high-value transactions, since loyal users are price-tolerant). Contextual signals (time delta, average rating) supplement price so that exploitation isn't purely cost-driven. I tuned ε empirically across {0.01, 0.05, 0.10, 0.20}. Below 0.05, recommendations stagnate and users see the same long-tail items repeatedly. Above 0.10, exploration dilutes the high-reward signal and weighted hit rate drops. ε = 0.05 was the sweet spot, enough exploration to refresh the feed, not enough to disrupt it.

Upper Confidence Bound would have been a more principled choice in pure ML terms, it adapts exploration to per-arm uncertainty rather than using a fixed rate. I rejected it for a business reason: UCB explores aggressively in early stages, allocating recommendations to lower-priced items simply because they're under-explored. For Premium users this conflicts directly with the monetisation goal of surfacing high-value items consistently. The greedy-ε approach is a slightly less elegant ML choice but a substantially better business choice for this segment.

Model 5: Hybrid ALS + CBF (Cold-Start Users)

The cold-start problem in book recommendation systems arises when new users, such as our Totally New Users segment with zero purchase history, spending, or engagement, join the Kindle platform, making it challenging for collaborative models to infer preferences or generate relevant suggestions. To tackle this, our Hybrid ALS + CBF model combines Alternating Least Squares (ALS) for collaborative filtering with Content-Based Filtering (CBF). ALS performs matrix factorization on the user-item interaction matrix to learn latent features representing user preferences and book characteristics, excelling with large datasets and implicit feedback like purchases but struggling with users lacking interactions. CBF addresses this gap by leveraging product_embedding to recommend books based on intrinsic features, requiring no user history. Initially, CBF suggests a single, highly popular book with favorable attributes (e.g., popular genres, high ratings) to engage Totally New Users, enabling immediate interaction. As users engage, ALS updates their latent profiles by comparing interactions with similar users, gradually shifting to personalized collaborative recommendations over time, ensuring a seamless transition from content-driven to user-specific suggestions.

The architecture includes two parallel components scored and combined. First, the CBF component (Feature Compressor). Linear layer with ReLU activation. Input: 5 scalar features + 300-dim Word2Vec product embeddings + 600-dim review embeddings. Hidden state 256 → output 128 dimensions. Dropout 0.3, learning rate 0.001, 15 epochs. Sequence aggregation by averaging produces a 128-dim user representation. Scoring via cosine similarity against item embeddings. Next, the ALS component. 50-dimensional user and item embeddings learned via Alternating Least Squares on the user-item interaction matrix. Scoring via dot product. Final score = 0.5 × ALS + 0.5 × CBF. Equal weighting is deliberate during early evaluation; once production data accumulates, the weights can be re-tuned per-user based on interaction count (more interactions → higher ALS weight).

Model Architecture

Backend Architecture

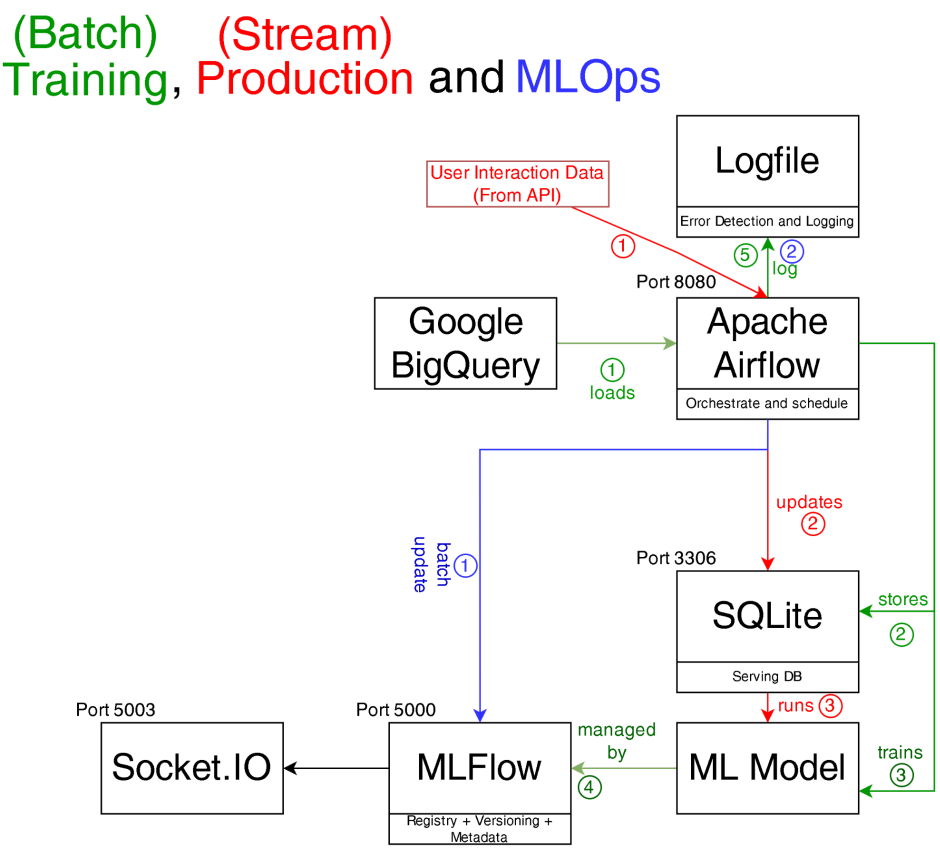


Figure 14: Pipeline for Training, Production and MLOps

Batch Training (Green Numbered Arrows)

① Airflow pulls historical and large-scale datasets from BigQuery to initiate scheduled batch jobs.

② Processed outputs or serving-time features are stored in SQLite for quick lookup during inference..

③ The ML model is trained using batch-processed data generated from BigQuery → Airflow pipeline.

④ After training, Airflow registers new model versions and artefacts in MLflow for lifecycle tracking. MLflow handles model versioning, metadata, reproducibility, and act as single source of truth for deployable models.

⑤ Log Training Issues such as NULL column checks and model convergence warnings.

Stream Production (Red Numbered Arrows)

① Real-time user interaction events are streamed via Kafka and made available for downstream processing and analytics.

② Processed or aggregated results from streaming data are periodically written by Airflow into SQLite for fast access

③ The serving layer reads lightweight, up-to-date application data from SQLite during model inference.

MLOps (Blue Numbered Arrows)

① After training, Airflow registers new model versions and artefacts in MLflow for lifecycle tracking.

② MLflow handles model versioning, metadata, reproducibility, and act as the single source of truth for deployable models.

Frontend Architecture

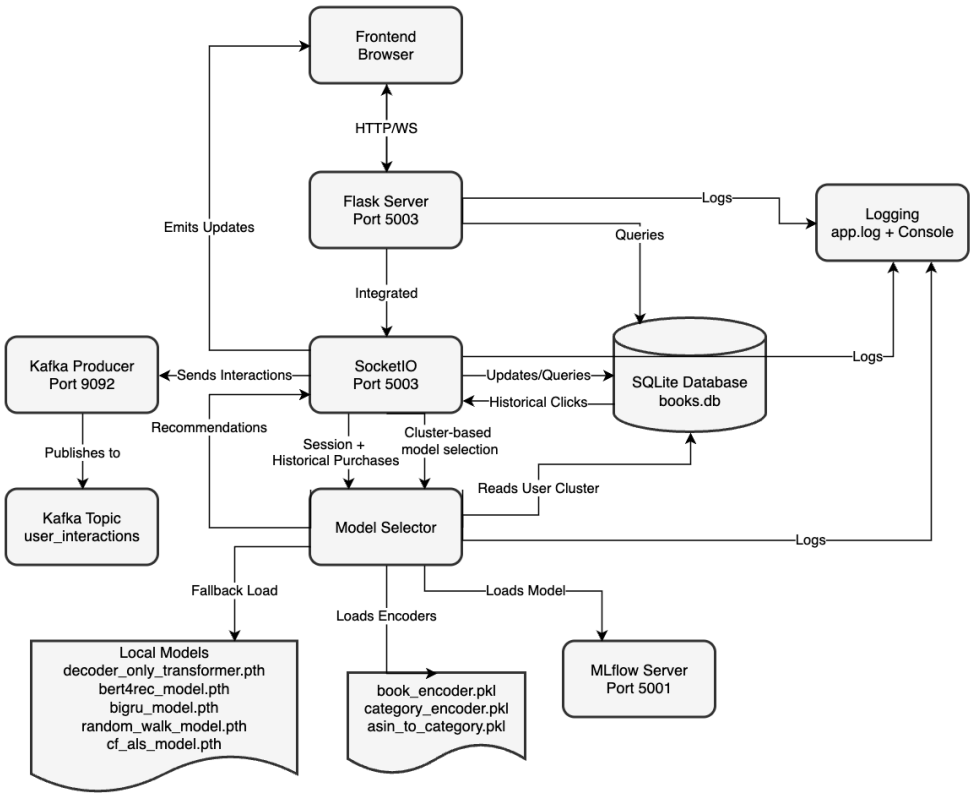


Figure 15: Book Recommendation System Front End Architecture

Stage	Component	Description
User Interaction	Frontend (Browser) → Flask Server → Kafka Producer → Kafka Topic (user_interactions)	User purchases are captured at the frontend, sent to Flask via HTTP/WebSocket, then forwarded to Kafka for publishing to the topic.
Real-Time Processing	SocketIO Server (with Flask) → Kafka → SQLite (books.db)	Consumes interaction data from Kafka, queries historical clicks and user clusters from the SQLite database, and prepares data for recommendation.
Model Selection	Model Selector → MLflow Server / Local Models	Processes encoded click data, selects and loads recommendation models (from MLflow or local fallback), and generates recommendations based on clusters.
Recommendation Delivery	Kafka → Flask Server → SocketIO → Frontend	Recommendations (top_books) are sent back via Kafka, consumed by Flask, and delivered to the frontend via SocketIO.
Logging	Flask Server & SocketIO Server → Logging System (app.log + Console)	Logs from the Flask and SocketIO servers are sent to the logging system for monitoring and debugging.

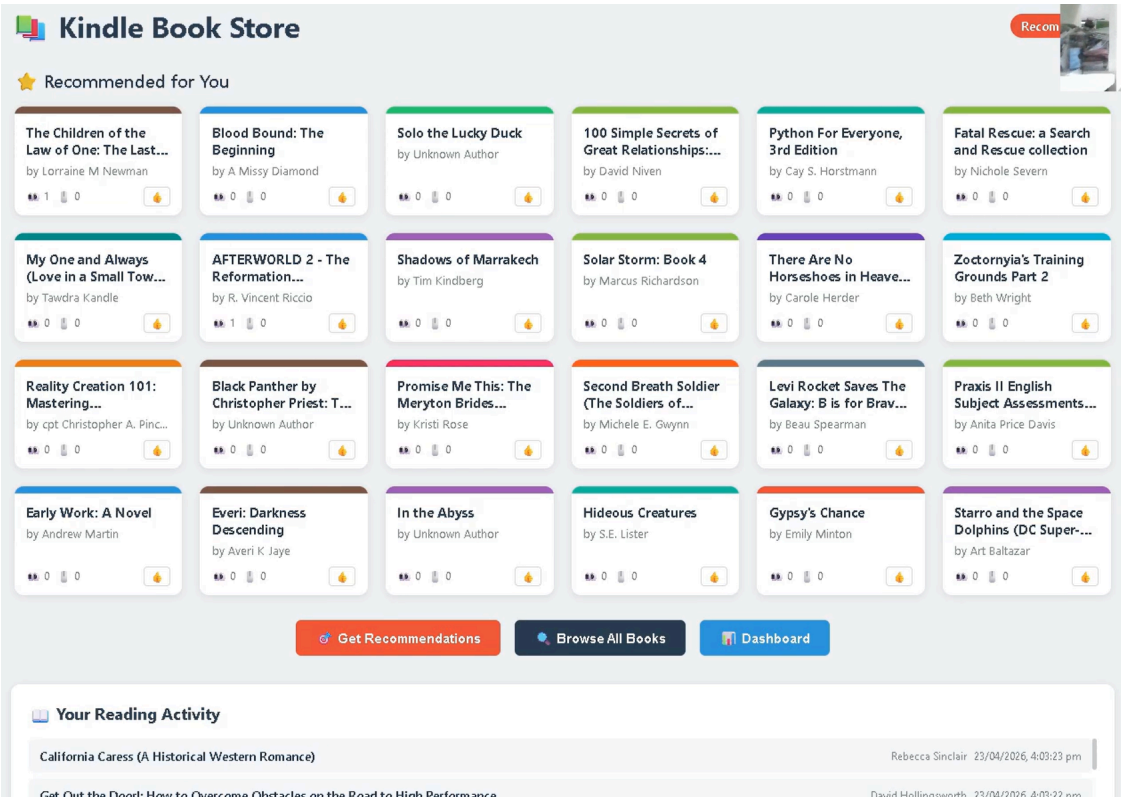


Figure 16: Model Interface

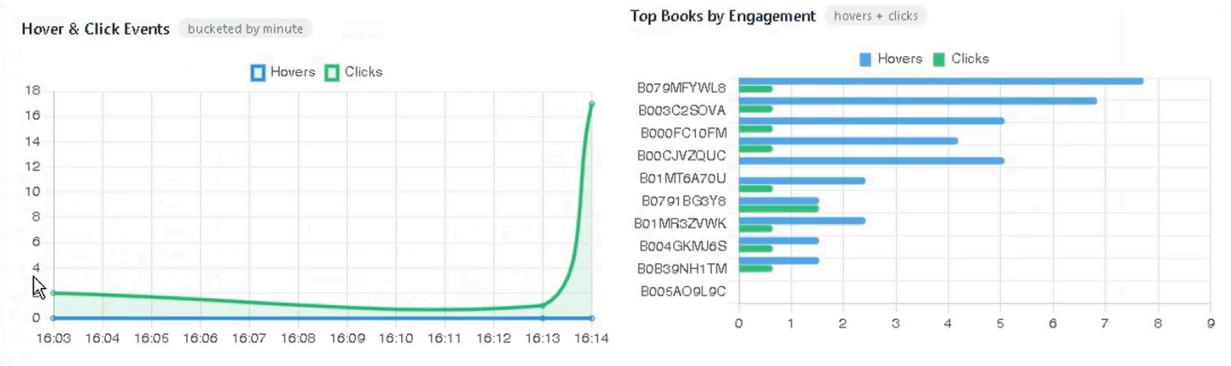


Figure 17: Data Dashboard

DataOps and MLOps

The DataOps and MLOps stack rests on three pillars: **BigQuery** for warehoused data, **Docker** for environment isolation, and **MLflow** for experiment tracking and model versioning.

Versioning and reproducibility. Every model training run is logged in MLflow with its hyperparameters, training metrics, environment dependencies (pinned in requirements.txt), and resulting artefacts (.pth model files). This means any past model version can be reproduced bit-exactly by rebuilding the Docker image at the recorded git SHA, restoring the BigQuery snapshot, and rerunning the training script with the logged parameters. Twenty-five experiments are currently tracked, organised by RecSys_C{n}_{model} naming convention.

Containerisation. Each of the five models has its own Docker image (mlops-bert4rec, mlops-lstm, mlops-random_walk, mlops-als, mlops-mab), each with its own pinned dependency set. Containers are orchestrated by docker-compose up -d, which spins up all five training services in parallel against a shared MLflow tracking server on localhost:5000. Pinned dependencies guarantee that every collaborator and every CI run produces identical training behaviour.

Streaming versus batch boundary. The pipeline is intentionally hybrid: batch processing (Airflow DAG, weekly cadence) handles the heavy lift, refreshing GMM cluster assignments, retraining all five models, and updating BigQuery. Stream

processing (Kafka + SocketIO) handles real-time interaction capture and recommendation serving. The two communicate through SQLite, which acts as the materialised view of the latest batch outputs. This separation lets us scale them independently: model training happens overnight on bulk hardware, while serving runs continuously on lighter infrastructure.

Observability. Every Flask request, SocketIO event, and Model Selector decision is logged to app.log and the console, with structured fields for user ID, cluster assignment, model invoked, and recommendation latency. This logging is the foundation for downstream A/B testing and model performance monitoring.

Evaluation

One of the main problems is the inference time of the models are different. Users that are routed to BERT4Rec (28.9% of them) will require a larger amount of compute. This is mitigated by the segment of users that are routed to a cheaper model such as Random Walk (at 36.1%) that are in the new / dormant cluster. Across all clusters, the segment-routed ensemble achieves a weighted-average Hit Rate @ 24 of 0.0386, compared to a single-model baseline of 0.0344 for BERT4Rec applied to all users. This represents a **12.2% relative improvement** from segmentation routing alone, suggesting that there is indeed some value in cluster based routing.

One of the main problems encountered was class imbalance in features, and feature absence across all models. For example, hyperparameters such as exploration, exploitation will not be existent in Bert4Rec compared to MAB or Random Walk. All models also use different features that aim to maximise their scores, and therefore, direct comparison between models amy not be theoretically precise.

Another main problem that was mitigated was the cold start problem recommendation engines face. Pure alternating least squares cannot generate user embeddings without prior interactions, so there is a need to hybridise it with content-based filtering using product embeddings. This is why Cluster 5 uses CBF+ALS rather than ALS alone.

After deliberating for 9 weeks, I took the risk to pivot away from my current group with only 5 days left until project submission. This is the best I can produce in 5 days as an individual with no external help. More could have been done in terms of visualisation and clarity but I was unable to debug it in time before submission. Training the Machine Learning Models was time consuming, and was a huge bottleneck in the rush to project submission.

