Faculty of Engineering

Department of Electrical and Computer Engineering

NeAR Me

Final Report

2023.12

Alice Ye, 20700916

Michael Sawyer, 20765348

Shizhen Li, 20785413

Aidan Foster, 20721410

Pouya Mehrannia

March 22, 2023

I. Abstract

Google data from 2019-2021 shows that searches for "open now near me" have grown 400% year-over-year. More and more, people are using digital maps not only for navigation, but also to explore and learn about their surroundings. However, with existing map and navigation mobile applications, exploring and learning about the surroundings is far from a smooth experience while on the move. Try searching, swiping, spreading, pinching, pressing, tapping, reading and reorienting while negotiating busy streets and uneven terrain! NeAR Me is a mobile application that uses augmented reality (AR) to enrich people's travelling experience by rendering information about their surroundings in real-time, right in front of their cameras. Users can post AR content to topics at their locations for other users to see, and subscribe to topics of interest (e.g., restaurants, concerts, must-visit places, anything) to have AR content from those topics rendered around them. NeAR Me provides users with a highly visual experience, allowing them to place their content in the world and explore it with ease!

II. Acknowledgements

We would like to thank Prof. Pouya Mehrannia, our consultant, for his continuous support and guidance throughout the duration of the project.

III. Table of Contents

IV. List of Figures

V. List of Tables

1 Overview

1.1 Motivation

Nowadays, with about 84% of the world population owning a smartphone [1], people rely on digital devices to learn about and connect to the surrounding world more than ever. Statistics show an increasing trend that people continue to rely on the map and navigation applications on their mobile phones [2]. Google data from March 2022 demonstrates that “open now near me” searches have grown globally by over 400% year-over-year [3]. This means that people are no longer using digital maps to just look up directions, but are using them more and more to learn about interesting places near them while they are on the move. The problem is that, with the existing map and navigation mobile applications, exploring the surroundings is not a very smooth experience while one is on the move; they need to stop, search for what’s happening near them (possibly in another application), and then pick a destination to go to. Why can’t we make this experience smoother? As mobile augmented reality (AR) gains popularity worldwide [4], it is a perfect option to present location-based information in a convenient way while on the move. NeAR Me uses AR to enrich people’s experience while they are on the move by rendering information about their surroundings in real-time, right in front of their camera.

1.2 Project Objectives

The objective of NeAR Me is to eliminate the need for people to look up information about their surroundings while they are on the go. Users can subscribe to, or publish location-based information to, topics of interest. As users move around, information about places or things near them, from topics they are subscribed to, are displayed in AR, in real-time.

1.3 Block Diagram

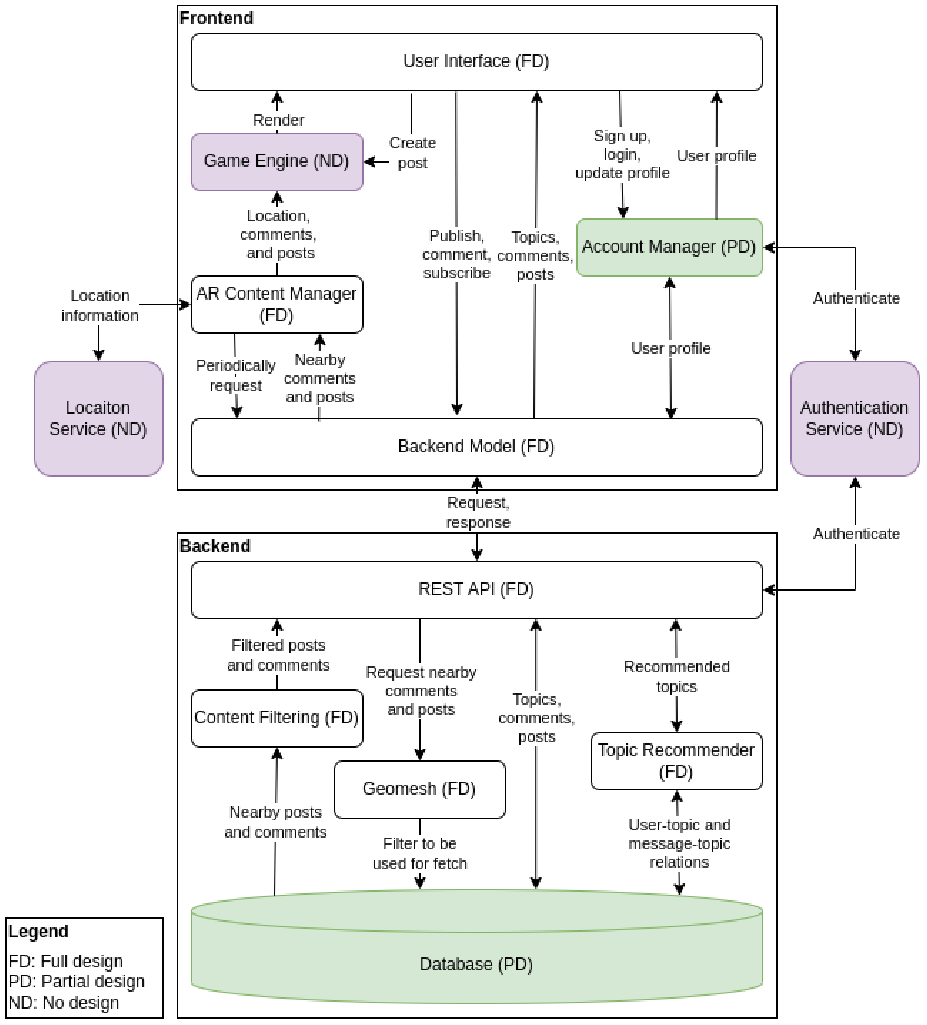


Figure 1-1 Block diagram of the NeAR Me subsystems

The following sections of this article are arranged as follows: in section 2, we list the specifications for this project; in section 3, we discuss the frontend and backend module design decisions and iterations in detail; in section 4, we show the methods and results we used to test and validate the performance specifications of our project; finally, in section 5, we conclude our project.

2 Project Specifications

2.1 Functional Specifications

Table 2-1 Functional specification of NeAR Me project

ID	Subsystem	Description	Necessity
FS1	UI, Account Manager, Authentication Service	Users must be able to create a new account with a username and password.	Essential

FS2	UI, Account Manager, Authentication Service	Users must be able to login with the correct username and password.	Essential
FS3	UI, Backend Model, REST API, Database	Users must be able to subscribe to topics from a set of default topics (e.g. #food, #shopping, #music).	Essential
FS4	UI, Backend Model, REST API, Database	Users must be able to create and publish/subscribe to custom topics.	Essential
FS5	UI, Backend Model, REST API, Topic Recommender	Users must be able to receive topic recommendations for topics they are not subscribed to.	Non-Essential
FS6	UI, Backend Model, REST API, Database	Users must be able to become authorized to publish to topics at a specific location.	Essential
FS7	UI, Backend Model, REST API, Database	Users must be able to publish to topics only at locations they are authorized to publish at.	Essential
FS8	UI, Backend Model, REST API, Database	Users must be able to comment on any topic at any location.	Essential
FS9	REST API	Users must be able to see only information from topics that they are subscribed to.	Essential
FS10	AR Content Manager, Game Engine, UI	Users must be able to read comments.	Essential
FS11	UI, AR Content Manager, Content Filtering	Users must be able to change the number of entries rendered in AR to between 1 and 100 entries.	Essential
FS12	AR Content Manager, Geomesh	Users must be able to see content in AR that is up to 100 m away.	Essential
FS13	AR Content Manager, Backend Model, REST API, Geomesh, Content Filtering	Users must be able to see new AR content appear within 1 s of them moving.	Essential
FS14	UI, Backend Model, REST API, Database	Users must be able to like/dislike the displayed contents.	Non-Essential
FS15	UI, Account Manager, Authentication Service, Database	Users must be able to change their profile avatar, username, and description.	Non-Essential
FS16	AR Content Manager, Game Engine, Database	Users must be able to adjust the AR content's GPS location and size before publishing it to a topic.	Non-Essential
FS17	Account Manager, Authentication Service	Users must have to login only once.	Non-Essential
FS18	Content Filtering	Users can filter the displayed contents by preference (i.e. preference on topics, sort by newest, most-liked, etc.)	Non-Essential
FS19	Game Engine	Users must not see AR content if it is behind a physical object.	Essential
FS20	UI, Game Engine	Users must be able to publish entries as large as 2.5 m by 2.5 m.	Essential

FS21	UI	Users must be able to insert text input for publication with an upper bound of 140 characters.	Essential
FS22	UI	Users must be able to insert image input for publication with a 5 MB upper bound on image size.	Non-Essential

2.2 Non-functional Specifications.

Table 2-2 NeAR Me non-functional sepcifications

ID	**Description**	**Essential/Non-Essential**
NFS1	The backend must be scalable across multiple servers, at least one per region served, to ensure service availability.	Essential
NFS2	Users can be served by the app in the case of a single server failure.	Essential
NFS3	The mobile application must be compatible with most AR-capable models of Android devices running Android 7 or newer.	Essential
NFS4	The total cost of the project should not exceed 600 CAD.	Non-Essential
NFS5	The user interface should provide a smooth user experience; easy to navigate.	Non-Essential
NFS6	Users must be able to change the font size for the UI text display.	Non-Essential
NFS7	Users must be able to publish up to 1 topic entry per minute.	Non-Essential
NFS8	Users must be able to see AR content within 5 m of its actual location.	Essential

3 Detailed Design

3.1 Frontend

There are four external modules that are connected to our frontend:

- **Game Engine**: This module renders the UI components, and AR comments and posts.
- **Location service**: This module provides the user's current geographical position.
- **Authentication service**: This module registers and authenticates users. It provides the frontend with an authentication token, which is then sent to the backend along with each request. The interaction between this module and the backend is discussed in Section 3.2.
- **Backend**: This module serves the frontend data from the database upon request.

The frontend ties these external modules together, delivers AR content to the user's screen, and achieves the specifications listed in Section 2. At a high level, the frontend has three main functions: managing the AR content around the user, handling UI interactions, and communicating with the different external modules. The following parts of this section are arranged as follows: first, we discuss how the AR Content Manager manages AR content around the user; then, we discuss the UI flow; finally, we discuss the design decisions we made when selecting and designing the different external modules.

3.1.1 AR Content Manager

The AR Content Manager is responsible for managing which comments/posts are rendered to the user's screen and where on the screen they are rendered to. It uses two pairs of lists (one pair for comments and the other for posts), four different routines, and the user's current location to do so.

In the pair of lists, one list, **comments_list** or **posts_list**, stores the *inactive* comments or posts near the user, while the other, **active_comments** or **active_posts**, stores the *active* ones. Active comments/posts are those that are currently being rendered to the user's screen.

The four routines are **fetch()**, **update_comments()**, **update_posts()**, and **render_comment()**. These routines run concurrently. Two instances of **fetch()** are used to request comments/posts from the backend and update the contents of **comments_list** and **posts_list**. The **update()** routines are used to update the contents of **active_comments** and **active_posts** based on the user's current location. The **render_comment()** routine renders **active_comments** on the user's screen, while the game engine renders **active_posts**. The pseudo-code for these routines follows:

```
fetch():
  every 500 ms:
    request 100 nearest comments/posts from backend
    update comments_list/posts_list
    if comments_list/posts_list size > MAX_SIZE:
      sort comments_list/posts_list by distance from user's current location
      remove furthest entries from comments_list/posts_list until size == MAX_SIZE

update_comments():
  loop:
    wait until active_comments size < MAX_ACTIVE_COMMENTS
    if location was updated:
      sort comments_list by distance from user's current location
    if first commment in comments_list is within 100 m of user's location (FS12):
      move comment from comments_list to active_comments
      render_comment()

update_posts():
  loop:
    wait until location is updated
    move all active_posts to posts_list
    sort posts_list by distance from user's current location
    for the first posts in posts_list within 100 m of user's location (FS12):
      move post to active_posts
      tell game engine to render post at its location

render_comment():
  render comment starting off the right edge of the user's screen
  every frame until the comment moves off the left edge of the user's screen:
    move comment left a small distance so it appears to move at a constant speed
  stop rendering comment
  move comment from active_comments to comments_list
```

3.1.2 User Interface

The user interface is built with Unity and consists of the following parts:

- User authentication scenes (login, register, reset password);
- The main scene (AR camera);
- Topic publication and subscriptions scenes;
- The user homepage scene (user profile, subscriptions, publications, custom topics, locations);
- The app settings scene.

The transitions between scenes are described in Figure 3-1 below and an explanation with screenshots is provided in Section 4.1.

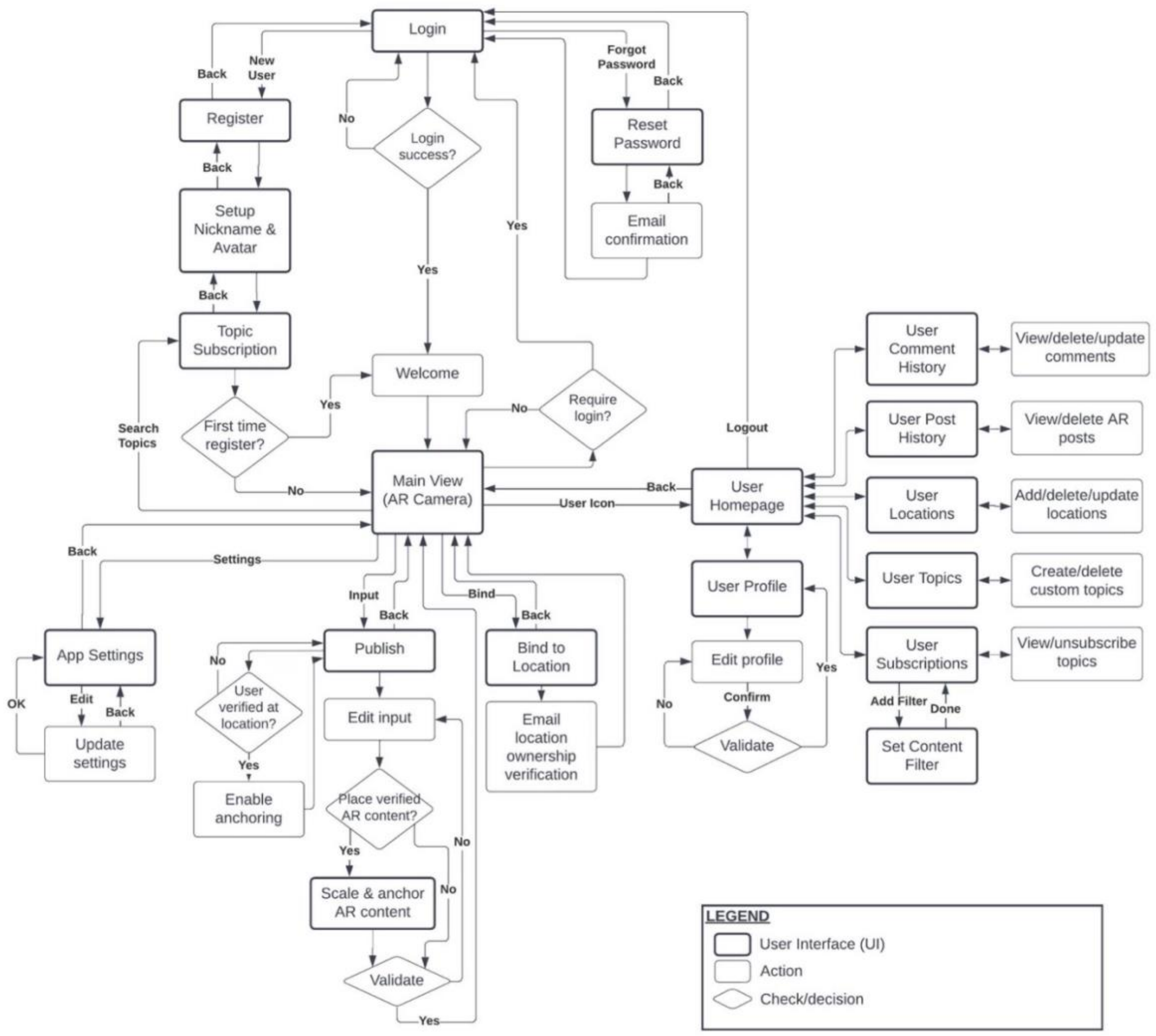


Figure 3-1 NeAR Me frontend UI flow

3.1.3 Account Manager

The account manager is responsible for user authentication and user management.

We use a third-party service for secure user authentication to speed up our deployment process and focus on the core logic implementation. We compare the two most popular authentication services for mobile apps in Table 3-1 and rank the options in a decision matrix in Table 3-2 After considering the options, we select Firebase as our authentication service because it is well-documented, easy to integrate, and free to use.

Table 3-1 Comparison of user authentication approaches

	Firebase	**Auth0**	**Custom Implementation**
Pros	Good documentation; many sample projects.	Easy to integrate.	Independent of 3rd party; full control of data flow and data privacy.
	High availability.	Good MFA support.	No cost.
	Easy to integrate.	Data integrity.	Optimizable query.
	No cost (free tier usage).		
	Data integrity.		
Cons	Dependent on 3rd party (Google).	Expensive.	Encryption complexity; protect our own data.
	Limited support for IOS.	Fast changing APIs.	Extra implementation effort on the backend.
	Limited querying capabilities.	Dependent on 3rd party.	Difficult to add-on MFA.
	Difficult to predict future cost.	Poor developer support.	Complexity in data syncing.

Table 3-2 Decision matrix for the user authentication approaches

	Firebase	**Auth0**	**Custom Implementation**
Ease of implantation	2	1	0
Cost (cheap = high score)	1	0	1
Security	2	2	1
Total	**5**	**3**	**2**

The account manager fits into our system as follows:

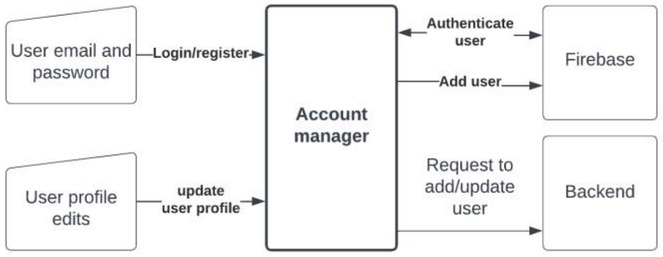


Figure 3-2 Account manager integration

When the user attempts to login, the account manager uses the Firebase APIs to authenticate the user (FS2) and to maintain a persistent login session (FS17). When a new user is registered, the account manager adds the user to the Firebase database. When the user profile is modified from the UI, the account manager makes an API request to the backend to update the profile in the database.

3.2 Backend

Our specifications are mostly concerned with the user's experience with our system, and the backend is not directly concerned with the user's experience. However, though no backend specifications are explicitly outlined, it still must support the frontend in realizing our specifications. Thus, the backend design is focused on minimizing external module cost and increasing internal computation speed.

The backend is connected to three external modules:

- **Authentication service**: The interaction between this module and the frontend is discussed in Section 3.1. This module resolves the authentication token into a user identifier, which the backend uses when querying the database.
- **Backend framework**: This module's main functions are mapping requests from the frontend to different backend functions, and arranging the workload based on server resources. In practice, it also provides guidelines for our code structure and implementation.
- **Database**: This module stores the data required by the frontend in the data structures we designed and allows searching for this data through different criteria.