Conclusion

This project demonstrated that segment-aware recommendation routing, where users are first soft-clustered by Gaussian Mixture Model and then dispatched to a specialised model. The four behavioural segments we identified (Frugal, Selective, New/Dormant, Premium) each respond best to a different model family empirically.

The technical contribution is twofold. First, on the data engineering side, we built a closed-loop architecture that combines stream ingestion (Kafka), batch ETL (Airflow + BigQuery), feature engineering (Word2Vec + GMM), model training (Docker + PyTorch + scikit-learn), experiment tracking (MLflow), and real-time serving (Flask + SocketIO + SQLite). Each layer is independently scalable, observable, and reproducible. Second, on the machine learning systems side, we showed that a heterogeneous ensemble of five very different model families (transformer, recurrent, graph-traversal, bandit, matrix-factorisation) can be served behind a single routing layer with sub-second latency.

The broader implication is that recommendation quality and data engineering are inseparable: without reliable ETL, version-controlled models, and logged outputs, no model improvement can be measured, attributed, or rolled back safely. The infrastructure is the substrate that makes iterative improvement possible, and the segmentation-driven routing is the leverage point that makes the iteration worthwhile.

In relation to Kindle’s business operations, this project can provide the data infrastructure and analytical tools to optimize multiple aspects of the platform's ecosystem.. Marketing teams can now identify and promote emerging trends with confidence, shifting from reactive campaigns to proactive, data-driven strategies. Further,, the solution creates a virtuous cycle where improved recommendations lead to higher engagement, which in turn generates more data to further refine the system's accuracy.

References:

Basu, S. (2021). Personalized product recommendations and firm performance. Electronic Commerce Research and Applications, 48, Article 101074. https://doi.org/10.1016/j.elerap.2021.101074

Cilacı Tombuş, A., Eroğlu, E., & Altun, I. H. (2024). Impact of recommender systems in e-commerce: A worldwide empirical analysis. Journal of Innovative Science and Engineering, 8(2), 251–265. http://jise.btu.edu.tr/en/pub/issue/88513/1308353

Appendix B: Dataset Description

Customer Reviews Dataset

Feature Name	Description	Values
Author	represents the name of the author, including information about the author.	Unique count: 14981 Number of missing values: 3102
Average_rating	represents the average rating of the product shown on the product page (out of 5).	Values range from 1.0 to 5.0. Number of missing values: 0
Bought_together	represents the other books frequently bought together with the product.	Number of missing values: 20000
Categories	represents the list of all product book categories.	Top 3 Most Frequent Categories: {"['Kindle Store' 'Kindle eBooks' 'Literature & Fiction']": 6486, "['Kindle Store' 'Kindle eBooks' 'Romance']": 1867, "['Kindle Store' 'Kindle eBooks' 'Religion & Spirituality']": 1573} Unique count: 59 Number of missing values: 0
Description	represents the description of the book content.	Unique count: 8757 Number of missing values: 0
Details	represents the miscellaneous product details.	Unique count: 19992 Number of missing values: 0
Features	represents the product features in bullet point format.	Unique Count: 19500 Number of missing values: 0
Images	represents the images of the product, with each having different sizes (thumb, large, hi_res).	Unique count: 19642 Number of missing values: 0
Main_category	represents the main category of the product.	Value_counts: {'Buy a Kindle': 19647} Unique count: 1 Number of missing values: 353

Parent_asin	represents the Parent ID of the product, which groups related editions.	10-character alpha-numeric identifier, typically formatted as a mix of uppercase letters and numbers. Unique count: 20000 Number of missing values: 0
Price	represents the price of the book in USD.	Top 5 Most Frequent Price: {'0.0': 9154, '2.99': 1527, 'None': 1106, '4.99': 1021, '9.99': 927} Unique count: 715 Number of missing values: 0
Rating_number	represents the total number of ratings received for the product.	Values range from 1 to 138549. Number of missing values:
Store	represents the store name of the product.	Top 3 Most Frequent Store: {'Charles River Editors (Author) Format: Kindle Edition': 26, 'BookRags (Author) Format: Kindle Edition': 15, 'Lucia Jordan (Author) Format: Kindle Edition': 11} Unique count: 18354 Number of missing values: 0
Subtitle	represents the subtitle of the product.	Top 3 Most Frequent Subtitles: {'Kindle Edition': 19057, '1st Edition, Kindle Edition': 197, '[Print Replica] Kindle Edition': 130} Unique count: 76 Number of missing values: 356
Title	represents the title of the product.	Top 3 Most Frequent Titles: {'Swept Away': 3, 'Dubliners': 2, 'Into the Deep': 2} Unique count: 19971 Number of missing values: 0
Videos	represents the title and URL of the product.	Unique count: 216 Number of missing values: 0

Product Description Dataset

Feature Name	**Description**	**Values**

Asin	represents the ID of the product.	10-character alpha-numeric string, typically formatted as a mix of uppercase letters and numbers. Unique count: 18546 Number of missing values: 0
Helpful_vote	represents the number of times customers marked the review as “helpful”.	Values range from 0.0 to 888.0. Number of missing values: 0
Images	represents the images that customers post after they have received the product.	Unique count: 35 Number of missing values: 0
Parent_asin	represents the Parent ID of the product, which groups related editions.	10-character alphanumeric identifier, typically formatted as a mix of uppercase letters and numbers. Unique count: 18546 Number of missing values: 0
Rating	represents the rating of the product (from 1.0 to 5.0)	Values range from 1.0 to 5.0. Number of missing values: 0
Text	represents the text body of the customer review.	Unique count: 19668 Number of missing values: 0
Timestamp	represents the date and time when the review was posted.	Number of missing values: 0
Title	represents the title of the customer review.	Top 3 Most Frequent Titles: {'Five Stars': 382, 'Four Stars': 130, 'Good read': 71} Unique count: 16649 Number of missing values: 0
User_id	represents the unique ID of the reviewer.	28-character alphanumeric unique identifier, a structured combination of uppercase letters and digits. Unique count: 1354 Number of missing values: 0
Verified_purchase	represents whether the reviewer purchased the book (True / False).	A binary categorical flag (True / False).

		Value_counts : {True: 11419, False: 8581} Unique count: 2 Number of missing values: 0

Exploratory Data Analysis

After ingestion, we will perform a series of EDA to generate data visualisations that glean insights into the characteristics and patterns present in this batch

1. **Descriptive Statistics for rating, timestamp and sentiment_score**

```
Descriptive Statistics for Numerical Columns:
              rating     timestamp  sentiment_score
count  188704.000000  1.887040e+05    188704.000000
mean        4.539490  1.616674e+12         0.296447
std         0.885505  3.942644e+10         0.218113
min         1.000000  1.546303e+12        -1.000000
25%         4.000000  1.582891e+12         0.162500
50%         5.000000  1.619376e+12         0.294107
75%         5.000000  1.649716e+12         0.433333
max         5.000000  1.694593e+12         1.000000
```

Figure B1: Descriptive Statistics

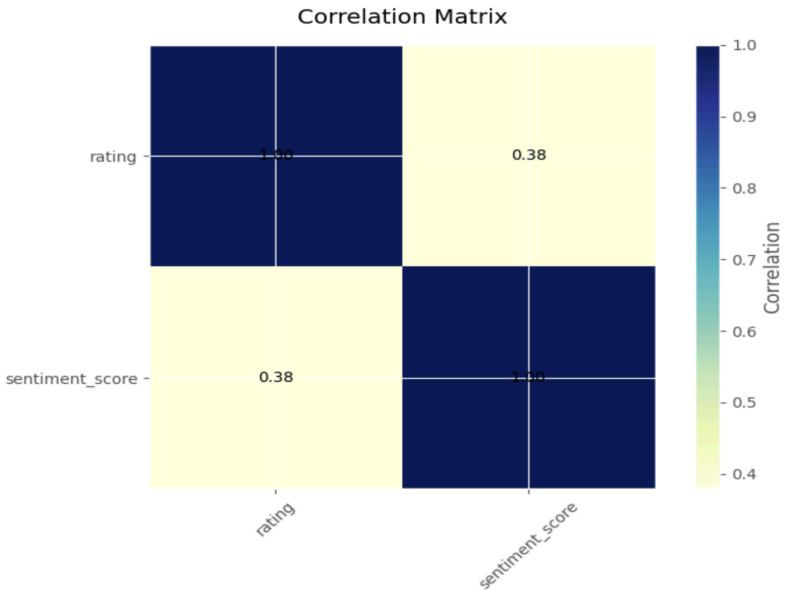


Figure B2: Correlation Matrix

2. **Distribution of Rating and Sentiment across Books and Users**

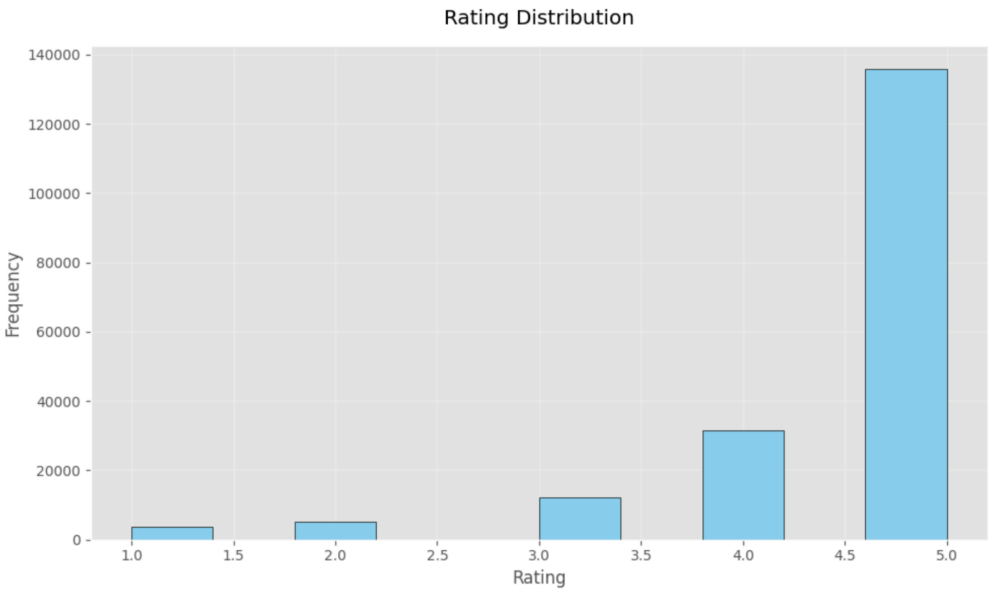


Figure B3: Rating Distribution

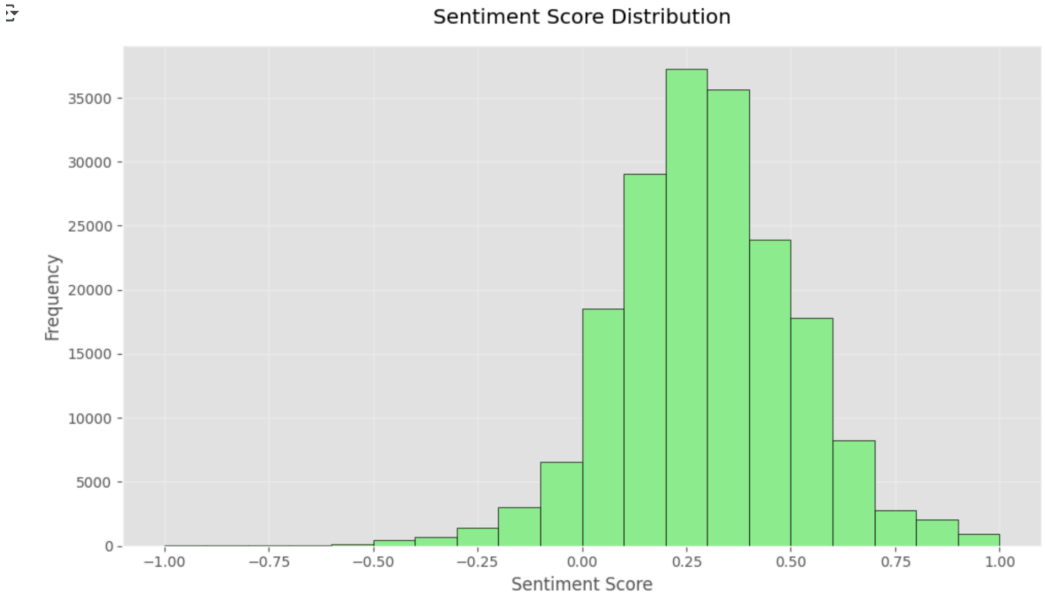


Figure B4: Sentiment Score Distribution

3. Top 10 Most Reviewed Books

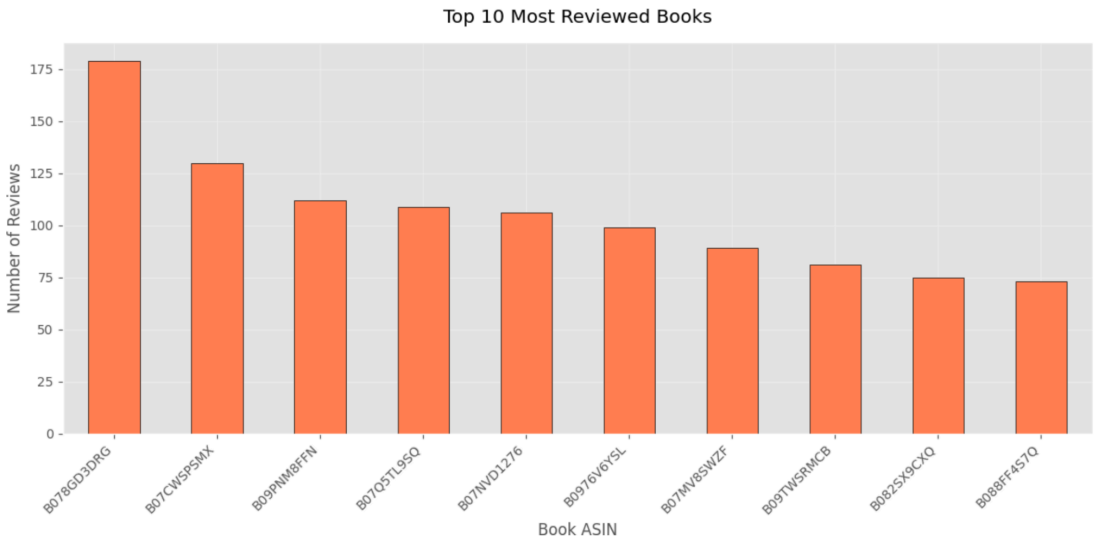


Figure B5: Most Reviewed Books

4. Average Rating Trend

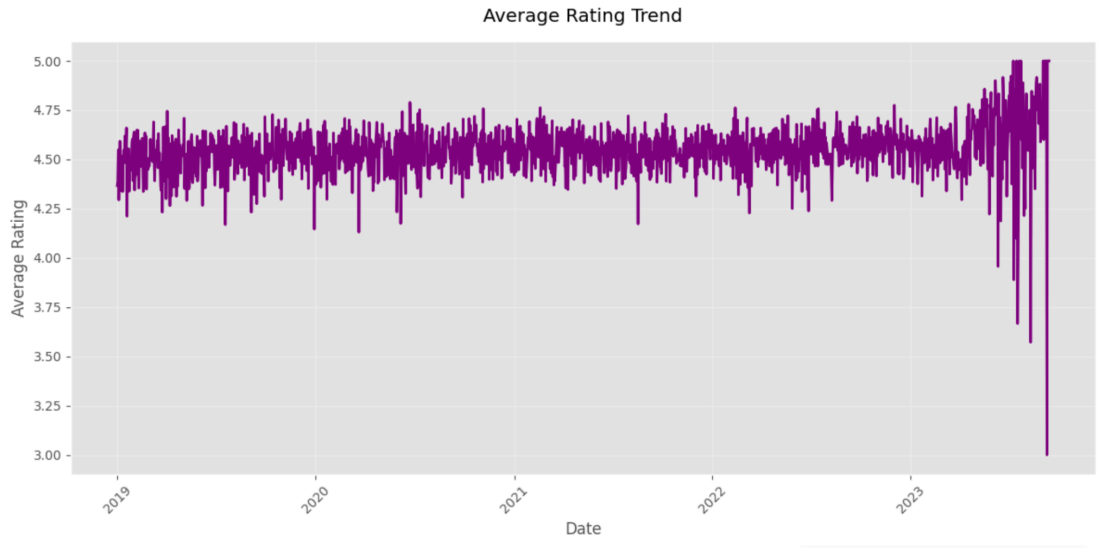


Figure B6: Average Rating Trend

```
=== Numerical Statistics ===
       average_rating  rating_number
count      1591371.00     1591371.00
mean             4.31         377.44
std              0.55        2536.58
min              1.00           1.00
25%              4.10           9.00
50%              4.40          34.00
75%              4.60         159.00
max              5.00      610476.00
```

```
=== Price Statistics ===
count    1358967.00
mean           4.13
std            8.79
min            0.00
25%            0.00
50%            0.99
75%            5.99
max         3000.00
Name: price_numeric, dtype:
```

Figure B7: Books Metadata

1. Distribution of Average Ratings Across Books

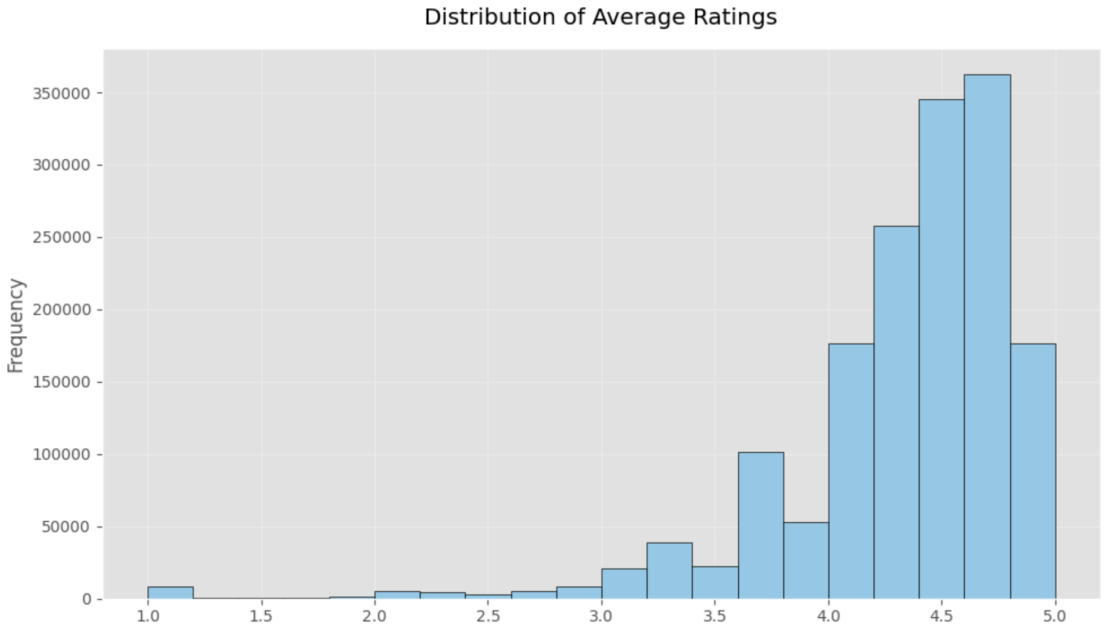


Figure B8: Average Ratings Distribution

2. Price Distribution

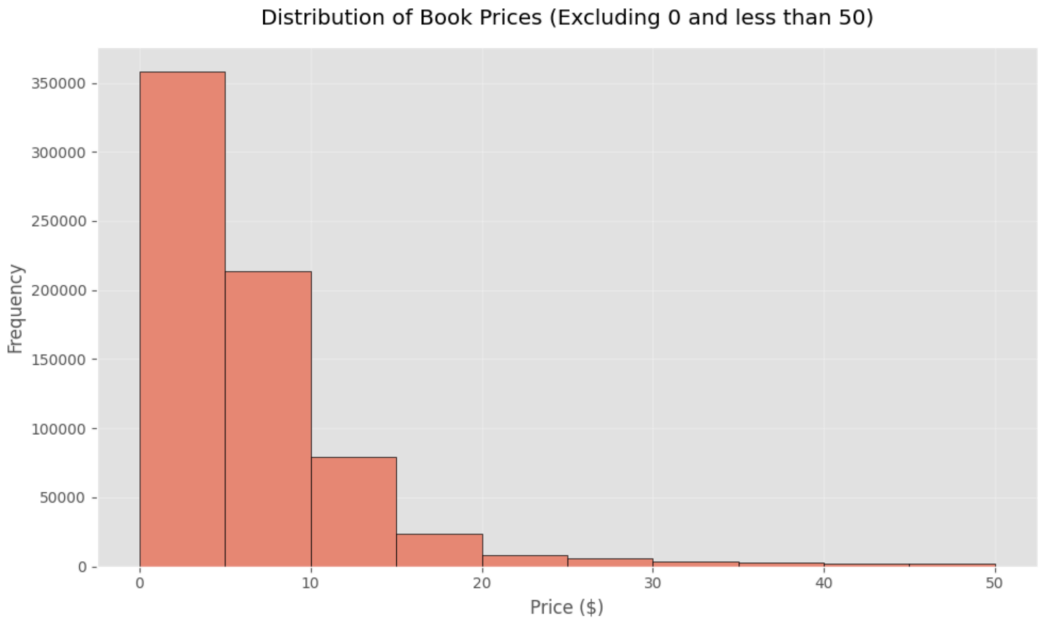


Figure B9: Price Distribution

3. Top Genres

```
=== Top 10 Genres ===
genre
Literature & Fiction          499505
Romance                       135756
Religion & Spirituality       127680
Science Fiction & Fantasy     106723
Children's eBooks             103999
Mystery, Thriller & Suspense   75856
Business & Money               67297
History                        53031
Teen & Young Adult             49693
Biographies & Memoirs          47160
```

Figure B10: Genres

4. Top 10 Books by number of ratings

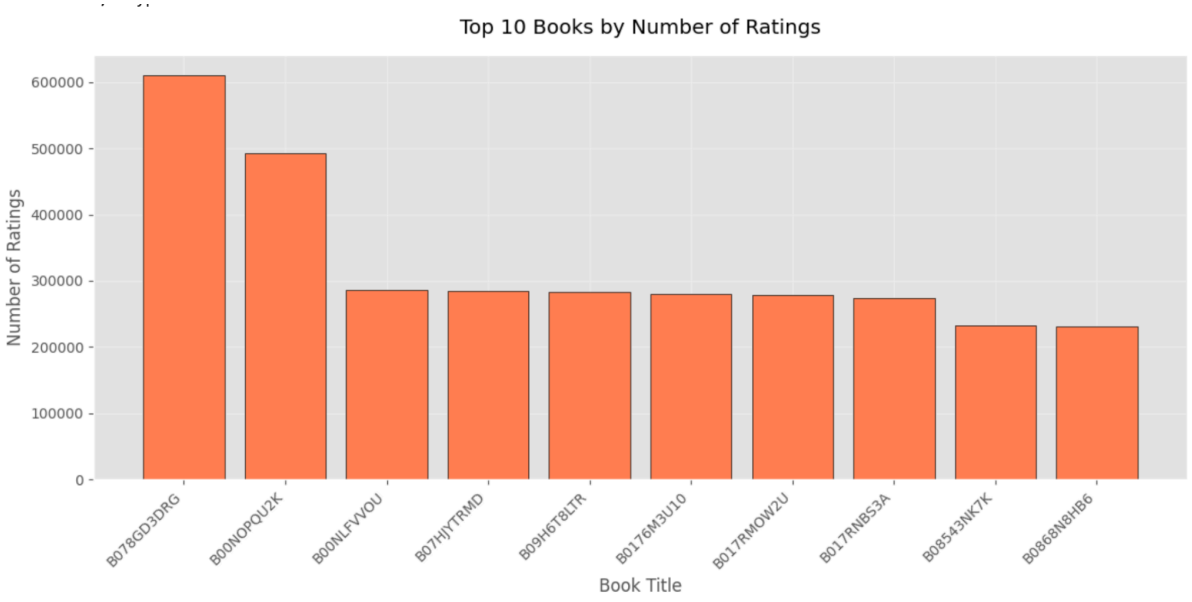


Figure B11:.10 Top 10 Books

General Assumptions about Customers:

1. **Inactivity:**

   After purchasing a Kindle book, a customer will likely purchase another Kindle Book based on prior purchasing patterns (exploitation) else something recommended to them (exploration), and there will not be a scenario where a customer stops purchasing books or becomes inactive.

   Justification:

   Historical purchasing behaviours show that customers engaged with the platform make repeated purchases if they remain active in general. We would like to consider any form of inactivity for an extended period to be an outlier scenario.

2. **Single Customer, Single Account:**

   Customer does not own multiple Amazon accounts or share his or her account, and all purchases on the customer's Amazon account is attributed to only one customer, so every interaction with the Amazon account is unique to each customer.

   Justification:

   Amazon's policies discourage account sharing and we assume that customers adhere to the policies.

3. **No External Influence**:

   Customers are not influenced by other external factors, such as promotions on other platforms, specific hardware or technical limitations. Customers are not influenced by external economic factors that may limit their ability to pay for books.

   Justification:

   We focus on how internal factors play a role in influencing purchasing behaviour. This assumption simplifies the problem and should be revisited when more data can be collected.

4. **No Duplicate Purchases**:

   Customers will not be able to purchase the same book twice. Any downloadable content, expansions or special editions do not replicate the book.

   Justification:

   There are no policies that are implemented to prevent users from purchasing the same Kindle book, however we felt that there is a huge decrease in customer utility when the customer purchases the same kindle book again. However, from the dataset, there are edge cases where the customer gets a book refunded and then re-purchases it again, and we would like to consider them as outliers.

5. **Responsiveness**

   Customers are highly responsive to Amazon book recommendations. Amazon book recommendations play a role in the customer's decision making process.

   Justification:

   We can analyse how our book recommendations impact the customer's receptiveness to existing books.

6. **Gifting**

   Gifting is infrequent and only takes up a small proportion of purchases made by customers. We are unable to estimate the number of books that are gifted or not.

   Justification:

   This assumption helps us simplify the model by focusing on customers being 'self-centred' rather than external influences from gift purchases possibly by family or friends.

7. **Text Reviews are analysed assuming they are in English**

   Due to the lack of time, we will only do our analysis on English reviews. We can ensure consistency in our dataset for more accurate analysis and comparisons.

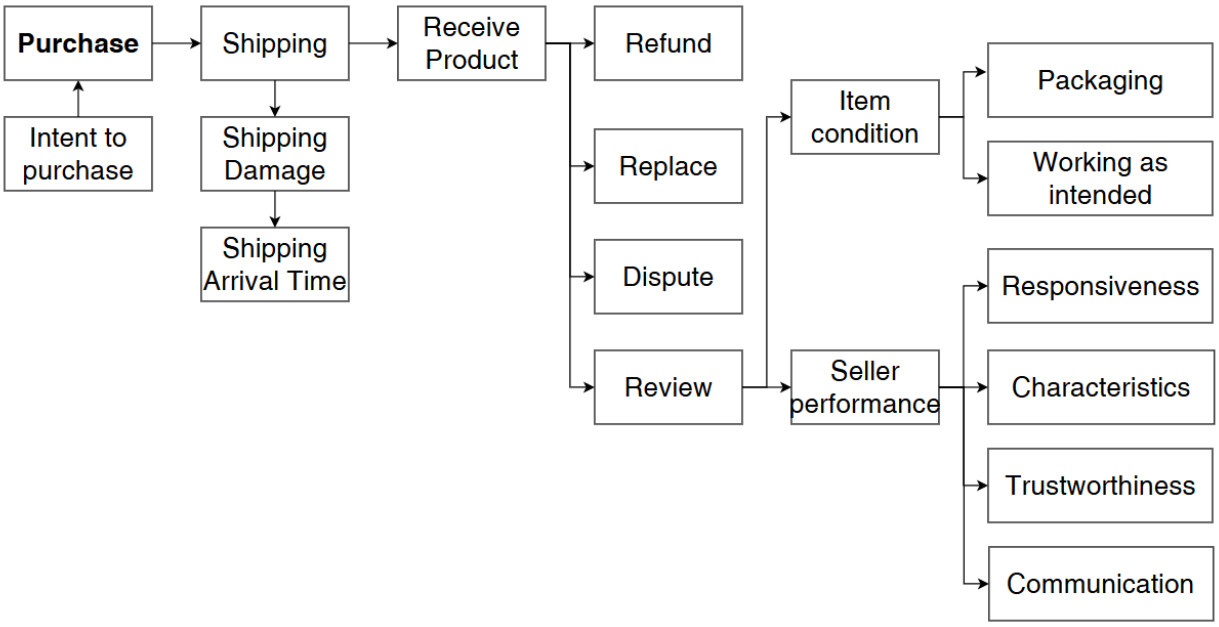


Figure B12: Our understanding of how a customer purchases a Kindle Book and the processes that are usually described in their review. We perceive customers talking about either of the steps in their review to be them trying to appeal to credibility.

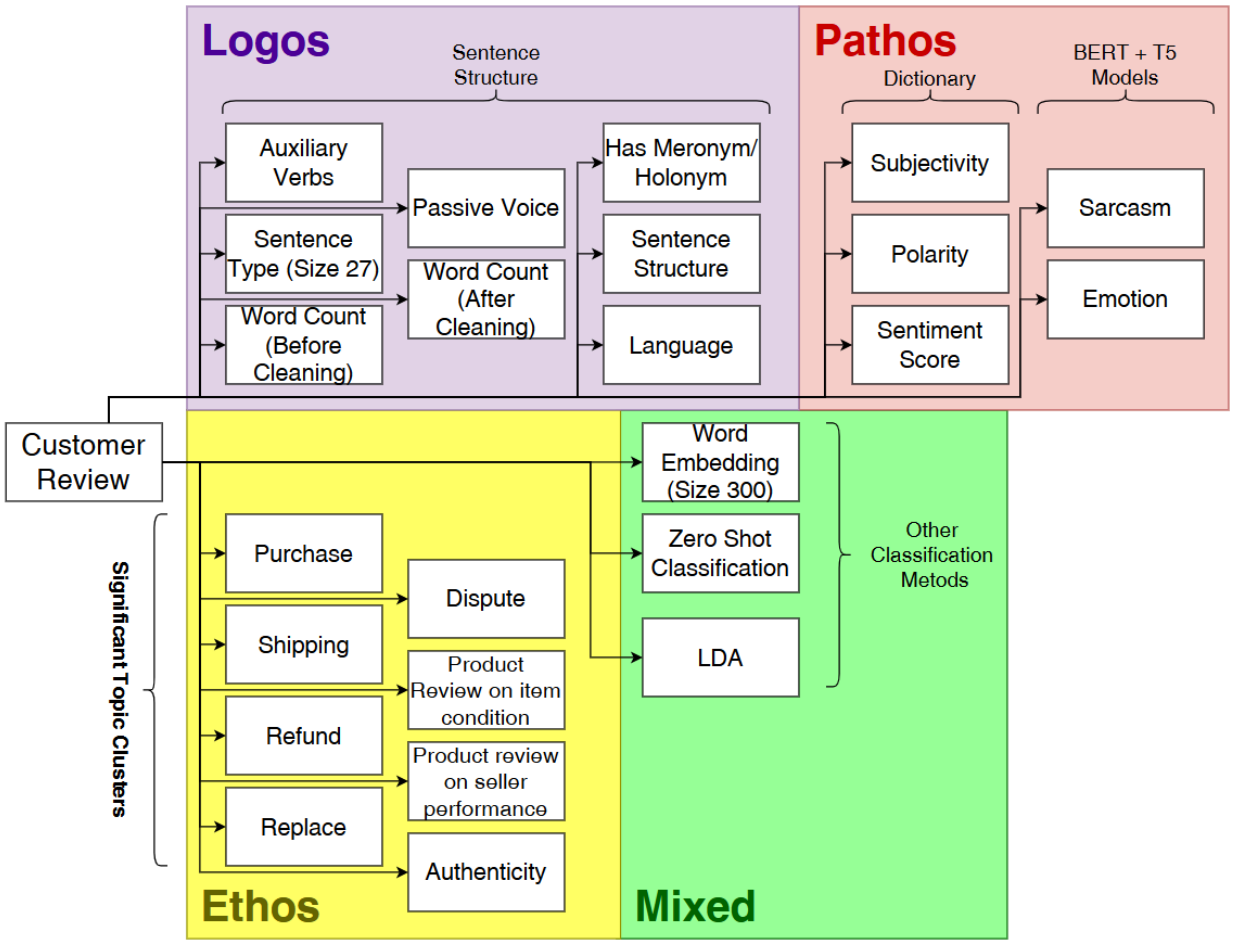


Figure B13: Our understanding of why a customer writes his or her review is based on the Aristotelian triangle, where customers tend to appeal to credibility, logic or emotions in their review. We hope that through heavy feature engineering of customer reviews, we can better recommend Kindle books to customers.

Appendix C: Data Pipelining

Data Ingestion

To automate our data pipeline with Airflow, we must establish a consistent period for ingesting data, and we have decided on a period of one week. This periodicity ensures that new cleaned and processed data is incorporated on a monthly basis into our analysis and modelling, maintaining the relevance and accuracy of our predictive models as it gets exposed to the most current trends and patterns in our hotel bookings data. The availability of timestamps enables incremental ingestion. By tracking the maximum timestamp from the previous run (e.g., stored in a metadata table or Airflow variable), the extraction task could be modified to fetch only reviews and metadata with timestamps greater than the last processed value. This reduces redundant processing and optimises resource usage.

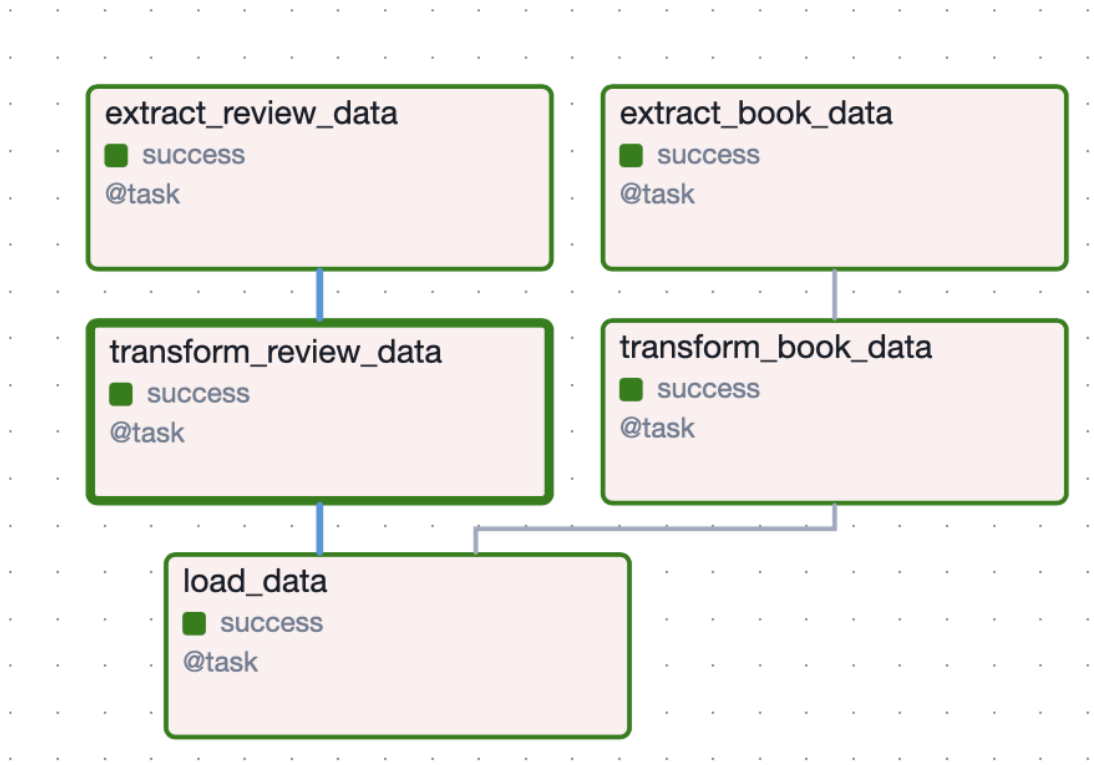


Figure C1:Airflow Graph

We transfer the transformed Kindle Store reviews and metadata into Google Cloud BigQuery for persistent storage and downstream use. This step ensures the processed data is structured and accessible in a cloud-based data warehouse.