To achieve the frontend's needs, we first designed our backend as follows:

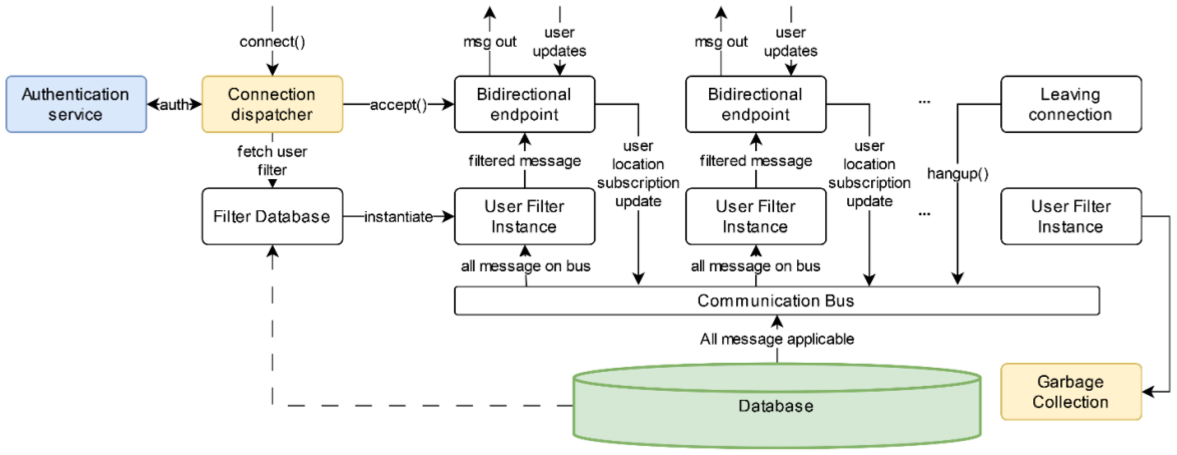


Figure 3-3 First design of backend interconnection

This design has the following problems that need to be addressed:

- Waste of bandwidth: the same data from database is copied to every connected user's filter instance, and a very small subset of it is useful.
- Waste of memory: every newly connected user has a filter instance created, which can easily overwhelm memory.

- Computationally intensive: the system maintains an open connection for every user, and every user update causes a change in the whole system. This is especially problematic for location-sensitive messages.

To address these problems, we re-designed the backend as follows:

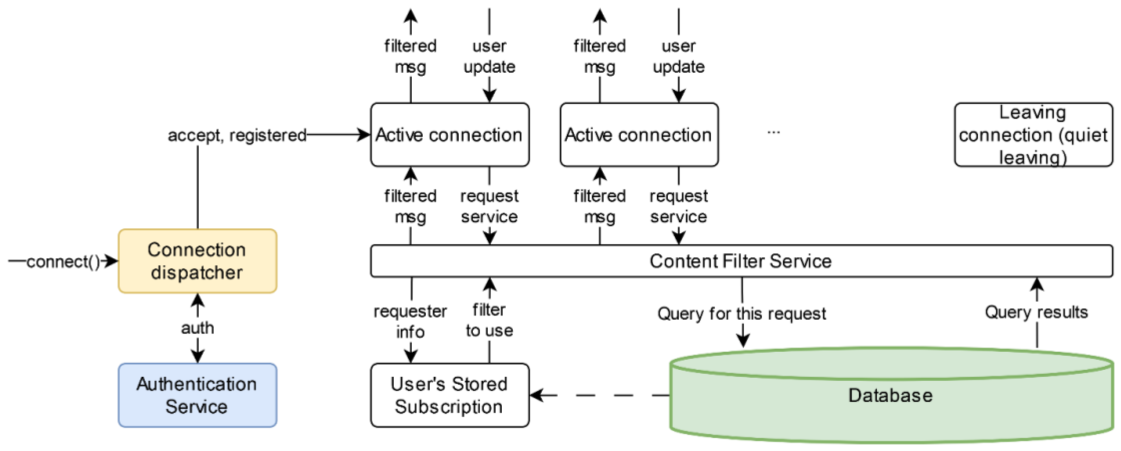


Figure 3-4 Second design of backend interconnection

This design considers the sporadic nature of user requests, requiring way less data communication and filter service instances. However, it has the following problems:

- Not suited for mobile use: because Near Me is a mobile app that the user uses while moving, connections usually need frequent resetting, which need to go through the dispatcher each time.
- Data duplication in the frontend: with only current location information from the frontend, the database will fetch all nearby messages, many of which already exist on the user’s screen.

To address these issues, we re-designed the backend to use a stateless HTTP model as follows:

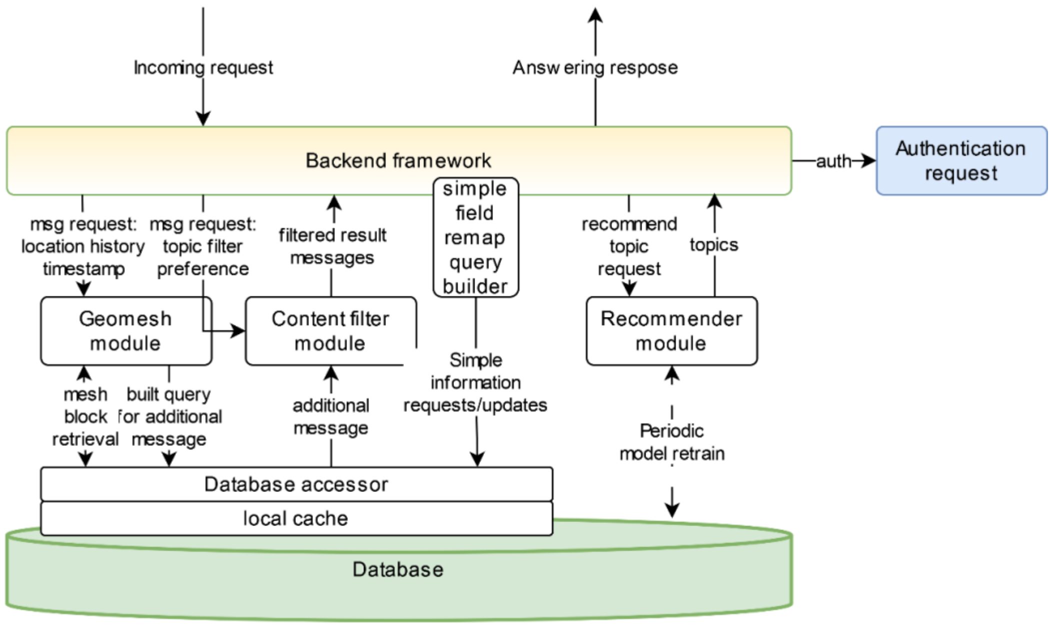


Figure 3-5 Final design of backend interconnection

The remaining parts of this section are arranged as follows: first, we discuss the decisions we made about external modules; then, we elaborate on the different submodules in the backend.

3.2.1 Framework

The purpose of having a backend is to process and answer user requests that require complex computation and database communication. Under the assumption that the backend must handle many user requests from all over the world and respond with near-real-time latency, the backend framework must be optimized (high concurrency and database access speed), scalable, and reliable. Given the time constraint, we favor the backend framework that is easy to interface with, integrate, and implement.

In Table 3-3, we compare the two most popular backend frameworks, and consider a serverless cloud approach (an approach where we deploy the backend functions and database as cloud functions rather than managing a server ourselves). Based on this comparison, we decide to use **Java Spring Boot** as our backend framework.

Table 3-3 Decision matrix for backend framework

	Java Spring Boot	**Python Django**	**Serverless Cloud Deployment**
Response speed	3	2	2
Ease of use	2	2	3
Scalability	2	1	3
Reliability	3	2	1
Total	10	7	9

We first compare Java Spring Boot and Python Django. Spring and Django are both easy-to-use frameworks with good scalability; however, compared to the high concurrency (multithreading) support in Java Spring Boot, Python Django is slower in terms of processing and can only handle one HTTP request at a time. Java Spring Boot is more scalable and reliable than Python Django because Spring is more opinionated and modularized than Django [8]. By default, Django templates handle errors quietly, and the REST framework can only handle one HTTP request at a time [9]. Next, we compare the serverless cloud deployment options with Java Spring Boot and Python Django, based on team member experience (Google/AWS cloud). The serverless option scores the highest in terms of ease of use and scalability because we do not need to manage our own server and infrastructure, and computing resources will be allocated and charged per request. However, in exchange for ease of use and automated scaling, the response time might not be guaranteed because resource allocation is dependent on a third party. For this same reason, and with the concern of unexpected cost, the serverless option scores the lowest in terms of reliability.

3.2.2 Database

For the database, we consider the following aspects to be the most important to our system:

Table 3-4 Aspects, weights, and reasons for database decision matrix

Aspect	Weight	Reason
Query by GPS speed	3	Positioning objects in AR requires accurate position data, which necessitates that position-based queries are done quickly, and since that is the focus of the app, this aspect should have the most weight
Query by Username speed	2	In order to expediently deliver subscribed-to content to the user, the system needs to be able to quickly retrieve the list of the user's subscriptions
Query by Topic Speed	2	In order to filter content to display to the user, the system will need to quickly return contents pertaining to a specific topic
Scalability to multiple servers	1	In order to maintain fast query speeds as user counts increase, the requests may have to be spread across multiple servers, but this is not the most vital to operation (NFS1)
Comment post/query speed	1	Since comments cannot all be viewed at the same time, the speed at which a comment can be posted/queried is not the greatest concern in choosing a database design

We consider the following possible database designs/choices:

- **Design 1**: Separate databases for each topic, or databases indexed by topic
- **Design 2**: Separate databases per geographic region, indexed by location
- **Design 3**: Single monolithic database for all backend data

Table 3-5 Decision matrix for database designs

Aspect (Weight)	Design 1	Design 2	Design 3
Query by GPS speed (3)	1	3	2
Query by Username speed (2)	1	3	2
Query by Topic Speed (2)	3	2	2
Scalability to multiple servers (1)	2	3	1
Comment post/view speed (1)	2	2	1

Total Score	15	25	16

From the above table, Design 2, where databases are separated by geographical area, and content is indexed by geographic area, appears to be the best option, so that is what we choose.

For the choice of database infrastructure, we compare the most predominant open-source database solutions, namely, MongoDB, Redis, and PostgreSQL:

Table 3-6 General Comparison of Database Platform Options

	MongoDB	**Redis**	**PostgreSQL**
Pros	Sharding support [17] (NFS1) Free "sandbox" for design [16] Replication and auto-failover [18] (NFS2) Pay-per-read "serverless" option [16] Fully JSON operation [21] Allegedly faster than Redis for very large databases [21]	Embedded pub-sub with pattern matching Cheap dedicated servers with replication relatively cheap [19] Developer GUI Free tier available [19] Key-value pair operation [21]. "Master-master" replication besides "master-slave" [18]	Experience with relational databases Free database program Cross-platform support
Cons	Lack of experience with NoSQL databases Expensive dedicated server [16]	Lack of experience with NoSQL databases Manual actions scaling up Free tier only includes a single database [19], limiting testing strategy Worse JSON support than MongoDB (from personal testing) Poor documentation (from personal testing)	Dedicated servers Limited scaling support [17] Low flexibility [20]

As designing and prototyping requires frequent changes to our implementation, PostgreSQL with its less flexible structure is quickly ruled out. We dive deeper into the difference between MongoDB and Redis.