Setup and Process: The pipeline authenticates with BigQuery using a service account key located at big-query-key.json. It establishes a secure connection to the 'IS3107-454516' project.

Reviews Schema: rating (FLOAT), sentiment_score (FLOAT), parent_asin (STRING), user_id (STRING), timestamp (TIMESTAMP).

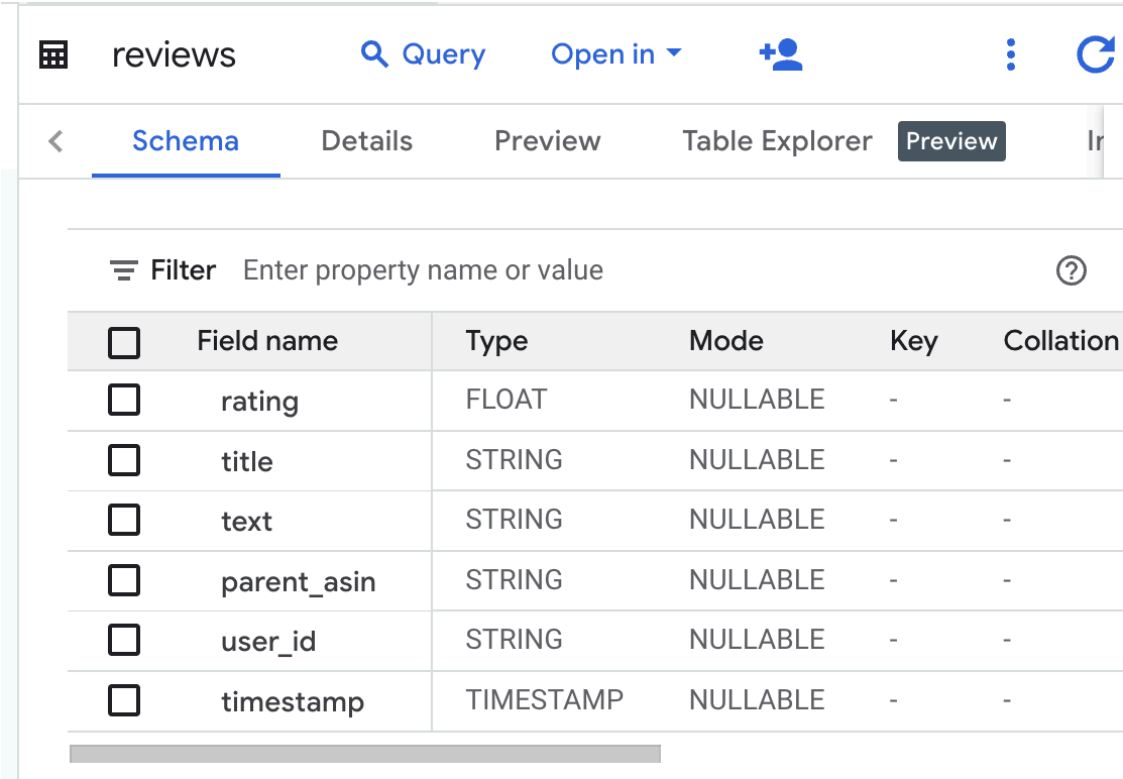


Figure C2: Review Schema

Metadata Schema: title (STRING), average_rating (FLOAT), rating_number (INTEGER), price (FLOAT), store (STRING), genre (STRING), parent_asin (STRING), publisher (STRING), publication_date (STRING), language (STRING).

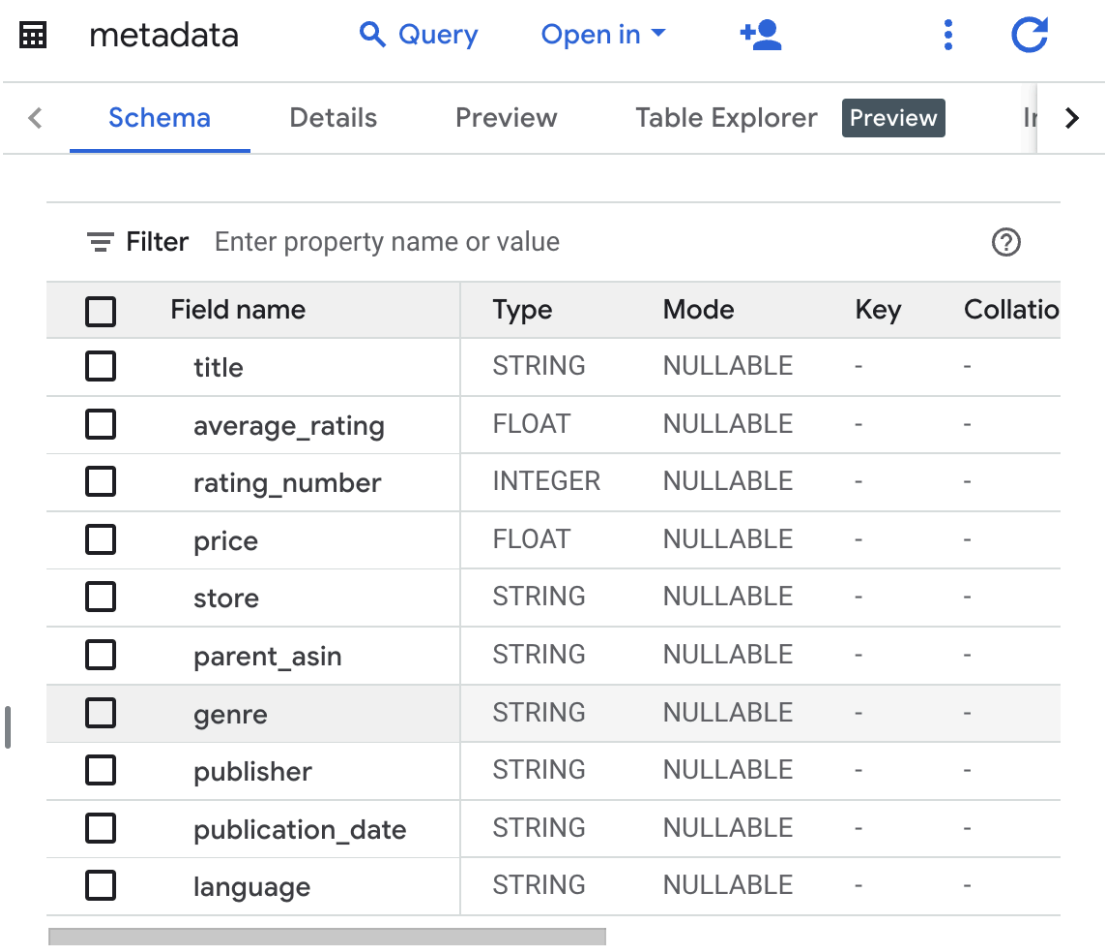


Figure C3: Metadata Schema

The transformed datasets are then uploaded to two tables in the kindle_store dataset.

This streamlined loading process ensures the data is efficiently transferred, structured, and stored, fulfilling the pipeline's objective of delivering clean, transformed data to a cloud destination.

Appendix D: Machine Learning + MLOps

Model Experimentation

The MLOps DAG orchestrates the flow of data and models across the pipeline. Experiments were managed using MLflow, with each model tracked and logged as a distinct experiment.

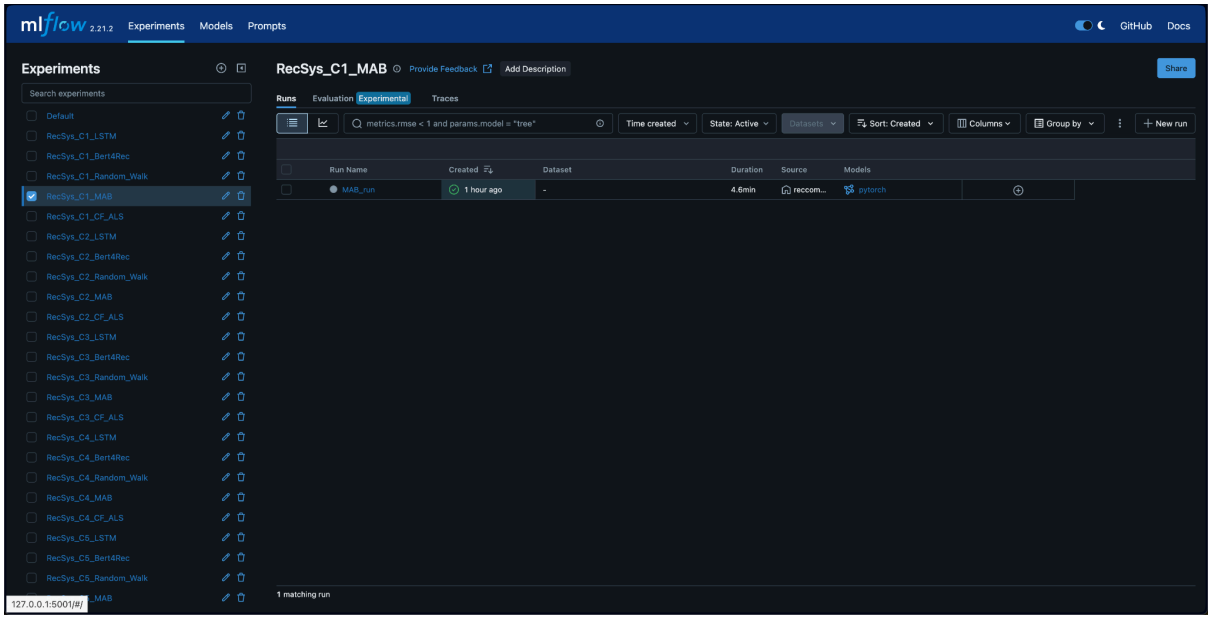


Figure D1: List of Experiments on MLflow

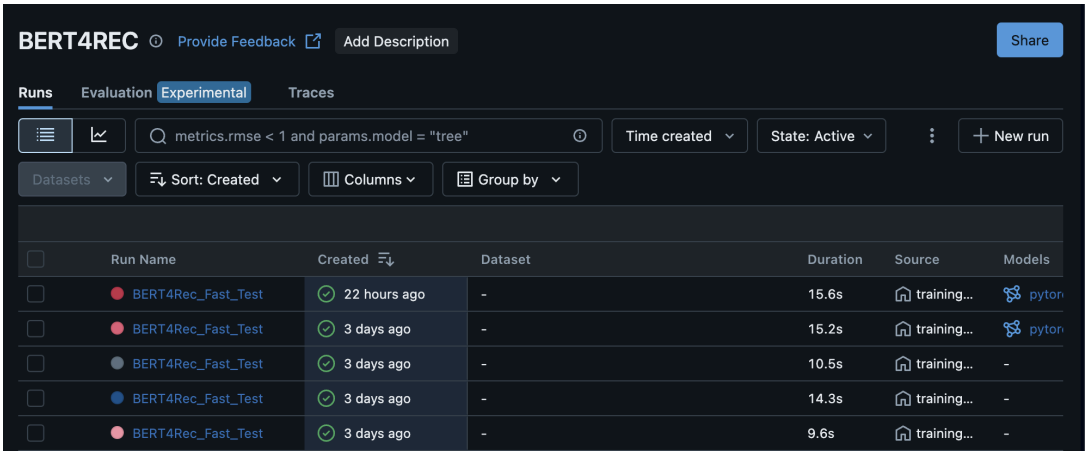


Figure D2: Runs for Bert4Rec

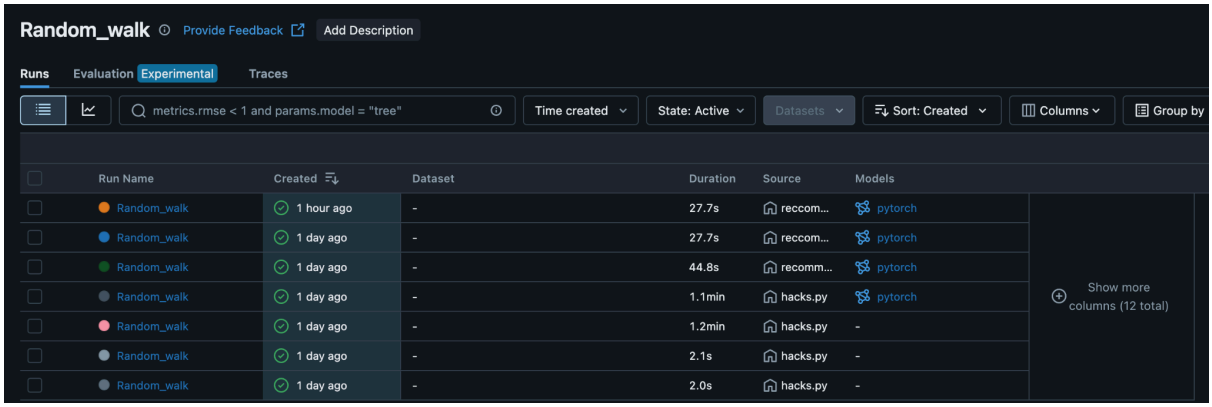


Figure D3: Runs for Random Walk

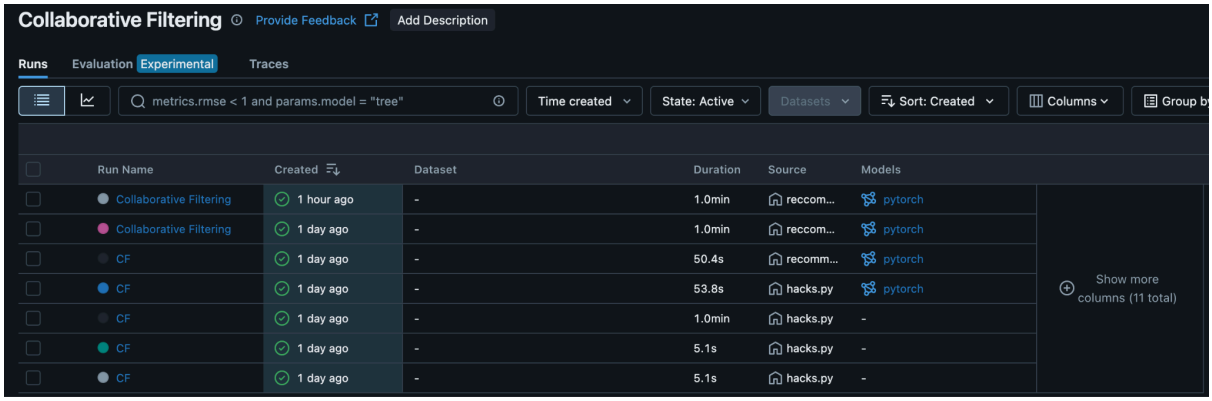


Figure D4: Runs for Collaborative Filtering



Figure D5: Runs for MAB

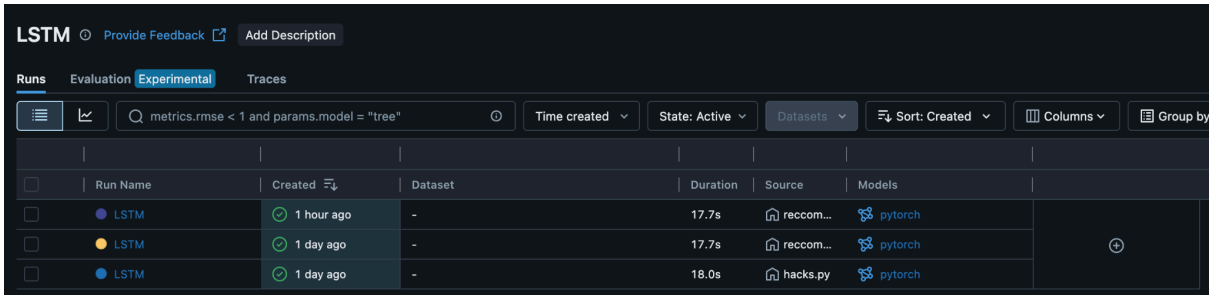


Figure D6: Runs for LSTM

For each model, several hyperparameters were logged in the MLflow run. For example, the hidden_size, num_epochs, num_layers, num_books etc. This is to allow for a quick look at the parameters the model is running with and if it should be deployed over other models with different parameters.

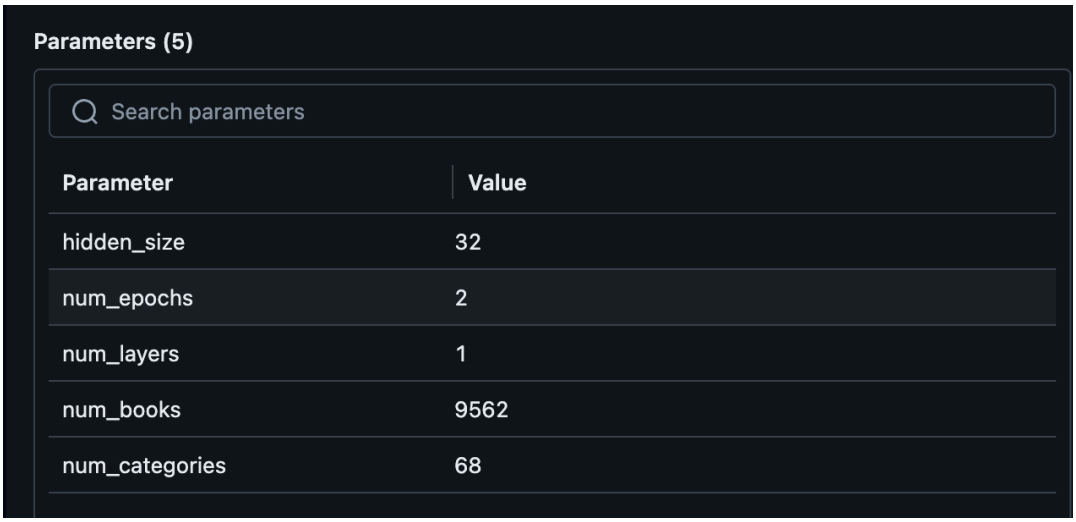


Figure D7: Hyperparameters for the Bert4Rec Recommender

Cluster	Description	Model Used
Cluster 1	Frugal shoppers	LSTM
Cluster 2	Selective spenders	BERT4Rec
Cluster 3	New / Dormant users	Random Walk
Cluster 4	Premium or loyal shopper	MAB
Cluster 5	New user	Collaborative Filtering

Figure D8: Table of best performing model for each cluster

Cluster 1 - Long Short Term Memory (LSTM)

A Long Short-Term Memory (LSTM) model leverages recurrent neural network (RNN), characterised by its suitability for sequential data. The model is designed to predict user ratings for books based on their past interactions, particularly focusing on price sensitivity, a characteristic of frugal shoppers.

To train the LSTM, we used the following features:

Numerical	Categorical (Encoded)	Text (TF-IDF)

rating: Past ratings given by the user. **average_rating:** Average rating of the book. **rating_number:** Total number of ratings for the book. **price:** Price of the book. **days_since_publication:** Number of days since the book was published. **print_length:** Length of book	**user_id:** Unique identifier for the user. **parent_asin:** Unique identifier for the book. **categories:** Genre of the book. **author:** Author of the book.	**title:** Title of the book, transformed using TF-IDF.

Figure D9: LSTM features

The model utilises user-specific purchase and review history, as well as information about the books themselves, such as price, genre, and author, to forecast ratings. Feature ratings reveals reader's taste while user_id allows the model to learn individual user preferences. Book metadata provides insights into the book's characteristics, influencing user ratings and potential outreach. Temporal information timestamp is involved in sequence creation but not training to capture temporal patterns and dependencies in user behaviours and understand how their preferences evolve over time. Altogether, they enable the model to capture a holistic view of the factors influencing user ratings to target them with relevant products and deals.

The model consists of an embedding layer, an LSTM layer, and a fully connected layer. The embedding layer transforms categorical data, like book IDs, into numerical representations that the LSTM layer can process. The LSTM layer captures temporal dependencies in user behaviour by maintaining a hidden state that evolves with each input in the sequence. The fully connected layer acts as an output layer, mapping the LSTM's final hidden state to a predicted rating. The model is designed to process sequences of user interactions, where each sequence represents a user's purchase history over time. The sequence includes features like the price of books, genre, and author, enabling the model to learn user preferences based on both individual items and their order.

The LSTM model is effective in predicting patterns of frugal shoppers because it explicitly considers price as a key feature. By incorporating price information into the input sequence, the model learns how price influences a frugal shopper's purchasing decisions. Frugal shoppers might demonstrate certain behaviour patterns, like opting for cheaper books or waiting for discounts. The LSTM model's ability to capture these sequential patterns through the input sequence and the recurrent nature of the LSTM layer makes it well-suited for understanding and predicting the behaviour of this shopper segment.

This approach provides substantial business value by enabling personalised recommendations and targeted promotions. Understanding how frugal shoppers react to price changes and discounts helps Kindle optimise pricing strategies. For example, offering discounts or coupons at the right time can incentivise purchases without compromising profit margins. This type of analysis might identify certain genres or authors preferred by frugal shoppers, providing insights for product development decisions. The LSTM model's capability to predict preferences and behaviour drives value across multiple facets of a business, leading to increased customer satisfaction, improved sales, and operational efficiency.

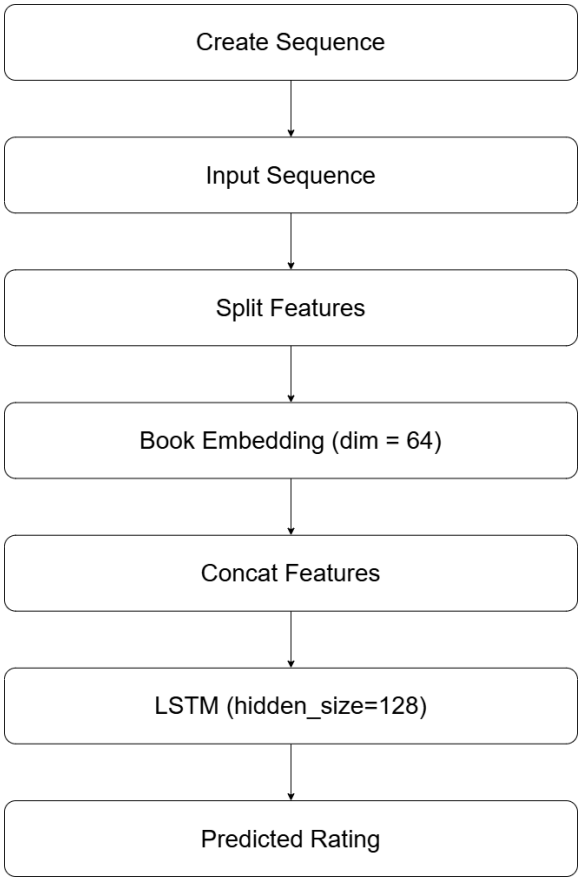


Figure D10: *LSTM Model Architecture*

Cluster 2 - BERT4REC

We chose BERT4Rec for this problem because it overcomes the inherent limitations of traditional unidirectional sequential recommendation models, making it exceptionally well-suited for predicting the next Kindle book a user might engage with based on their reading history.

Unlike left-to-right models such as RNN-based GRU4Rec or the Transformer-based SASRec, which encode a user's sequence, like [SciFi_Novel_A, Thriller_X, SciFi_Novel_B], by restricting each book's hidden representation to only prior interactions, BERT4Rec leverages deep bidirectional self-attention to fuse context from both earlier and later books in the sequence. By allowing each book to incorporate information from both directions, BERT4Rec captures richer reading patterns, such as a user alternating between genres or following a series, delivering more accurate and personalised Kindle recommendations.

To train this bidirectional model efficiently, BERT4Rec adopts the Cloze task, a key innovation from the paper, where random books in the sequence (e.g., [SciFi_Novel_A, [MASK], SciFi_Novel_B]) are masked and predicted based on their surrounding context. This approach, inspired by BERT in NLP, avoids information leakage that would occur in bidirectional next-item prediction, allowing the model to learn robust representations by jointly conditioning on left and right contexts.

Selective Spenders are characterized by making decisions based on quality. They make relatively few purchases but their purchases are diverse and are deliberate in what they buy. BERT4Rec outperforms competing models for this cluster, suggesting it is better at understanding and anticipating the needs of selective, quality-driven users.

This could be due to its bidirectional self-attention mechanism, BERT4Rec analyzes the entire context of a user's interaction history instead of processing sequences in a single direction. This approach is especially important for users who exhibit diverse interests yet do not follow a predictable sequence. By incorporating information from both before and after a given item in the sequence, BERT4Rec captures more nuanced patterns and makes better use of the available data. BERT4Rec uses a Cloze-style objective, which masks random items in the user sequence and trains the model to predict them based on surrounding context. This method encourages the model to learn stronger item representations, especially when user interaction data is limited. For selective spenders this training approach extracts more value from each interaction. It allows the model to pick up on subtle signals and quality-driven patterns that might be lost in models focused on volume or frequency.

A simple train-test split is used for validation with the dataset sorted by chronological order. This choice is ideal for our recommendation system, as it captures the temporal aspects of user behavior. Selective Spenders exhibit thoughtful, evolving preferences over time, with sparse but diverse purchases. By training on the first 80% of interactions (earlier data) and testing on the last 20% (recent data), the split mirrors real-world deployment, where the model predicts future purchases based on past behavior. This preserves the temporal sequence, unlike k-fold cross-validation, which shuffles data and disrupts time-based patterns. The approach is also computationally efficient for large, sparse datasets, ensuring robust evaluation without excessive overhead.

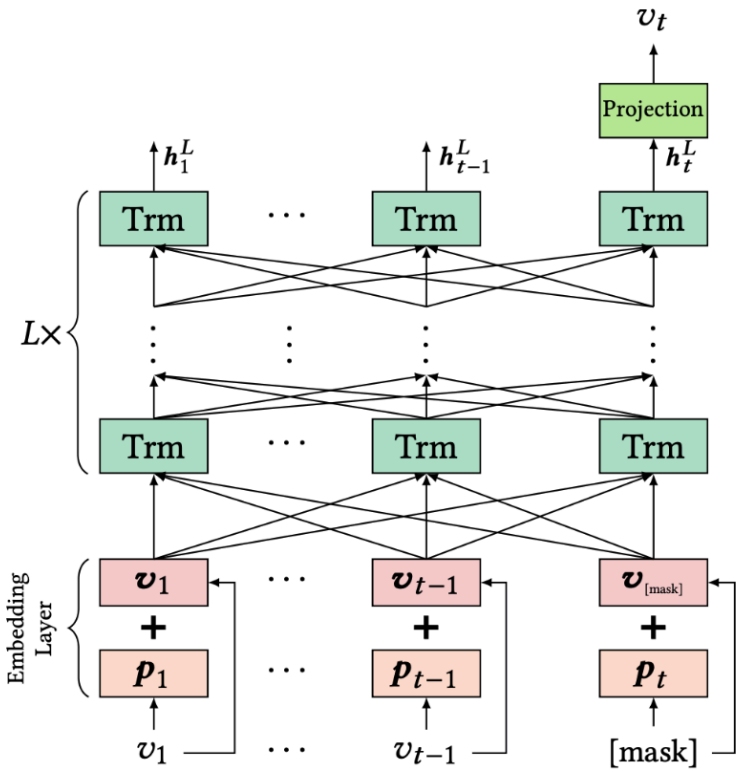


Figure D11: Bert4Rec Model Architecture

Cluster 3 - Random Walk

Random walk is a natural fit for recommendation tasks especially for Kindle book recommendations since the recommendations are sequential in nature based on time. This suggests that we can have each book id as the node and any combination of latent or exogenous features as the edge connecting the nodes together. Through modelling a directed temporal graph, we hope that the random walks on the graph can capture temporal and behavioural proximity. Unlike matrix factorisation, neural network or content-only methods, random walks allow transition to less obvious but contextually meaningful books which is particularly crucial for Kindle where hard-to-find items within large assortments may be highly relevant yet sparsely interacted with. For our case, the edge payload captures content similarity through the genre type, price, consumer ratings and time delta to give a representation of the interaction context.

Additionally, to recommend 24 items, we considered four ways to recommend 24 items. Since random walk explores things in a graphical basis, we considered giving the user the option to choose if he wants to explore 24 separate items from the last node (so do a Breadth First Search from the last node), or if the user wants to explore a chain of 24 items that are related to each other by sequence (so do a Depth First Search from the last node), or a combination of both techniques so (2,12), (4,6), (6,4), (12,2) where the left hand side is breadth based search and the right hand side is depth based search. We decided to leave this option available for the user, where the search method is set default to breadth based.

We believe Random Walk did especially well for new or dormant users with at least one purchase as users belonging to this category tend to come with few interactions. With a sparse amount of personal history, a random walk starting from the few interactions the user has can mean meaningful paths through previous paths that past users have walked. Furthermore, Random Walk is more probabilistic in nature and this naturally favours nodes that are more well connected and recent through travelling to nodes with higher degrees and contextual relevance. This allows dormant users to still get fresh suggestions that are up to date while preserving the relevancy of what the user previously bought. From a business perspective, our intention for customers that lie within the new and dormant customer range is to push customers toward the

other three clusters. Customers who are new and dormant are still very volatile in their behaviour as we have too little information to work off from their previous purchases.

As a further evaluation step, we acknowledge that random walkers are exploitative in nature. While there exists some randomness, given multiple iterations the results will be the same. Taking inspiration from Contextual Multi Armed Bandits, we introduced exploration into the walk so that the user can still get new “rabbit holes” to visit and perhaps through these explorations they are better able to find what they want and further be transferred to one of the other three clusters. First, each node's outgoing edges are treated as arms with context features (mean edge payload), and weights guide selection. Next, we include context-aware traversal through contextual arms (which are edges) be scored by inverse time delta (weight) and contextual similarity (average user rating, price), thus enabling preference for recency and quality. Last but not least, we added a global constant noise which adds stochasticity to ensure that paths explored can be diverse.

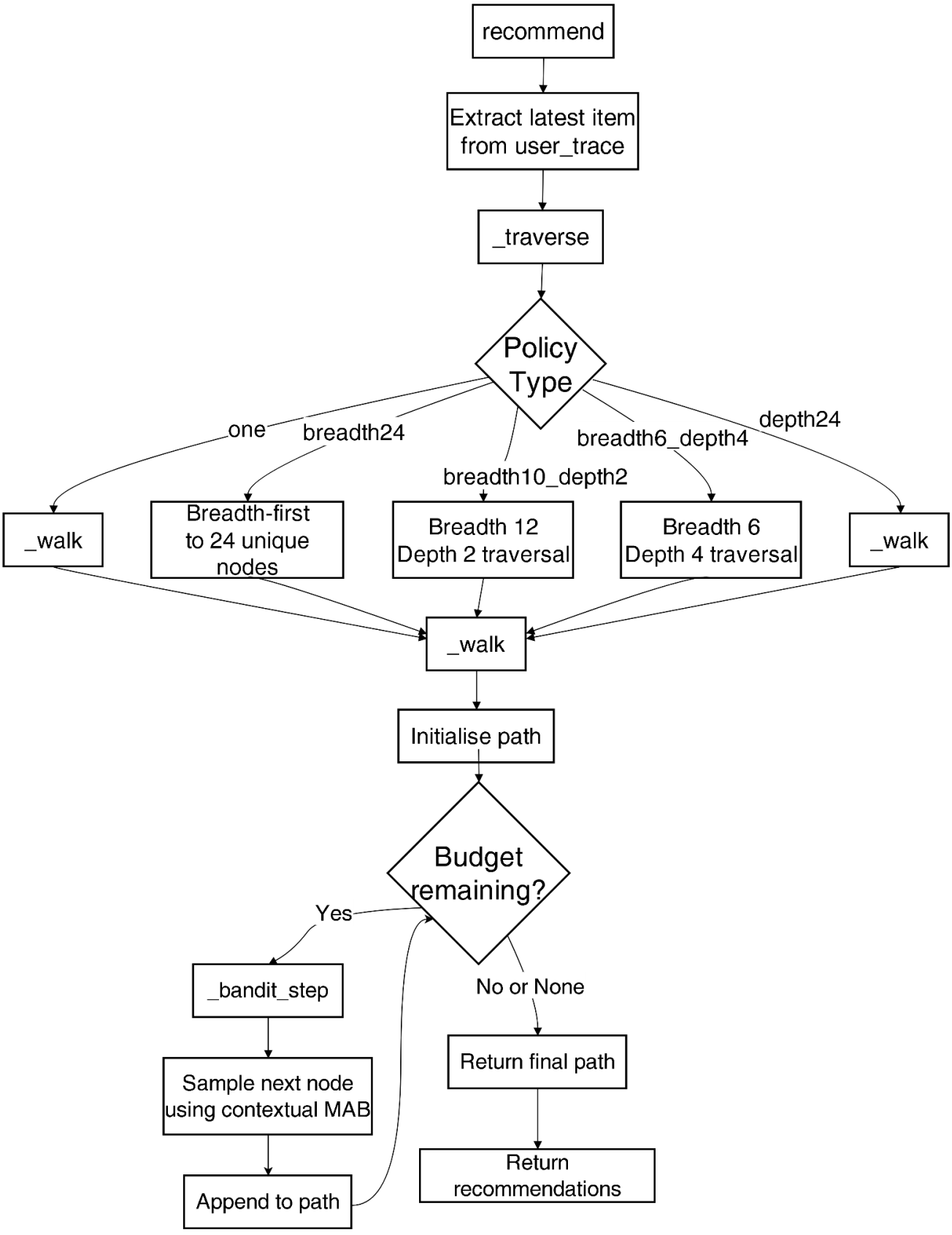


Figure D12: Process and flow of Random Walk recommendation system with Contextual Multi Armed Bandit used as the exploratory step. (Errata Breadth 6 instead of 5 and Depth 4 was used)

Cluster 4 - Multi-Armed-Bandit (MAB)

We implemented a greedy epsilon Multi-Armed Bandit (MAB) architecture to enhance the exploration-exploitation tradeoff. Specifically, we adopted a contextual greedy epsilon strategy with an epsilon value set at 0.05, which means that 95% of the time, the algorithm selects the arm (i.e., edge or recommendation path) with the highest expected reward, and 5% of the time, it explores a random edge. This design choice balances the need for consistent, high-reward recommendations with the flexibility to discover new, potentially better options that may not yet have enough interaction history to dominate the reward space.