To further compare between MongoDB and Redis, a benchmark script was developed in Python that randomly generates 10000 AR content database entries in JSON format (more entries would be more desirable, but the Redis free tier has only 30MB of storage [19]), connects to a free-tier database for each option, uses a specified number of threads (each representing a user) to write all of those entries in each database, averages the time taken for each database transaction to get the write speed, then filters the databases for text-based AR content entries (see entry spec below) to get the filter speed. The following data was collected with 25 threads (so each thread sequentially makes 400 writes to each database):

Table 3-7 Database Access Speed MongoDB vs Redis

	MongoDB	**Redis**
Average Write Speed (ms)	53.9	53.4
Filter Speed (ms)	~0.0	2.4

It must be noted that the filter speeds cannot be considered in isolation, only as part of a comparison, as the fraction of entries that are text-based is randomly determined before the tests (both databases have identical data sets). Only 25 threads were used because the free tier of Redis has a limit where up to only 30 connections can access the database concurrently [19].

While MongoDB is ever so slightly slower than Redis for writing single entries to the database, most of the database interactions are going to be reads, as a user is almost constantly reading entries for content regarding their surroundings but will write comments/content to the database comparatively infrequently. The increased need for fast reads makes MongoDB the better option, as in the benchmarking, searching the (admittedly small) database for a particular string was practically instant with MongoDB, whereas with Redis the same read took multiple milliseconds.

Therefore, we decide to use **MongoDB** for the backend database.

3.2.3 Geomesh Module

As an initial naïve design of providing the frontend with surrounding messages, it would simply be a Euclidean distance filter applied to all messages in database. However, it quickly turns out to be too much computation to satisfy FS13. Thus, we designed the Geomesh module to group user messages into geographical blocks. By doing so, we can query for surrounding messages by finding surrounding geomeshs and querying message based on that. Geomesh also serves as a great way to tell if the frontend has pulled a message.

Posts, while sparse over the whole map, especially outside of cities, are likely to cluster around shopping districts in city centers. Geomesh won't help if all of messages belong to one block. To alleviate this problem, geomesh block resolution needs to be adaptive. To achieve this, we use the following code for geomesh splitting:

```
Geomeshs = fetchAllFromDatabase()
while (Geomeshs.hasNext()) :
    mesh = Geomeshs.Next()
    msgs = fetchMsgFromGeomesh(mesh)
    if (msgs.count > THREASHOLD && mesh.length > MIN_LENGTH) :
        // splitting mesh (square) into 4 (squares)
        [lat, lng, length] = mesh   // lat, lng based on upperleft corner
        mesh1 = [lat, lng, length/2]
        mesh2 = [lat - length/2, lng, length/2]
        mesh3 = [lat, lng + length/2, length/2]
        mesh4 = [lat - length/2, lng + length/2, length/2]
        storeMeshs(mesh1, mesh2, mesh3, mesh4)  // NOTE mesh1 updates mesh
        // reassign message's geomesh ID so reference still valid
        updateMsgBelonging(msgs, mesh1, mesh2, mesh3, mesh4)
        // add to processing stack in case more split needed
        Geomeshs.add(mesh1, mesh2, mesh3, mesh4)
RerunAboveAfter(INTERVAL, hours);
```

This is a task that runs every 4 hours. It splits geomesh blocks until the message count in each block is less than the threshold (100 in the current project setup) or until the minimum block size is reached (50m length in the current setup). A sample split (with different constants for illustration purpose) result is shown below:

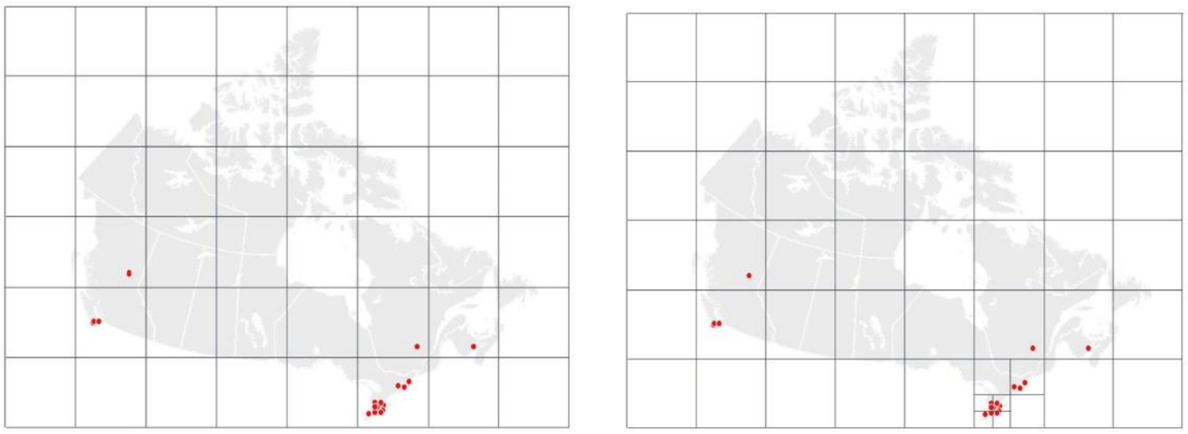


Figure 3-6 Geomesh split result after user posts (red dot)

While splitting is a mostly inactive periodic task, the geomesh module also actively processes user requests. This is described in the following pseudo code:

```
lastSurroundingBox = [lastLat, lastLng, RADIUS]
userSurroundingBox = [userLat, userLng, RADIUS]
lastSurrounding = GeomeshDatabase.query(mesh => {AND:
   [lastLat - RADIUS < mesh.top,
    lastLat + RADIUS > mesh.bottom,
    lastLng - RADIUS < mesh.right,
    lastLng + RADIUS > mesh.left]})
userSurrounding = GeomeshDatabase.query(mesh => {AND:
   [userLat - RADIUS < mesh.top,
    userLat + RADIUS > mesh.bottom,
    userLng - RADIUS < mesh.right,
    userLng + RADIUS > mesh.left]})
newMeshes = userSurrounding.exclude(lastSurrounding)
```

Note that the query passed to the database is composed of 4 logic comparators and 1 AND gate, which is way simpler than Euclidean distance calculation. The method also avoids duplicated messages by excluding blocks last fetched. A sample execution result is illustrated below:

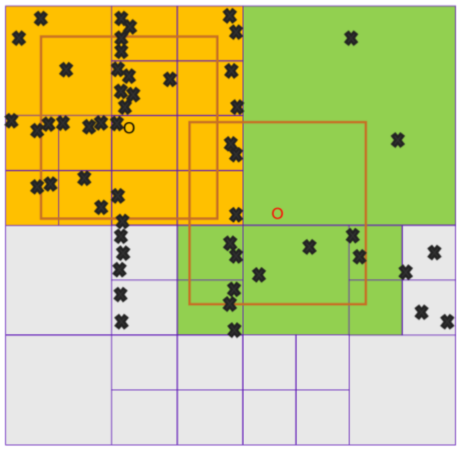


Figure 3-7 New geomesh (green) to fetch message from

This module is one of many efforts from the backend in satisfying FS13.

3.2.4 Content Filtering System

The content filtering system is responsible for reducing the number of entries contained in a response to the frontend. The number of entries per response must be limited because a given user can have an unbounded number of subscriptions, but the number of entries contained in a response must be bounded. It is not clear which entries should be rejected and which should be selected, so we have decided to give the choice to the user. The content filtering system can filter entries using one of the following conditions: most-liked, most-recent, and unsorted. This satisfies FS18.

The content filtering system uses a min-heap for determining which entries are returned to the frontend with each fetch content request. The heap is a binary tree structure, where every node on the tree has a value that is less than both of its children, which results in the minimum value in the heap being the root node of the tree, allowing for easy replacement when adding a value to a full tree.

In the frontend UI settings menu, a user can set the "display density" (the maximum number of entries stored in the heap) to integers between 1 and 100, select up to three topics to be given extra weight in the filtering, and select how the posts are sorted (most-liked, most-recent, or "unsorted"). Each added "preferred topic" getting the weights 2, 3, and 4 respectively, whereas by default, every topic has a weight of 1. When deciding whether a post gets added to a full heap and when sorting the heap, the values used in the comparisons between heap entries are the values being sorted on (likes for "most-likes, Unix timestamps for "most-recent", and 1 for "unsorted) multiplied by the weights of the entry topics. This ensures that in addition to reducing the number of entries returned, those entries returned are also more relevant to the user.

Example illustrations of how a post gets added to a non-full heap and how a post gets added to a full heap are shown below in Figure 3-8, and the filter algorithm can be described with the following pseudocode (the "min_heapify" function described but not shown, for brevity):

```
function filter(new posts, max_heap_size, filter_type):
  if filter_type == "most liked":
    metric is "like_count * weight"
  else if filter_type == "most recent":
    metric is "unix_timestamp * weight"
  else:
    metric is "weight"
  heap = new list
  for each new post:
    if heap size == max_heap_size:
      boolean minimum_must_be_discarded = (new_post_metric > heap_root_metric)
          if minimum_must_be_discarded:
            replace heap minimum (first element) with last element (Error! Reference
source not found.c)
            // recursively swap heap element with the lesser of its children,
            // stopping if both children are greater than the element (Error! Reference
source not found.d)
            min_heapify(root index 0, heap size)
    if heap size < max_heap_size or minimum_must_be_discarded:
      try to add new post to heap list (Error! Reference source not found.a and Error!
Reference source not found.d)
```

```
    while new post has a parent and parent_metric > new_metric:
      swap position of new post with its parent (Error! Reference source not found.b
and Error! Reference source not found.e)
  return heap
```