In our implementation, each outgoing edge from a book node is treated as an arm in the bandit framework. The reward function is based on the price of the recommended item, under the assumption that premium or loyal shoppers tend to be less price-sensitive and are more likely to interact with or purchase higher-priced items. By prioritising high-price items during exploitation, the model aligns with the purchase tendencies of loyal customers, offering recommendations that both match user preferences and drive higher value transactions for the business. The contextual nature of the bandit also incorporates additional information, such as time delta and average user ratings, ensuring that price is not the only determining factor and that there is still a preference for more recent and highly rated books.

This architecture is particularly suitable for the cluster of premium or loyal shoppers, as these users typically have a richer interaction history and more predictable behaviour patterns. With sufficient data on their past purchases, the model can make confident exploitation decisions while still maintaining a small probability (5%) of exploration. This exploration is valuable in surfacing niche, long-tail content that matches the user's historical preferences but may not have appeared frequently enough to be prioritised by traditional recommendation systems. The low epsilon value strikes a balance, it does not disrupt the overall experience for loyal users by showing too many irrelevant recommendations, but still provides a mechanism to refresh their feed with relevant discoveries, increasing engagement and user satisfaction.

Furthermore, from a business perspective, this strategy supports cross-selling and discovery within high-value segments, helping loyal users to explore beyond their usual scope while staying within the realm of high-price, quality content. It also offers a safeguard for data sparsity in specific sub-genres, as occasional exploration can illuminate underrepresented but potentially valuable paths. Overall, the greedy epsilon MAB architecture offers a principled and flexible approach to personalised, high-value recommendations for Kindle users, particularly within the premium and loyal segment.

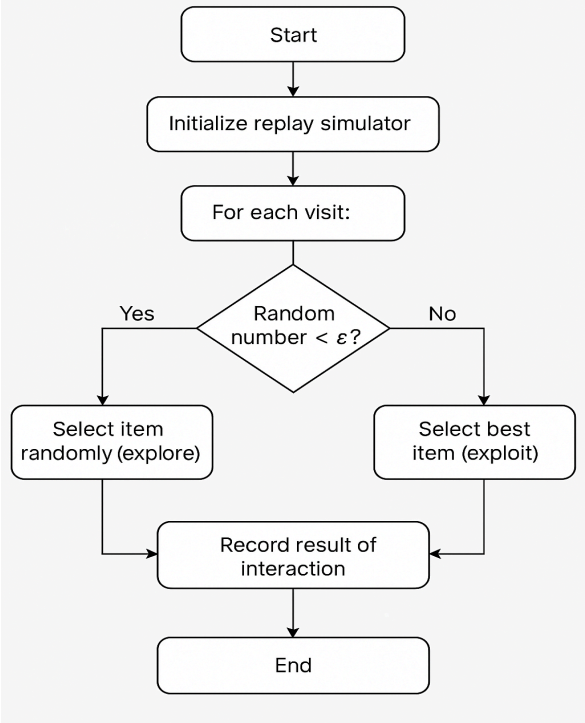


Figure D13: MAB Model Architecture

Additionally, we experimented with Upper Confidence Bound (UCB) as a potential strategy due to its ability to systematically balance exploration and exploitation by favoring arms with high uncertainty. However, we ultimately did not adopt UCB because it tends to explore more aggressively in early stages, which can conflict with our business goal of prioritizing high-value recommendations for premium and loyal shoppers.

Since our reward function is based on price, and loyal users are assumed to be less price-sensitive, our goal was to exploit high-reward (i.e., high-price) options as early and consistently as possible. UCB, by design, would allocate more recommendations to lower-priced items if they have not been explored much yet, leading to less efficient monetisation and potentially irrelevant suggestions for this user group.

In contrast, the greedy epsilon strategy with a low epsilon (0.05) allowed us to mostly exploit known high-price items while still reserving a small chance to explore underrepresented edges. This approach aligns better with the predictable behaviours of loyal users and supports business objectives like driving higher-value transactions and cross-selling within high-end content, without overwhelming users with lower-value or irrelevant recommendations.

Cluster 5 - Collaborative Filtering

The cold-start problem in book recommendation systems arises when new users, such as our Totally New Users segment with zero purchase history, spending, or engagement, join the Kindle platform, making it challenging for machine learning models to infer preferences or generate relevant suggestions. To tackle this, our Collaborative Filtering Model takes into account integrates Alternating Least Squares (ALS) for collaborative filtering with a Content-Based Filtering (CBF) component, leveraging both user-item interaction data and item metadata to deliver personalized recommendations for both returning and cold-start users. By creating a dense vector embedding through training on the corpora of book descriptions, Collaborative Filtering is able to capture the semantic relationships between books and each book can be represented as a fixed length vector in a 300 dimensional space.

ALS predicts user preferences by analysing past interactions such as ratings and purchases. ALS then decomposes the item item interaction matrix into a latent factor representation (through Word2Vec To address this). During training (fit method), the model optimises these embeddings to minimise the Mean Squared Error (MSE) between predicted and actual ratings, using historical item interactions.

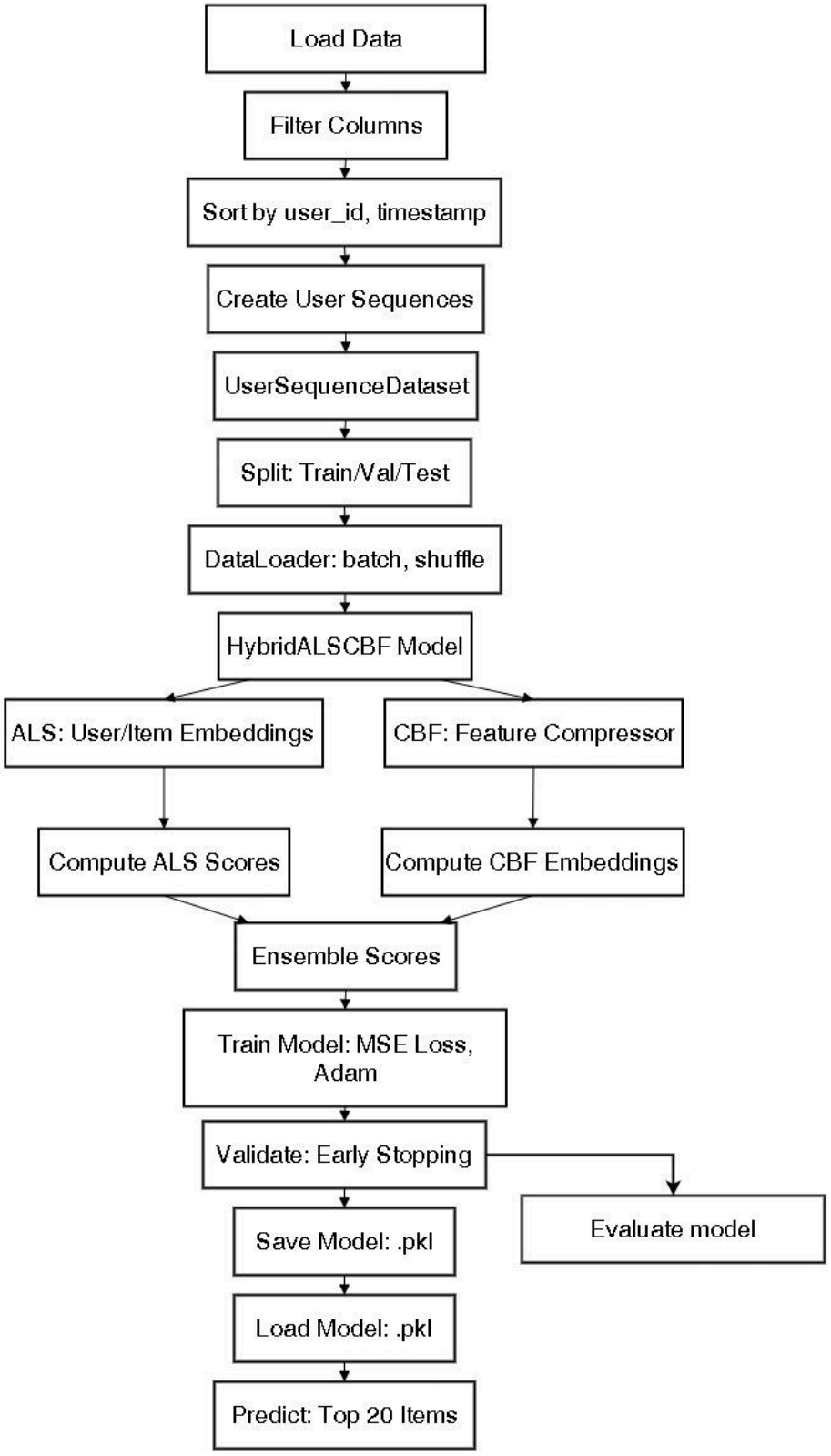


Figure D14: Collaborative Filtering Architecture

Version Control and Reproducibility

In terms of ML modeling, we manage versioning and ensure reproducibility using MLflow. This tool lets us track all model versions, revert to earlier ones when necessary, and store models as artifacts. These artifacts can later be accessed for future use or referenced via API calls on deployed models.

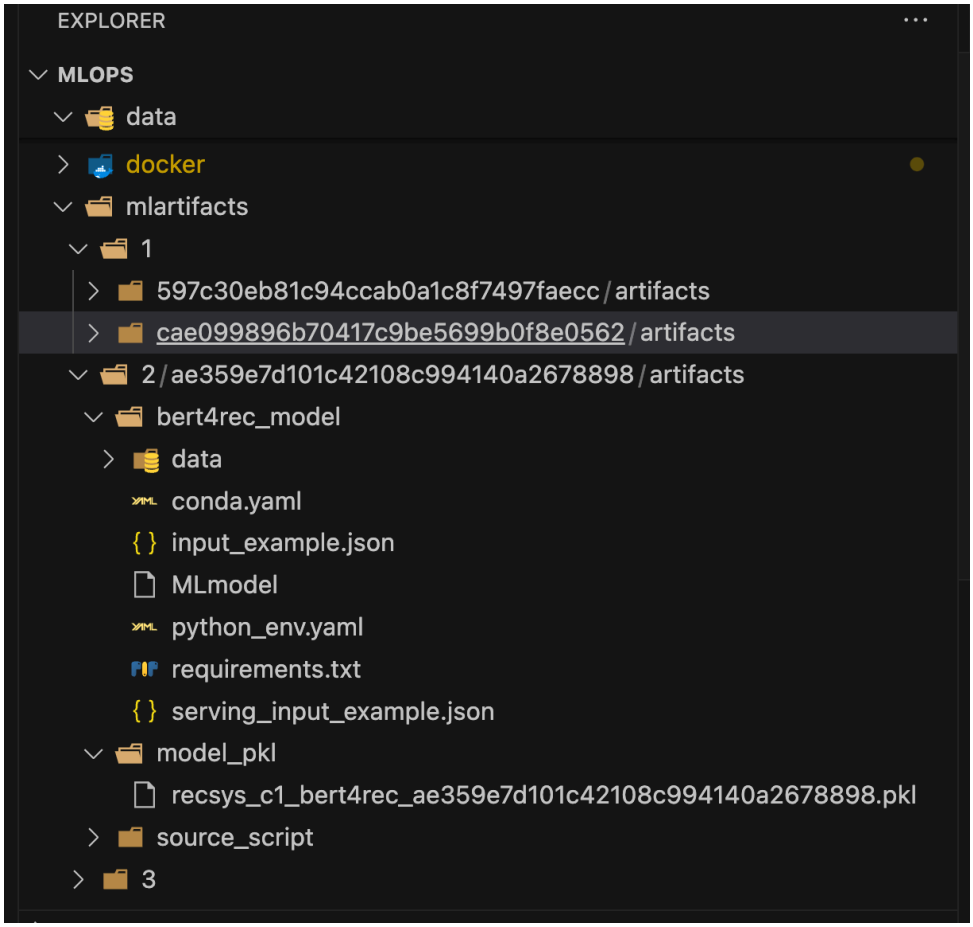


Figure D15: What MLflow Stores for Each Run

Using these files, we can seamlessly recreate the model and its capabilities. The pickle file, downloadable from the server, can be incorporated into other applications as required. The requirements.txt file ensures version consistency, simplifying the process of duplicating the setup in a virtual environment.

These workflows leverage the MLflow UI to deliver an intuitive MLOps interface. By combining automated data pipelines with containerised and serialised models, we streamline model training, deployment, and assessment. Ongoing staff feedback drives continuous enhancements, maintaining a robust and flexible MLOps framework.

Appendix E: Repository Documentation

Text Tree

The following is a text tree with brief description of what each file does in ./repo

```
.
├── books.db  > SQLite database that stores structured book and user-interaction data
├── dags
│   ├── MAB_model.pkl > MAB model
│   ├── amazon_mab_ge_rating_parallel_evaluation.ipynb > EDA of MAB
│   ├── classify.py > CLI py script for zero-shot text classification of customer reviews
│   ├── classify.txt > desc of classify.py
│   ├── etl.py > Apache Airflow TaskFlow DAG end-to-end weekly ETL pipeline for Aws Kindle Store data, ingestion, cleaning, sentiment enrichment, and BigQuery loading
│   ├── gmm.py > Performs GMM clustering on user behavioural data
│   ├── gmm_cli.py > CLI wrapper around gmm.py implementation that supports training, inference, batch processing, and visualisation
│   ├── gmm_cli.txt > desc of gmm_cli.py
│   ├── gmm_cluster_plots.pdf > output of GMM
│   ├── lda.py > Performs LDA clustering on user behavioural data
│   ├── lda_cli.py > CLI wrapper around lda_cli.py implementation
│   ├── lda_cli.txt > desc of lda_cli.py
│   ├── lda_cluster_plots.pdf > output of LDA
│   ├── lstm.ipynb > EDA of LSTM
│   ├── randomwalk.py > EDA of Random Walk
│   └── randomwalkdemonstration.py > demo of Random Walk
├── docker-compose.yaml > Spin up Apache Airflow, Celery, Redis, PostgreSQL
├── mlops
│   ├── bigquery.py >authenticate to BigQuery, list tables in a dataset, and preview rows from a selected table to verify data access and contents
│   ├── docker-compose.yml > spins up reccsys on ALS, BERT4Rec, LSTM, MAB, Random Walk each in own container + log in MLFlow
│   ├── mlruns.db > SQLite db as the Backend for MLOps
│   └── models
├── requirements.txt >py libraries used (NOTE apache airflow 2.10.5 to avoid conflicts)
└── webapp
    ├── books.db  > SQLite db as the Backend for Recc sys
    ├── consumer.log
    ├── consumer.py > asynchronous data bridge between Kafka and WebSocket client
    ├── dashboard.py > backend predicts, tracks behaviour, computes & streams events using Flask, Socket.IO, SQLite, Kafka
    ├── docker-compose.yml > starts Kafka stack with ZooKeeper and Kafka broker
    ├── mlruns.db > SQLite db as ML experiment tracking, model registry, evaluation, and observability system
    ├── no_mlflow
    │   ├── books.db  > SQLite db as the Backend for Recc sys
    │   ├── consumer.py > asynchronous data bridge between Kafka and WebSocket client
    │   ├── dashboard_read_local.py  > backend predicts, tracks behaviour, computes & streams events using Flask, Socket.IO, SQLite, Kafka
    │   ├── docker-compose.yml > starts Kafka stack with ZooKeeper and Kafka broker
    │   ├── generate_cluster_artefacts.py > convert model into 2D clustered book embeddings
    ├── templates
    │   ├── dashboard.html > Second page
    │   └── index.html > First page
    └── training.py > script trains, evaluates, and logs recommendation model for books + metric evaluation + MLflow experiment tracking
```

User Guide

Refer to howtorun.txt in repo
cd into repo using wsl or macos

```
#In 1st Terminal
wsl -d Ubuntu
source ~/venv/bin/activate
cd ~/IS3107
docker compose up -d
cd ~/IS3107/webapp/no_mlflow
docker compose up -d
nc -vz localhost 9092
cd ~/IS3107
nohup mlflow server \
  --host 0.0.0.0 \
  --port 5000 \
  --backend-store-uri "sqlite:///$(pwd)/mlops/mlruns.db" \
  --default-artifact-root "$(pwd)/mlops/mlartifacts" \
  > /tmp/mlflow.log 2>&1 &
cd ~/IS3107/webapp/no_mlflow
python consumer.py

#In 2nd Terminal
wsl -d Ubuntu
source ~/venv/bin/activate
cd ~/IS3107/webapp/no_mlflow
python dashboard_read_local.py

#In 3rd Terminal
wsl -d Ubuntu
source ~/venv/bin/activate
cd ~/IS3107/mlops
docker compose up -d
```