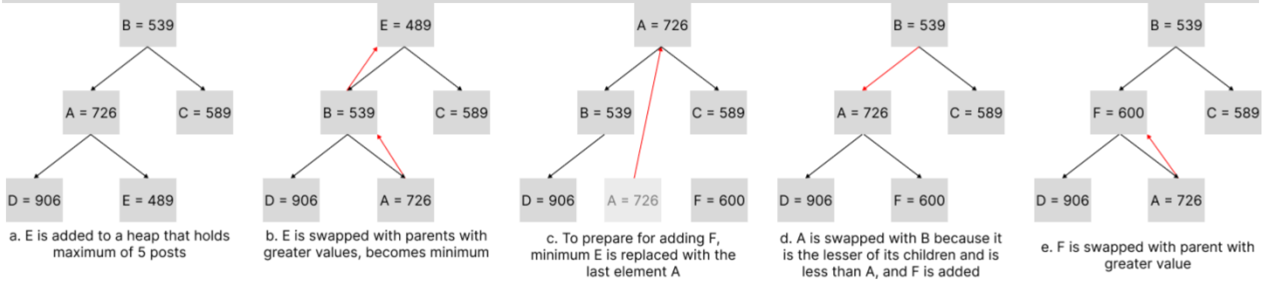

B = 539
A = 726
C = 589
D = 906
E = 489
a. E is added to a heap that holds maximum of 5 posts
E = 489
B = 539
C = 589
D = 906
A = 726
b. E is swapped with parents with greater values, becomes minimum
A = 726
B = 539
C = 589
D = 906
A = 726
F = 600
c. To prepare for adding F, minimum E is replaced with the last element A
B = 539
A = 726
C = 589
D = 906
F = 600
d. A is swapped with B because it is the lesser of its children and is less than A, and F is added
B = 539
F = 600
C = 589
D = 906
A = 726
e. F is swapped with parent with greater value

Figure 3-8 Process of adding entries to non-full heap and full heap

3.2.5 Topic Recommendation System

The topic recommendation system is responsible for suggesting new topics for users to subscribe to. The recommender algorithm can suggest based on topic similarity, user similarity, and topic popularity (FS5).

Referring to Figure 3-9, we design the topic recommendation system with hybrid methods (collaborative filtering + content-based filtering). We do not use the content-based methods alone because we do not have many usable features from the entities (user, topic) to model the relation, and we do not use the collaborative filtering approach alone because it can suffer cold start (newer topics not getting recommended).

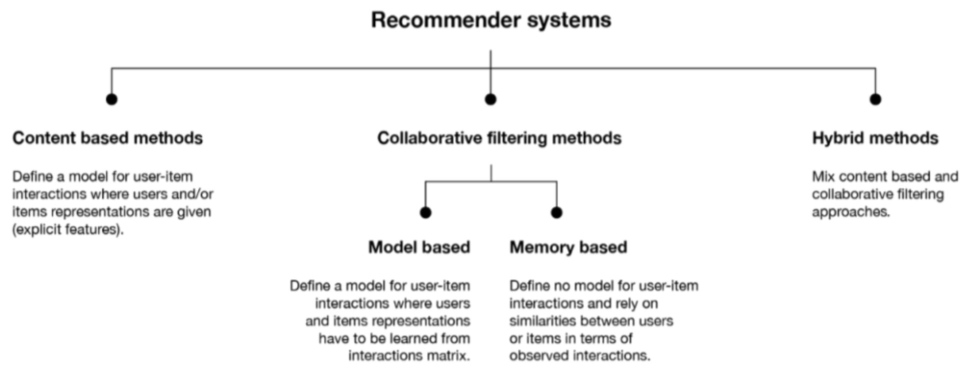

Recommender systems
Content based methods
Define a model for user-item interactions where users and/or items representations are given (explicit features).
Collaborative filtering methods
Model based
Define a model for user-item interactions where users and items representations have to be learned from interactions matrix.
Memory based
Define no model for user-item interactions and rely on similarities between users or items in terms of observed interactions.
Hybrid methods
Mix content based and collaborative filtering approaches.

Figure 3-9 Types of recommender systems [7]

Personalized topic recommendation requires users to have subscribed to topics. If a user has subscribed to none or too few topics, the recommender may not generate meaningful or any topic recommendation for the user; to prevent this situation, we add another layer to the topic recommender – a popularity filter. As shown in Figure 3-10, our resulting system attempts to generate topic recommendations for users from three directions: subscription-based user similarity, content-based topic similarity, and topic popularity.

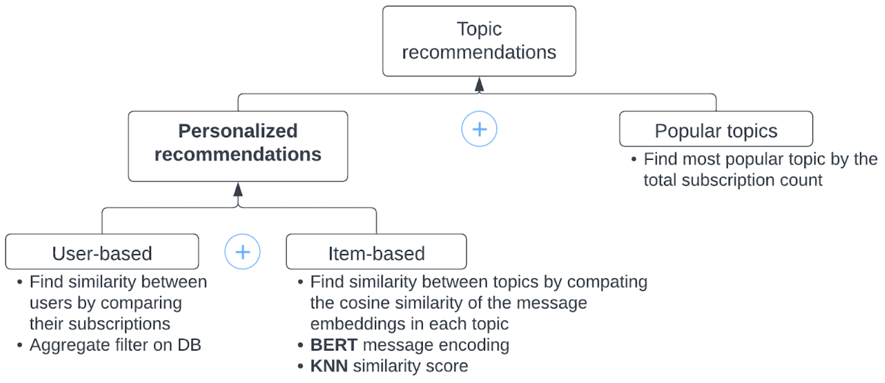

Topic recommendations
Personalized recommendations
Popular topics
• Find most popular topic by the total subscription count
User-based
• Find similarity between users by comparing their subscriptions
• Aggregate filter on DB
Item-based
• Find similarity between topics by compating the cosine similarity of the message embeddings in each topic
• BERT message encoding
• KNN similarity score

Figure 3-10 Topic recommender design

For a user U requesting topic recommendation in region R, the recommendation system's response logic is depicted in **Error! Reference source not found.** below.

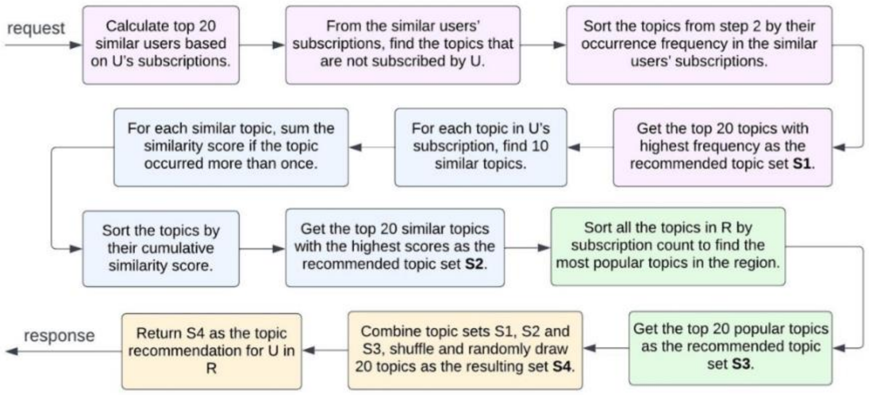

request
Calculate top 20 similar users based on U's subscriptions.
From the similar users' subscriptions, find the topics that are not subscribed by U.
Sort the topics from step 2 by their occurrence frequency in the similar users' subscriptions.
Get the top 20 topics with highest frequency as the recommended topic set S1.
For each topic in U's subscription, find 10 similar topics.
For each similar topic, sum the similarity score if the topic occurred more than once.
Sort the topics by their cumulative similarity score.
Get the top 20 similar topics with the highest scores as the recommended topic set S2.
Sort all the topics in R by subscription count to find the most popular topics in the region.
Get the top 20 popular topics as the recommended topic set S3.
Combine topic sets S1, S2 and S3, shuffle and randomly draw 20 topics as the resulting set S4.
Return S4 as the topic recommendation for U in R
response

Figure 3-11 Topic recommender logic overview

The user similarity is found by aggregated search on the database, utilizing features of MongoDB. For topic similarity calculation, we randomly sample min{100, existing} messages from each topic, encode the message text to vector using BERT (Bidirectional Encoder Representations form Transformers), and then find the similar topics by KNN and cosine similarity score. KNN is our chosen algorithm because we have a small initial dataset, and simple models such as KNN generally perform better on small datasets compared to more complex models. Text encoding is the most time-consuming step in the procedure, and to minimize latency in the recommender response, we introduce a two-level cache design.

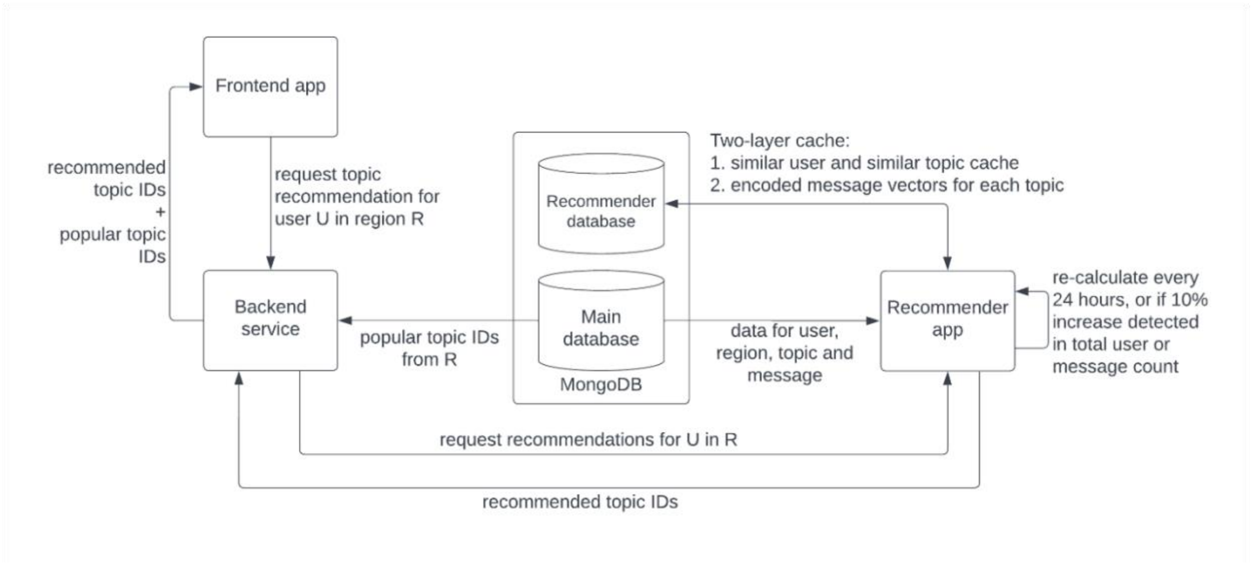

Frontend app
recommended topic IDs + popular topic IDs
request topic recommendation for user U in region R
Two-layer cache:
1. similar user and similar topic cache
2. encoded message vectors for each topic
Recommender database
Backend service
popular topic IDs from R
Main database
MongoDB
data for user, region, topic and message
Recommender app
re-calculate every 24 hours, or if 10% increase detected in total user or message count
request recommendations for U in R
recommended topic IDs

Figure 3-12 Topic recommender integration

The recommender has 3 cron jobs (i.e. tasks scheduled to run periodically at specific intervals):

1. Every 4 hours, check for total user increase in the Main database. If the increase is greater than 10%, then re-calculate the user similarities and update the similar user cache.
2. Every 4 hours, check for total message increase in the Main database. If the increase is greater than 10%, then re-calculate the topic similarities and update the similar topic cache.
3. Every 24 hours, re-sample and encode messages from all topics in the Main database and update the encodings cache.

In this setting, the topic similarity is calculated based on the message encodings cache. If at the time of calculation, the message encodings cache does not exist for a certain topic, message sampling and encoding is triggered, and the encodings cache is updated. Upon user request, the recommender looks for similar topics and similar users from its first level cache, and the response speed is guaranteed to be under 1 second (see Section 4.2 for the time analysis).

Python is the most suited tool for building the topic recommendation system because of its rich machine learning libraries. We create the recommendation system as a stand-alone python module, containerized to run on cloud and with an API endpoint exposed to communicate with our Java backend.

3.2.6 Database Accessor and Cache

To avoid different services accessing the MongoDB database individually and causing heavy network traffic, A singleton database accessor is designed to regulate backend services' access to database. While cross collection logic of a request is left for upper layer caller to handle, the database accessor exposes all util functions for each of the database collections, including:

- Accessing all fields based on item ID.
- Accessing a specific field of the item based on item ID.
- Querying for the ID based on a specific field. The field must be unique for each item.
- Modifying the item linked to the item ID.

- Modifying a certain field of an item based on item ID.
- Modifying a field which is a list. Including methods to add, remove, update the list.
- Other functions created from an on-demand basis, such as geomesh search query. Usually, these queries are multi-targeted or matching across different fields, and there's no better way to generalize them.

Besides serving as a simple query helper, database accessor also includes a least recently used (LRU) cache to facilitate repeated access of the same data. The LRU cache stores a copy of recently used document locally and intercepts upper layer query in case a local copy exists. This action can greatly reduce the number of times our backend actually calls the database, saving budget as request intensity drops, and improve the overall performance of backend API calls as less network delay is needed. This is one of many efforts from backend to satisfy FS13.

Every time a new query arrives, database accessor attempts to solve it locally with LRU cache before going to database. If there is a cache miss, or the query must go to database (multi-targeted), LRU cache saves a copy of the items. When a write happens, it writes through, updating both the cache and the database.

In case multiple backend exist, there is a possibility of a write-after-read (WAR) hazard causing user not receiving the most up-to-date data. However, this situation is considered unlikely and harmless for the following reason:

- All writes and reads on collections other than message are highly related: the action is requested by the user, and reader of this entry is mostly that same user. This causes writes and reads happening in a single threaded manner that don't cause concurrent problems.
- Algorithm design ensured querying of messages directly from database: the messages fetched based on geomesh is a multi-target query which guarantees fetching from database, where consistency is guaranteed by external module.
- Writes on published messages are limited and non-crucial: One could still cause the WAR by logging into same account from different devices. In this case, the user knows what was done and at least one device the user "possesses", as inferred from the ability to login, has the correct data.

3.3 Deployment Strategy

Under the assumption that our backend service must handle a vast amount of external HTTP(S) requests from users world-wide and respond with small latency (FS13 and NFS1), the backend deployment must be highly scalable and the requests to our backend must be load-balanced.

Table 3-8 Decision matrix for backend deployment strategy

	On-prem	**Google Cloud – Virtual machine**	**Google Cloud - Kubernetes**
Ease to implement	1	2	3
Stableness	3	2	1
Accessibility and fault tolerance	1	2	3
Scalability	1	2	3
Cost (cheaper = higher score)	2	1	3

Total	8	9	13

On-prem deployment is the most stable choice compared to the cloud deployment options but is at the same time less scalable and more difficult to configure. Running virtual machines on cloud is more scalable and accessible than on-prem deployment but is more expensive and less flexible than deploying the backend service with Kubernetes on cloud. Google Cloud Kubernetes Engine (GKE), with its attractive monthly discount for new projects, is our selected option.

We design our deployment workflow as shown in **Error! Reference source not found.**13. We use Docker to containerize our server module and recommender module, and to prevent build error due to incompatible host machine architecture, we build the container images with Google Cloud Build. The images are then uploaded to the cloud Artifact Registry for the later Kubernetes access, as well as for record keeping. As the final step, GKE pulls the container images from the Artifact Registry, allocates cloud resources according to our deployment specifications, and exposes a public endpoint for external HTTP access. Continuous Integration and Delivery (CI/CD) is enabled through GitLab CI/CD with this deployment strategy.

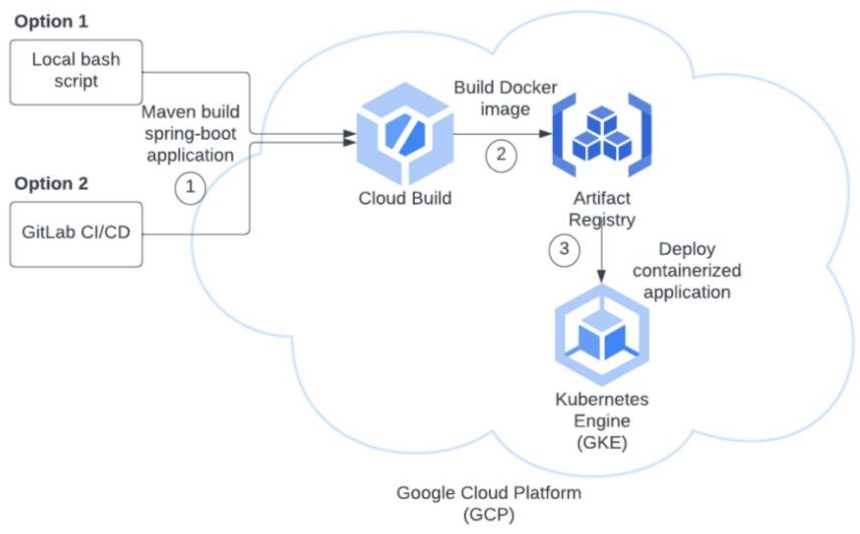


Figure 3-13 Kubernetes-based backend deployment on Google Cloud

With the Kubernetes deployment architecture, fault tolerance of our backend service is also achieved (NFS2). In our design, the server module and the recommender module are each deployed in a 'Pod' and interfaced by a 'Service'. Multiple instances of the server module and the recommender module can be deployed at once, and when one instance is faulted, its corresponding 'Service' would automatically redirect the request traffic to other running instances. By attaching a 'HorizontalPodAutoscaler' [23] Kubernetes resource to the server and recommender deployment, we also achieve automatic horizontal scaling (i.e. automatically adjust number of deployed instances according to the request amount), which contributes to NFS1 in our specifications.

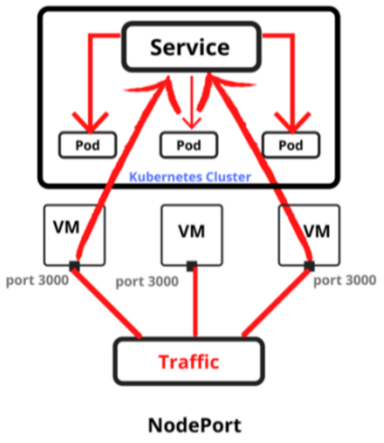


Figure 3-14 Backend Kubernetes service routing [24]

3.4 API Design

By API design we specifically mean the design of data communication between the frontend and backend. All our APIs follow the HTTP web request protocol and use JSON to serialize complex data structures in the request body. We design a few data structures, and reuse data structures for similar functionalities, while ignoring certain fields in different situations. The data structures are as follows:

Table 3-9 Structure used in API communication

Name	Fields	Description
message	Basics: user, topic, geomesh, coordinate, likes, content, timestamp Post properties: isAR, format, size, rotation, text arrangement User specific: liked	Details about the comment or post to the specific user.
usermeta	Basics: Name (firebase ID), likes, subscriptions Ownership: topics, locations, liked messages	Metadata of a user, used to display on user homepage. Lists are reduced to a count
topic	Name, region, messages under topic, owner	Details of a topic
location	Name, address, coordinate, owner	Details of a location
ARReq	Current and last coordinates, topics to fetch, filter preference, timestamp	Specific designed params to support Geomesh and Content Filtering functionality.
ARRes	List of messages, message update/removal controls, timestamp	Specific designed params to support AR content manager in its actions, ensuring consistency between frontend and backend

We group APIs together by the information these APIs touch on, and differentiate them by HTTP methods. All APIs need a user token for identification. The APIs are as follows:

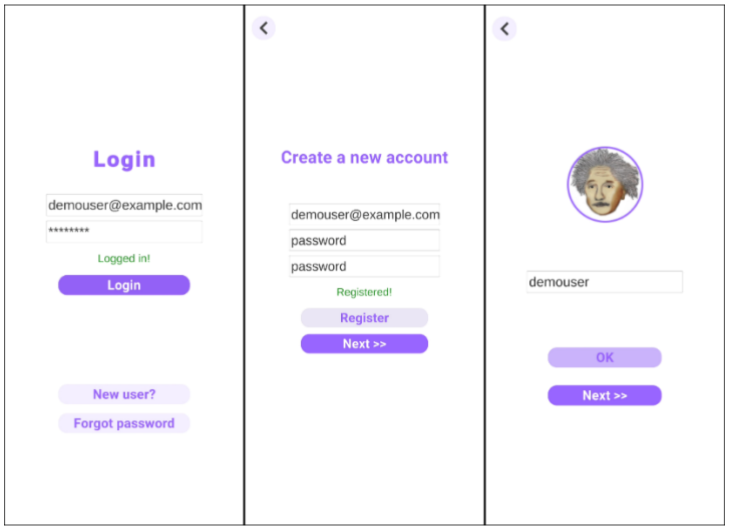
Table 3-1010 Frontend-backend APIs

URL	Method	Request	Response	Description
/nearme	POST	ARReq	ARRes	Main service API for getting nearby message on AR view
	GET	Current location Target topic	List of Message	Supplementing service API, used in map view showing all surrounding messages.
/message	POST PUT DELETE	Message (ID)	Rsp code	Add/modify/delete message the user made. Delete only need ID.
	GET	Count Timestamp Type	List of Message	Get the user's message postings of different type before a timestamp, count is used for batch fetch
/user	POST GET DELETE	User ID / none	UserMeta / Rsp code	Add/get/delete user in database, usually user is inferred from auth token, only certain GET requests pass in user ID.
/subscription	POST GET DELETE	Topic ID list / none	Topic list / Rsp code	Slightly different from naming: GET gets all subscription, POST and DELETE change some subscription
/location	POST PUT GET DELETE	Location ID Name Coordinates	Location list / response code	Add/modify/get/remove user's binded location. Only name field is allowed to change, thus requests simply pass data in params.
/healthCheck	GET	none	ok	Debugging use to check server status
/isVerified	GET	Coordinates	T/F	Whether user is in verified location
/getTopicList	GET	Coordinates	Topic list	Get all topics in user's region
/getRecommendedTopicList	GET	Coordinates	Topic list	Use recommender to recommend topics to user
/updateLikes	GET	Message ID increment	Rsp code	Like a certain message
/getFBUser	GET	User ID	UserRecord	Retrieve user info stored in Firebase

4 Prototype Data

4.1 UI Analysis

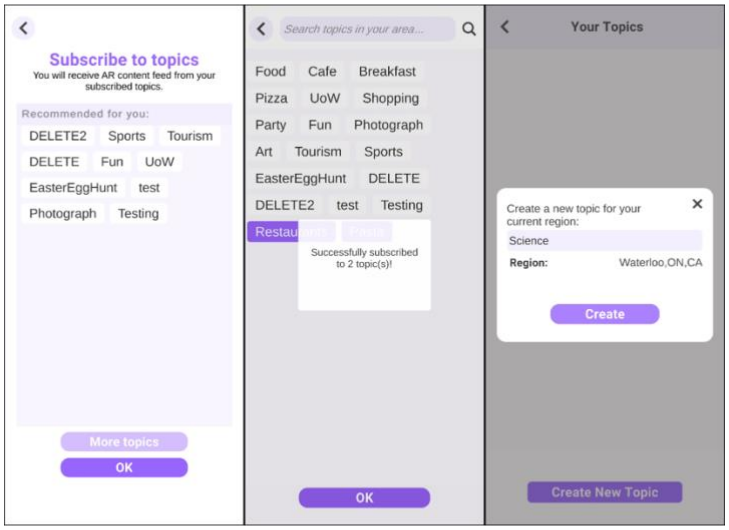
When the user first starts Near Me, they are brought to the login scene. If they already have an account, then they can login with their email and password (FS2). If they were previously logged in, then Near Me will automatically log them in (FS17). If the user doesn't have an account, then they can create one (FS1). Once they create their account, they are given the option to change their profile picture and username (FS15).



demouser@example.com
****
Logged in!
demouser@example.com
password
password
Registered!
demouser

Figure 4-1 From left to right, the login, registration, profile creation

Once the user creates their account, they are given the option to subscribe to topics. The user can choose from recommended topics (FS5) or default topics (FS3). They can also create and subscribe to custom topics (FS4). In Figure 4-2, the user subscribes to the topics “Restaurants” and “Pasta”, and they create the topic “Science”.



You will receive AR content feed from your subscribed topics.
Successfully subscribed to 2 topic(s)!
Create a new topic for your current region:
Science
Waterloo,ON,CA

Figure 4-2 From left to right, recommended topics, all topics, and topic creation

Besides subscribing to topics, the user can also publish comments and posts to topics (FS7, FS8). The left column of Figure 4-3 shows the user publishing the comment “Try our soup of the day!” to the topic “Restaurants”. The other two columns of Figure 4-3 show the user creating a text post and an image post. The user can input up to 140 characters in their text post (FS21) and can upload images up to 5 Mb in their image post (FS22). The user can adjust the style, colour, rotation, size, and position of their post before publishing it (FS16). The size slider scales the post between 0.10 m by 0.10 m to 2.5 m by 2.5 m (FS20). By tapping on the location on their screen, the user can adjust the position of their post.

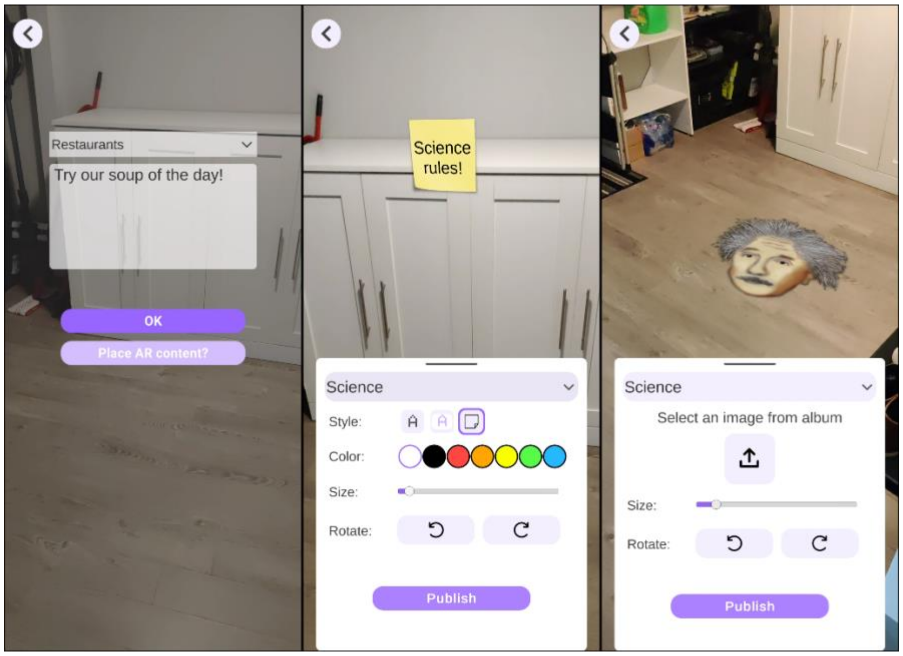


Figure 4-3 From left to right, comment creation, text post creation, and image post creation

In the main scene of Near Me, the user can see the comments and posts around them, from the topics they are subscribed to (FS9, FS10). In the left column of Figure 4-4, the user has clicked on their comment and liked it (FS14). In the right column of Figure 4-4, the user sees their text and image posts. The top left portion of the image post is occluded by the wall and ceiling (FS19).

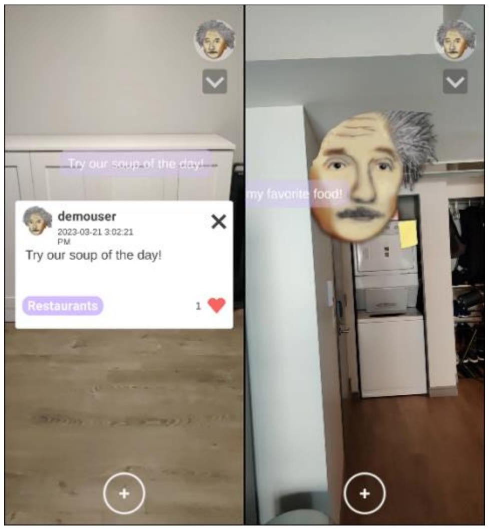
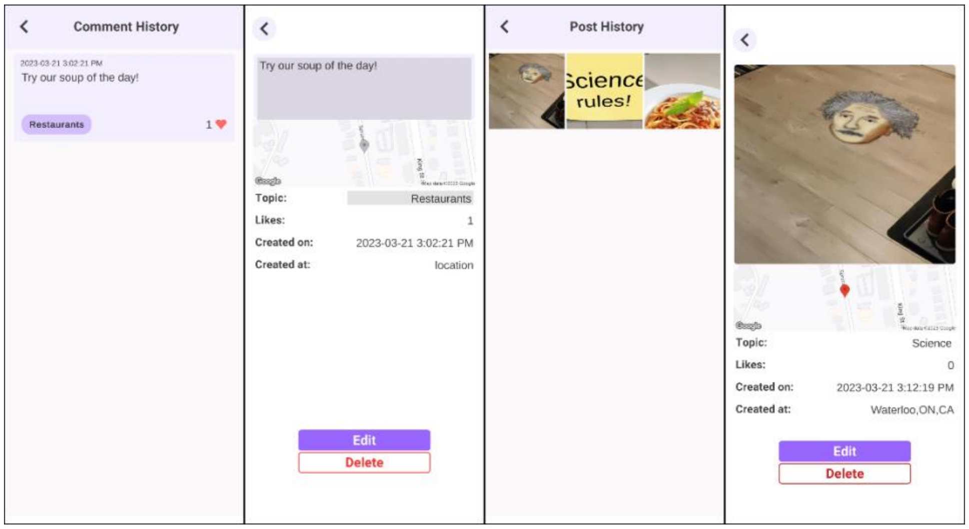


Figure 4-4 From left to right, AR comments, and AR posts

After publishing comments and posts, the user can view their publications, as is shown in the first and third columns of Figure 4-5. By clicking on a specific comment or post, the user can view information such as its location and creation date.



2023-03-21 3:02:21 PM
Try our soup of the day!
Restaurants 1

Try our soup of the day!
Topic: Restaurants
Likes: 1
Created on: 2023-03-21 3:02:21 PM
Created at: location

Topic: Science
Likes: 0
Created on: 2023-03-21 3:12:19 PM
Created at: Waterloo,ON,CA

Figure 4-5 From left to right, comment history, a comment from that history, post history, and a post from that history

4.2 Prototype HTTP Request Latency Analysis

In FS13, we specify that “user should see new AR content appear within 1s of their moving”. This is difficult to answer with different scenarios tied to how AR content is presented to user:

- User just opened the app and expects seeing content around. In this case, backend needs to prepare all data of the user’s surrounding, and frontend need to instantiate them all in Unity game engine.
- User is using the app and walking around slowly. In this case the surrounding changes slowly, backend has merely any task, and frontend only have few posts to change in the scene.
- User is using the app and moving fast. In this case, backend is in a frequent need to push info in a few geomeshs; frontend will have more rendering job.