Appendix F: Additional Diagrams

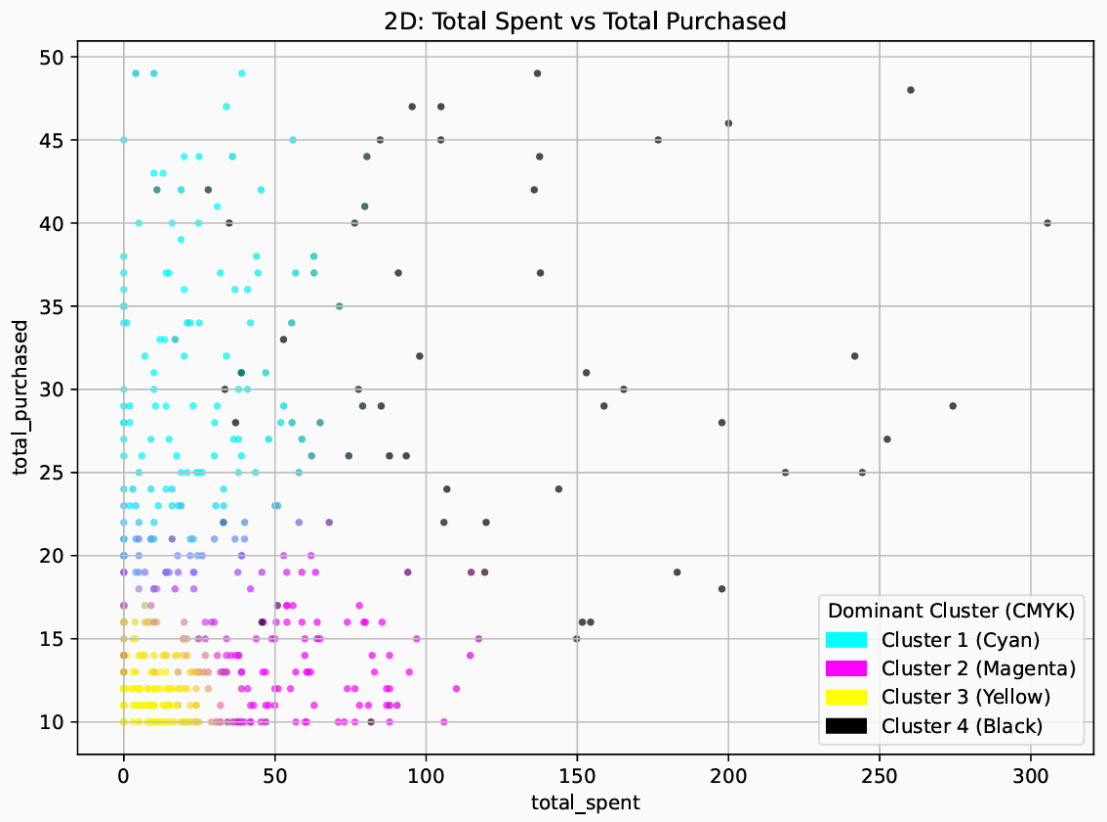


Figure F1: 500 randomly sampled users clustered through Gaussian Mixture Model with Total Spending as the x-axis, and Total Books Purchased as the y-axis, showing a distinct difference between the four clusters.

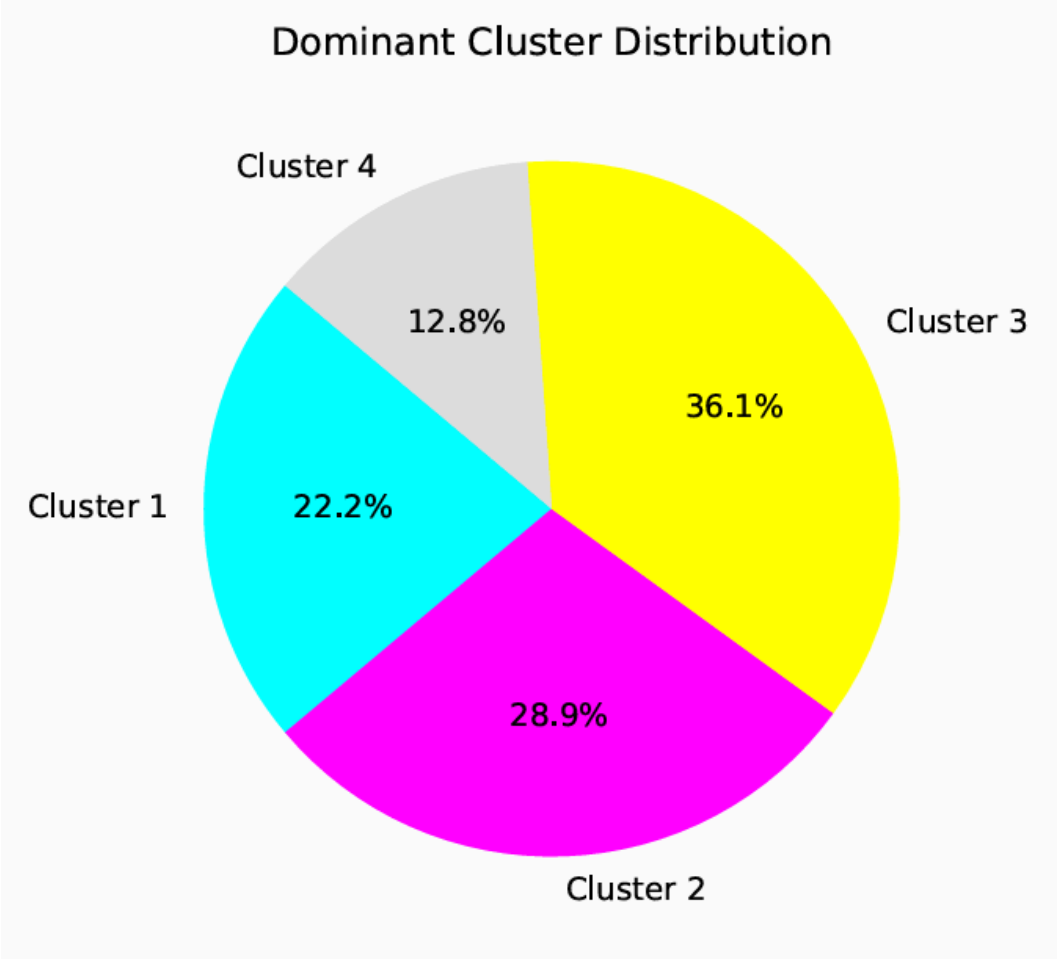


Figure F2: Distribution in percentage that a user is assigned to dominant clusters 1,2,3 or 4

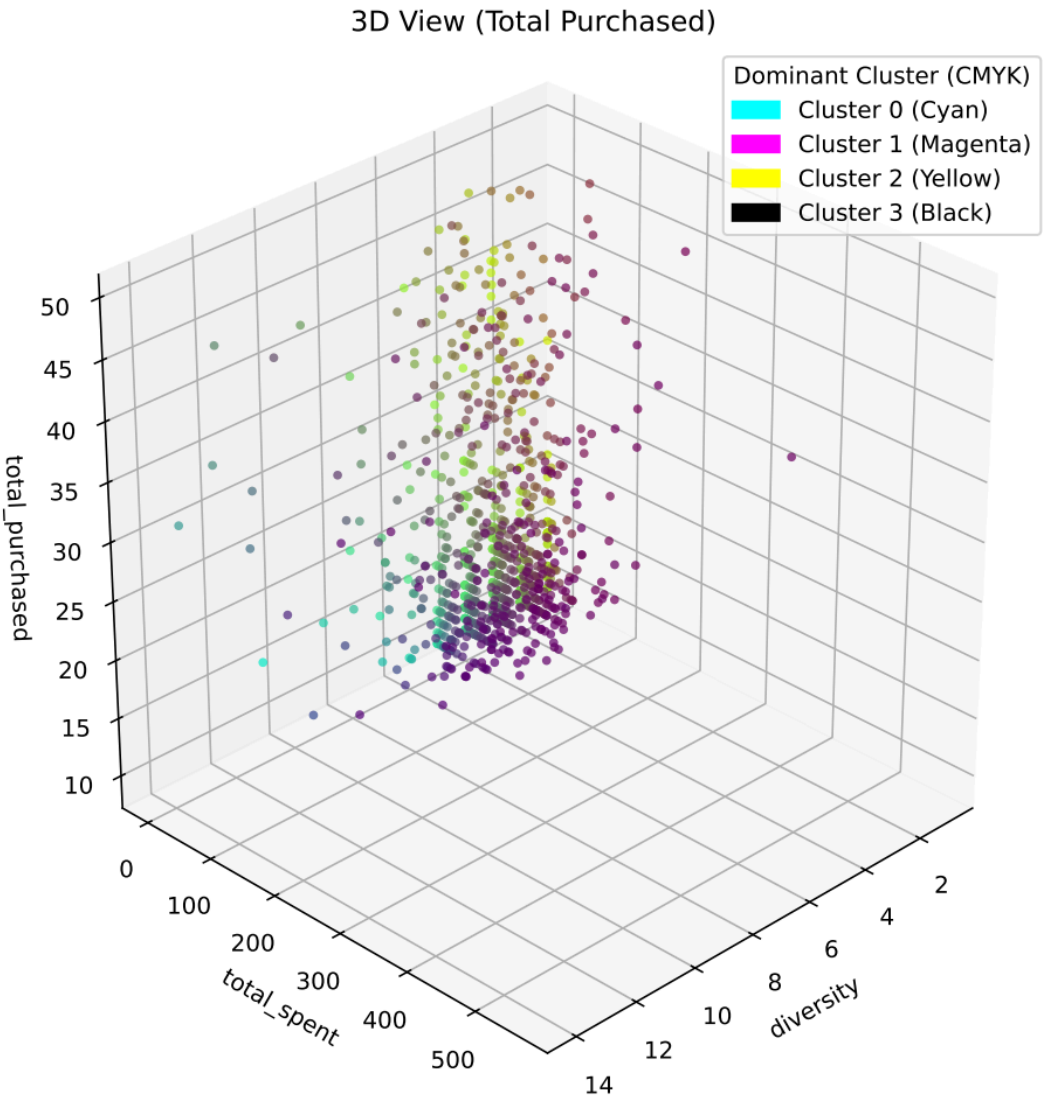


Figure F3: 500 randomly sampled users clustered through Latent Dirichlet Allocation with Diversity as the z-axis, Total Spending as the x-axis, and Total Books Purchased as the y-axis. Each point is coloured using CMYK, proportional to the probability of belonging to each of the four clusters, thus dark purple is a combination of clusters 1 and cluster 3. The legend anchors each cluster to its dominant base colour. Even with hyperparameters alpha and beta tuned to 0.01 and 0.01 to try to separate the colours more, there is still significant overlap, and it is hard to derive meaning from highly overlapped clusters with no theme. Further, cluster 1 takes up a majority of probability in each cluster.

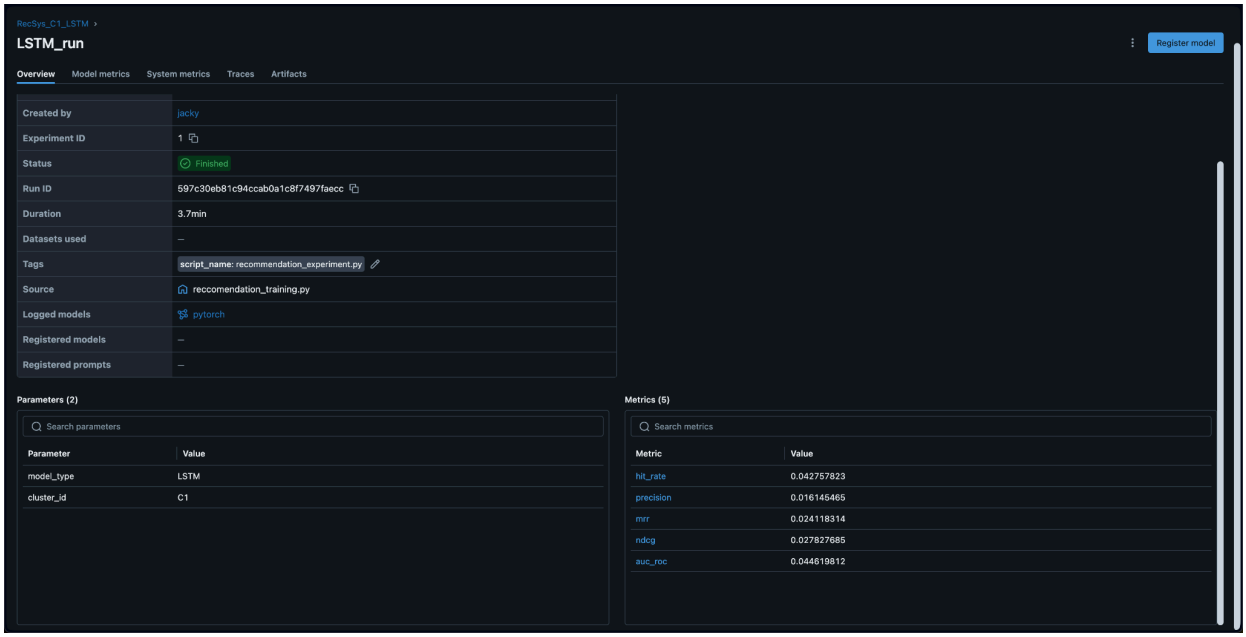


Figure F4: Logging Model Metrics

```
all_predictions = {}
for cluster_id, (model, model_type) in cluster_models.items():
    if not encoded_clicks:
        all_predictions[cluster_id] = []
        continue
    if model_type == 'pytorch':
        predictions = predict_pytorch(model, encoded_clicks, book_encoder, top_
    elif model_type == 'pickle' and cluster_id == 4:
        predictions = predict_random_walk(model, encoded_clicks, book_encoder,
    elif model_type == 'pickle' and cluster_id == 5:
        predictions = predict_cf_als(model, encoded_clicks, book_encoder, top_k
    all_predictions[cluster_id] = predictions
```

Figure F5: Clustered Predictions

```python
# Load the most recent model for each type from MLflow
def load_cluster_models():
    mlflow.set_tracking_uri("http://localhost:5001")
    client = mlflow.tracking.MlflowClient()

    cluster_mapping = {
        1: ('lstm', 'pytorch'),              # frugal shoppers
        2: ('bert4rec', 'pytorch'),          # selective spenders
        3: ('random_walk', 'pickle'),        # low purchase count
        4: ('mab', 'pickle'),               # premium/loyal shoppers
        5: ('cf', 'pickle')                 # new users
    }

    cluster_models = {}
    for cluster_id, (model_name, model_type) in cluster_mapping.items():
        try:
            runs = client.search_runs(
                experiment_ids=["0"],
                filter_string=f"tags.model_name = '{model_name}'",
                order_by=["attribute.start_time DESC"],
                max_results=1
            )
            if not runs:
                logger.error(f"No runs found for model {model_name}")
```

Figure F6: Flask Setup of clusters

```python
# Prediction functions
def predict_pytorch(model, encoded_book_ids, book_encoder, top_k=20):
    with torch.no_grad():
        max_seq_len = 10
        if len(encoded_book_ids) < max_seq_len - 1:
            input_seq = [0] * (max_seq_len - len(encoded_book_ids) - 1) + encoded_b
        else:
            input_seq = encoded_book_ids[-(max_seq_len - 1):] + [1]
        input_tensor = torch.tensor([input_seq], dtype=torch.long).to(device)
        logits = model(input_tensor)
        if isinstance(model, LSTMModel):
            probs = torch.softmax(logits, dim=0)
        else:  # BERT4Rec
            probs = torch.softmax(logits[0, -1, :], dim=0)
        top_k_probs, top_k_ids = torch.topk(probs, top_k)
        book_ids = book_encoder.inverse_transform(top_k_ids.cpu().numpy())
        return list(zip(book_ids, top_k_probs.cpu().numpy()))
```

Figure F7: Prediction Functions

Model Evaluation

Cluster 1

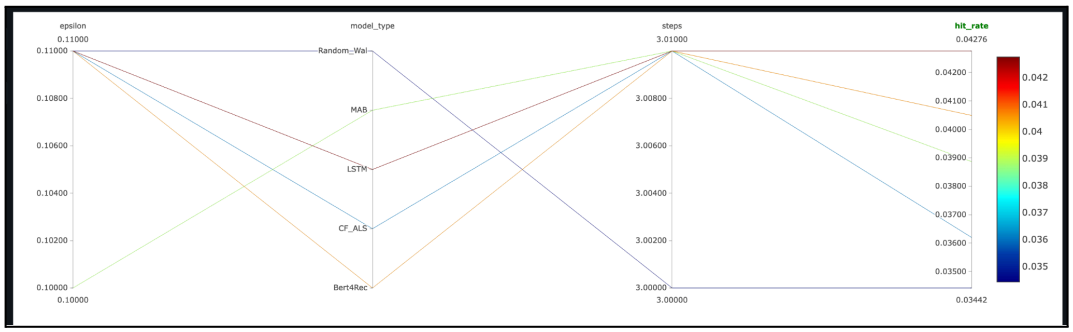


Figure F8: Cluster 1

Cluster 2

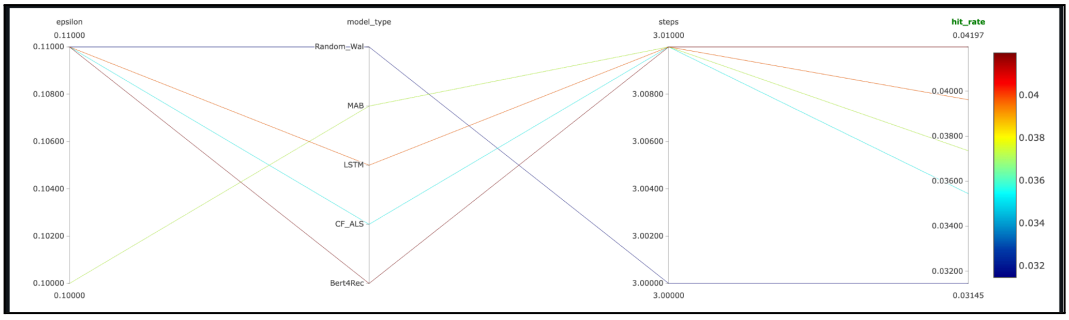


Figure F9: Cluster 2

Cluster 3

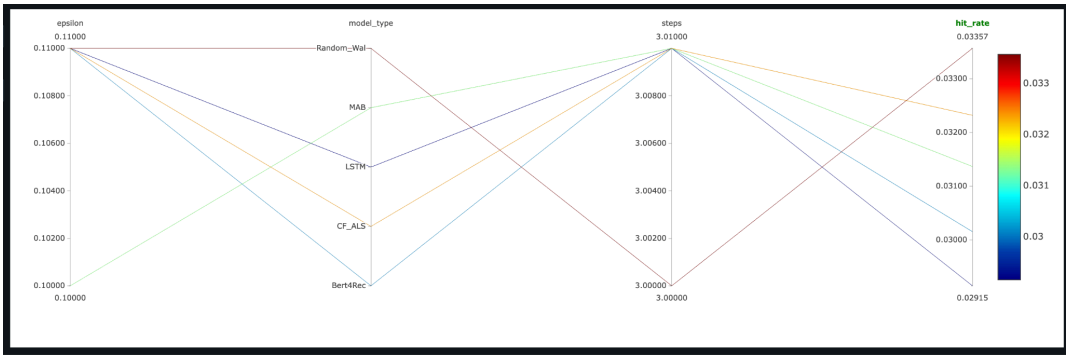


Figure F10: Cluster 3

Cluster 4

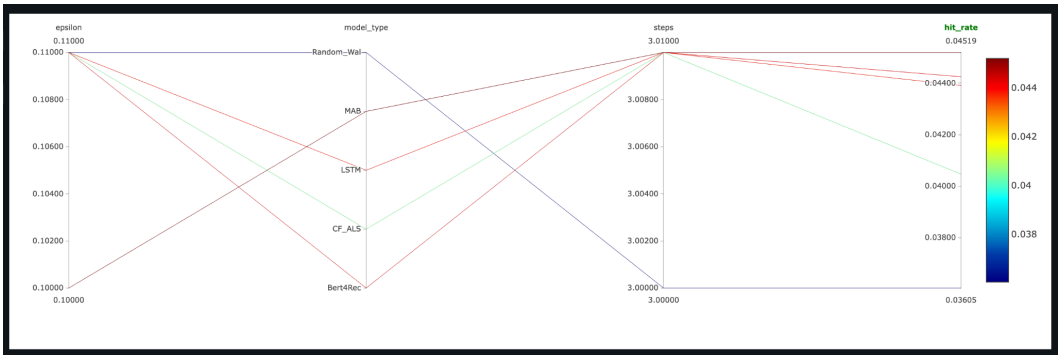


Figure F11: Cluster 4

Cluster 5

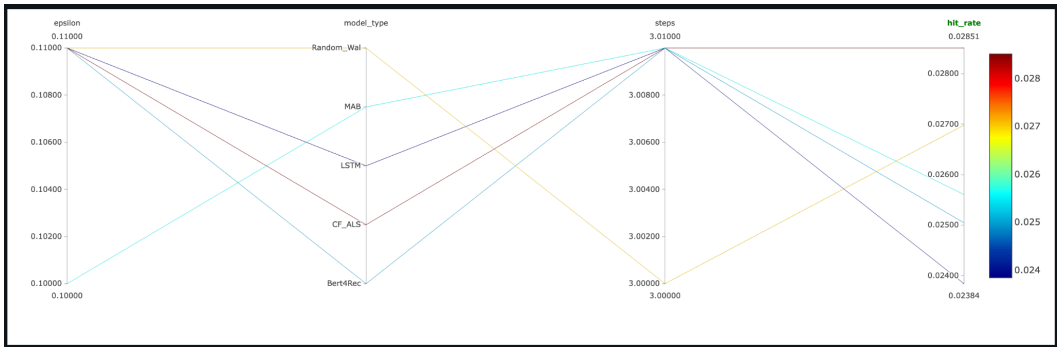


Figure F12: Cluster 5

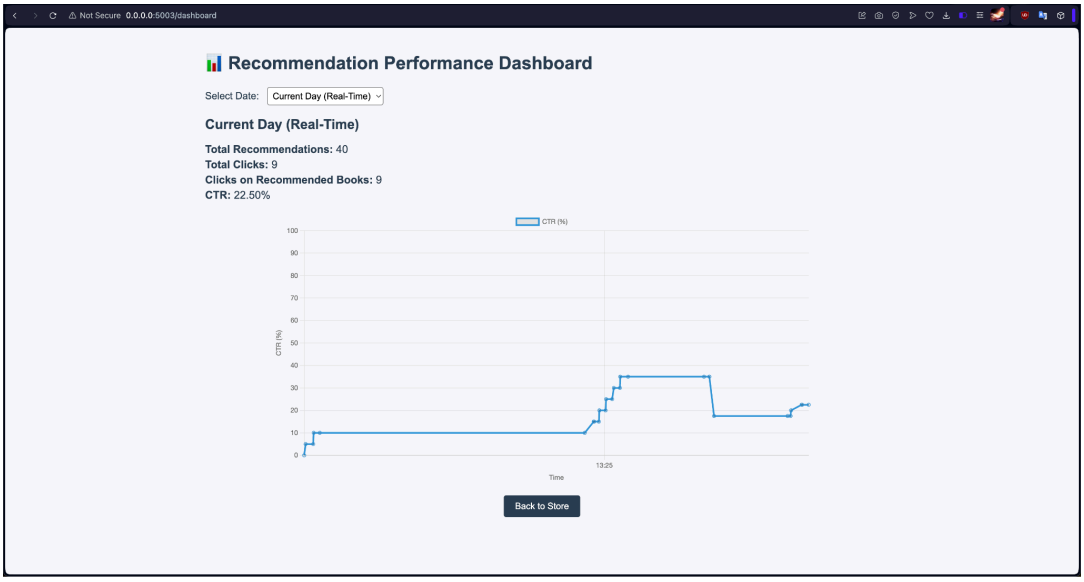


Figure F13: Additional Dashboard Page

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.google.cloud.operators.bigquery import BigQueryExecuteQueryOperat
from airflow.providers.docker.operators.docker import DockerOperator
from airflow.operators.dummy import DummyOperator

# Default arguments for the DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Define the DAG
with DAG(
    'retrain_models_from_bigquery',
    default_args=default_args,
    description='A DAG to read data from BigQuery and retrain models using Docker contai
    schedule_interval=timedelta(days=1),  # Run daily
    start_date=datetime(2025, 4, 8),
    catchup=False,
) as dag:
```

Figure F14: Retraining DAG

```python
# Log and print results
metrics = {
    "Hit Rate": hit_rate,
    "NDCG": ndcg,
    "MRR": mrr,
    "Accuracy": accuracy,
    "Precision": precision,
    "F1 Score": f1,
    "AUC-ROC": auc_roc
}

print(f"\nEvaluation Metrics (Top {top_k}):")
for metric, value in metrics.items():
    print(f"{metric}: {value:.4f}")
```

Figure F15: Model Metrics

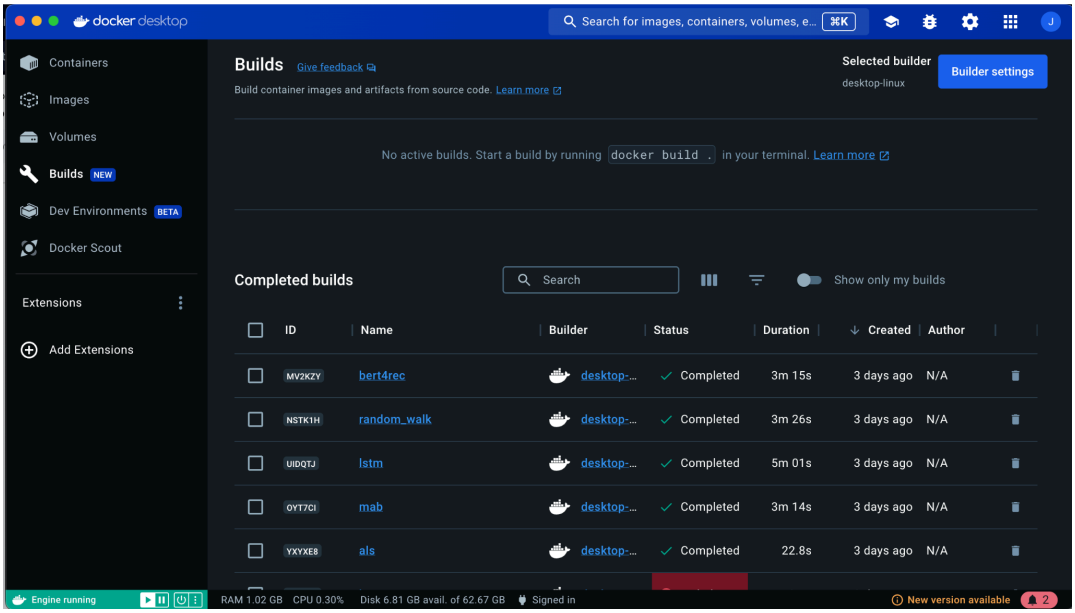


Figure F16: Docker Env containing the ML models

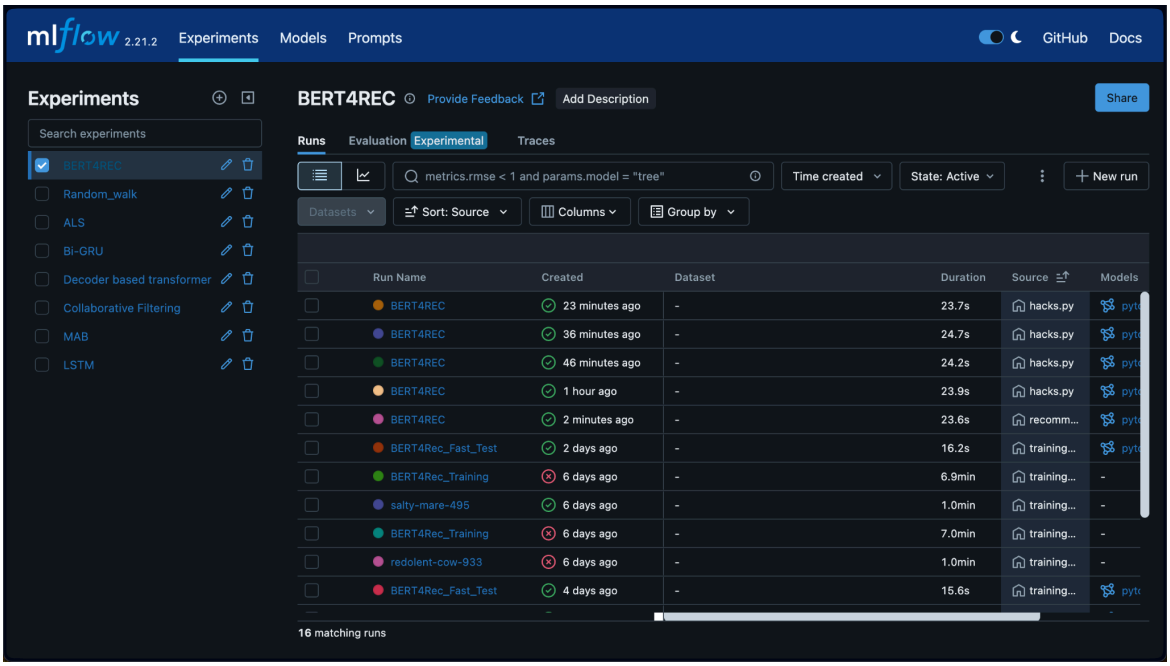


Figure F17: Monitoring and retraining deployed models

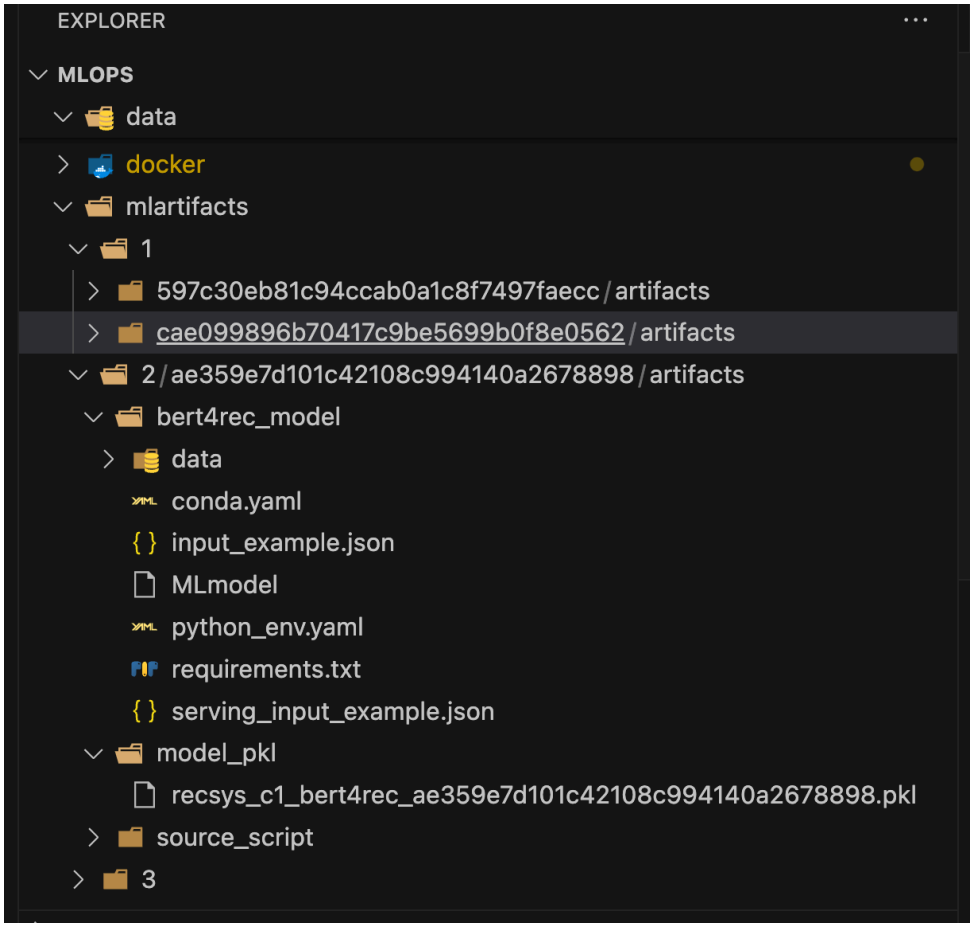


Figure F18: Artifacts stored by mlflow each run

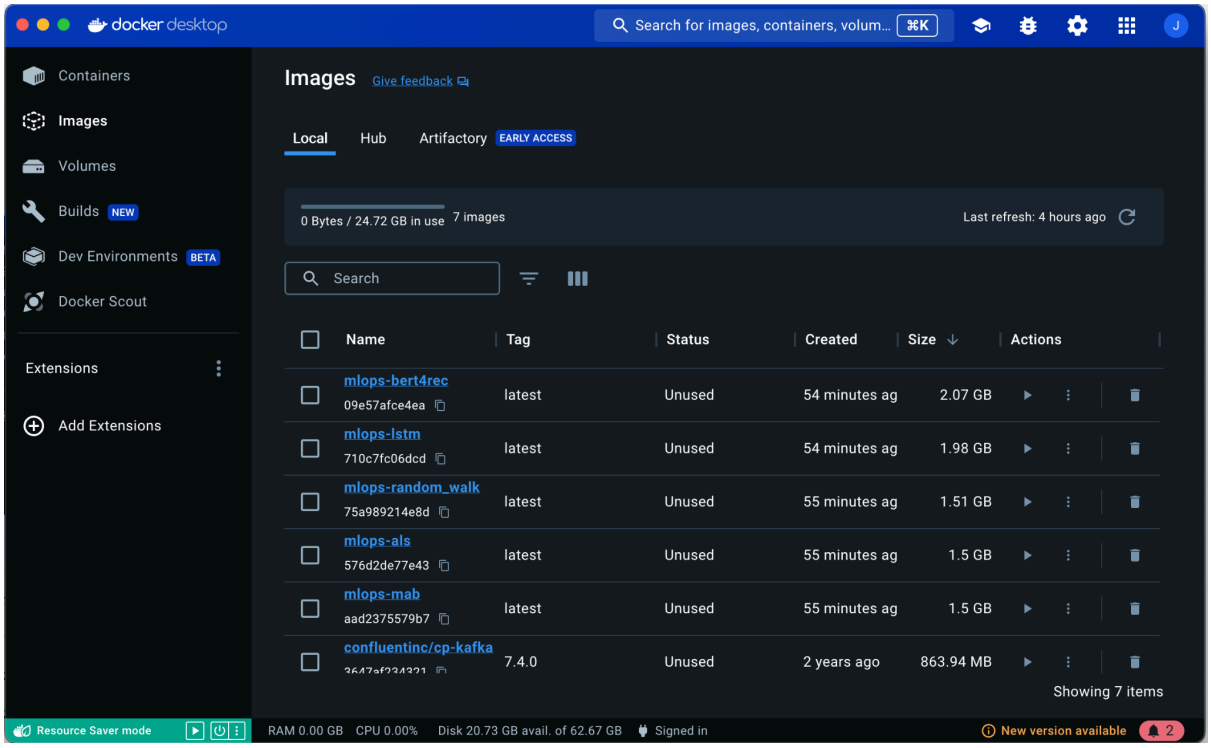


Figure F19: Docker Images for production run