As compared above, the situation where user just opened the app is the worst case scenario. However, a range of different factors such as user’s network situation and backend’s connection to database can still affect actual runtime. Thus, we create a script to simulate user just opening our app (pseudocode below):

```
publishedMsg = "";
thread DataProducer() :
    HTTPConn = connect2Backend();
    CenterLoc = [43.4723, -80.5449];
    while (true) :
        content = randomString(20);
        loc = randomLoc({near: CenterLoc, radius: 0.004});
        HTTPConn.publishMsg(content, loc);
        HTTPConn.readRes();
        logTime();
        publishedMsg = content;
        sleep(random({within: 3}))

thread UserSimulation() :
    HTTPConn = connect2Backend();
    CenterLoc = [43.4723, -80.5449];
    foundMsg = "";
    while (true) :
        loc = randomLoc({near: CenterLoc, radius: 0.001});
        HTTPConn.getNearMe(loc, noOldLoc);
        msgList = HTTPConn.readRes()["Message"];
        if (msgList.contains(publishedMsg) && publishedMsg != foundMsg) :
            logTime();
            foundMsg = publishedMsg;
        sleep(0.5);
```

The above code simulates a new user entering main AR scene and a source of information around that area. For every loop, the user requests for data at a point around an area. It then read the messages returned by backend and check if the last message published is among the list and profile the time of receival. The 0.5 second sleep simulates frontend behavior of sending request every 0.5 seconds This script is run concurrently with the data producer that generates these random messages. Both scripts run on the same device that is running backend and thus removes the network latency from user. The timestamps are compared to get the time for a new message to reach user:

Table 4-1 Time taken for user to receive new message from backend

Concurrent users	Min (ms)	Max (ms)	Mean (ms)
1	104.356	397.260	230.15611
2	120.133	396.101	227.32513
5	118.479	408.582	235.90420
10	115.988	431.923	277.50318

The backend is running in a controlled setting with 4 cores of 2.30GHz processor. Note that this is not the usual configuration, where computing architecture is different, and system resources can be viewed as infinite with the auto scaling. From the result, we can see a minimal effect of concurrent users overcrowding backend. And the maximum delay for data to reach frontend is below 500ms.

Beside the latency caused by backend data preparation, there are also possible delays from network situation of the user and frontend rendering of the AR object. These are highly variable across different user devices. However, we assume user network round trip latency to be below 200ms, and frontend rendering time to be way below 100ms as explained in previous section. With these delays adding up together, we are still able to achieve specification FS13.

4.3 Topic Recommender Analysis

With a Reddit dataset [22], we verify the speed and accuracy of our topic recommendation system. The chosen dataset is suitable for our test scenario because for each row in the dataset, we can parse out a message body and a tag/topic.

Table 4-2 Recommender test dataset meta data after processing.

Number of rows	**2246**
Number of topics	497
Max message per topic	172

Min message per topic	1
Average message length (char)	781.167

Table 4-3 Sample rows in the test dataset. Sample rows in the test dataset.

Topic	Message
Games	I also started playing OoS a few days ago, I had a blast. There's no inferior" Zelda titles
jobs	I think of my network" as the list of people I could phone today
programming	It depends on whether the DB is a first-class citizen" in how the application works
funny	do 5 seconds of research next time junior

To test the user-based recommender, we generate 200 user documents and from the test dataset, assign random topics to each user as the user's topic subscriptions. A test script is setup to pick a random user from the generated user set and apply the user-based recommender to generate topic recommendations. Similarity score is calculated as (number of match)/(total target user subscription). We attempt to generate 5 recommended topics based on the subscriptions of the top 10 similar users. From the test script execution logs, we obtain Table 4-3 and Table 4-4:

Table 4-4 User similarity finding by the user-based recommender

User	Similarity Score	Subscriptions
**** target_user**	-	NYKnicks, sociology, AskCulinary, infp, gentlemanboners
jtbyFH50hbe qC71	0.4	3DS, NYKnicks, AskCulinary, phoenix, Autoflowers
1j36RlnQdUn PA5d	0.2	self, TruePokemon, boston, NYKnicks, DetroitRedWings
3D5trkTmKA B0GKo	0.2	LeagueofLegendsMeta, sociology, firstworldproblems, alpinism, gundeals, WhatsInThisThing, zelda, TeamSolomid, stunfisk, OperationGrabAss, polandball
pLzKjCTxbii UstF	0.2	bourbon, LucidDreaming, fantasybaseball, mildlyinteresting, tales, iosgaming, TumblrInAction, CodAW, wendywright, NYKnicks, finance, mildlyinteresting
NXud6Y67d6 jIgbl	0.2	phoenix, MorbidReality, DetroitRedWings, bapcsalescanada, IT_CERT_STUDY, laptops, sociology, minerapocalypse, MakeupAddiction, Alabama
4Y4mcKSG8 kgwd1Z	0.2	CODZombies, MyLittleSupportGroup, videos, sociology, minerapocalypse
Lvs2JLUZ8B TMx9E	0.2	infp, ArtisanVideos, SanJoseSharks, PhilosophyofScience, GoNets, offmychest, h1z1, Seattle, lifehacks, tales, FL_Studio
XcQlTC19hr WRVh6	0.2	xxfitness, Browns, LambdaConspiracies, Screenwriting, NYKnicks, DetroitRedWings, Helldivers, masseffect, mildlyinteresting

** The target user we try to make recommendations for.

Table 4-5 User-based recommendation results

Topic	Occurrence frequency
DetroitRedWings	3
phoenix	2
mildlyinteresting	2
tales	2
minerapocalypse	2

From Table 4-4 and 4-5, we see that the user-based recommender can find similar users based on their subscriptions and select the topics with highest occurrence frequency and not yet subscribed by the target user to recommend to the target user.

To test the topic-based recommender, we read the topic and messages from the test dataset, and randomly select one topic as the target topic. We attempt to find top 10 similar topics as topic recommendation for a hypothetical user who has only subscribed to this target topic.

Table 4-6 Topic similarity finding by the topic-based recommender

Topic	Similarity Score
**** Games**	**-**
tales	2568.55
incremental_games	2271.22
Trove	2094.48
sharedota2	1965.49
nintendo	1942.48
iosgaming	1897.24
GameDeals	1745.83
firefall	1676.53
SimCity	1582.88

** The target topic we try to find similar topics for.

By inspection, we see that the topic-based recommender can find similar topics based on the meaning of the messages in a topic.

Lastly, we incorporate the multi-layer cache design described in Section 3.2.5 to test the actual latency of our recommendation system. The test is attempted with the following setup:

Table 4-7 Recommender test dataset meta data after processing.

Total messages	2246
Total topics	497
Total users	200
Target user subscription count	5
Number of similar users	20
Number of user-based recommendations	10
Number of similar topics	20

Number of topic-based recommendations	10

Table 4-8 Different recommender training and response time

	Time (train)	Time (recommend)
Topic-based recommender	0:08:53.532906	0:00:00.053471
User-based recommender	0:00:00.297700	0:00:00.182497

From Table 4-8, we see that with the multi-layer cache design, the recommender is able to produce recommendations within 1 second, greatly reduced the complexity of run-time calculation. Due to the cache design, increase in the size of the dataset or the number of user subscription should not greatly impact the recommender's performance. This contributes to NFS5 in terms of the smoothness in acquiring topic recommendations. FS5 is also satisfied based on Table 4-5 and 4-6.

5 Discussion and Conclusions

5.1 Evaluation of Final Design

As described in section 3 and 4, the final design meets all functional specifications and most of the non-functional specifications. As shown in Table 5-1 below, NFS4, the cost constraint is satisfied as well.

Table 5-1 Final project budget

	Cost (CAD)
Database	319.58
Google Cloud	101.65
Firebase	0
Others (printing)	90
Total	**511.23**

5.2 Use of Advanced Knowledge

The recommendation system design applies the knowledge from ECE457A and ECE457B with regards to the machine learning algorithms. The backend server design employs Java programming and project structuring knowledge from ECE351. The splitting of frontend and backend and the decision to utilize Kubernetes as the deployment strategy is an application of knowledge from ECE454, distributed computing. The database and data structure design uses concepts from ECE356.

5.3 Creativity, Novelty, Elegance

The AR content manager module is the focus of our creativity. The module allows real-time content anchoring and scaling with regards to proximity, and also location-based content/comment feed based on user preference. Different from basic AR object rendering, our system allows custom placement and adjustment of AR contents, syncs regional content updates for all users, and displays only contents relevant to both user and location.

Another novelty is the use of geo-mesh in our backend design. When updating the display at user's displacement, we divide the user's surroundings into geo-mesh to allow geographically distributed content rendering around the user. This design prevents the frontend from rendering crowded and repetitive contents in one area, thus improves the elegance of display.

5.4 Quality of Risk Assessment

As estimated in ECE498A, this project is purely software and does not introduce any safety hazard.

5.5 Student Workload

The workload was evenly distributed among the 4 team members, 25% each. Each of the team members have main area of focus, while usually collaborate in submodules across the system.

Table 5-2 Distribution of work among team members

Member	**Focused module**	**Workload**
Aidan Foster	Database Accessor, Content Filter, UI Map View	25%
Alice Ye	UI flow, Recommender System, Authentication, Project Management	25%
Michael Sawyer	AR Content Manager, AR-related UI	25%
Shizhen Li	Communication API, Geomesh System, Backend logic and cache	25%

6 References

1. Turner, A. (2022). *HOW MANY SMARTPHONES ARE IN THE WORLD?* Bankmycell. Retrieved May 23, 2022, from https://www.bankmycell.com/blog/how-many-phones-are-in-the-world
2. He, A. (2019, July 18). *People Continue to Rely on Maps and Navigational Apps*. eMarketer. Retrieved May 23, 2022, from https://www.emarketer.com/content/people-continue-to-rely-on-maps-and-navigational-apps-emarketer-forecasts-show
3. Ready, B. (2022, March). *2022 Retail Marketing Guide: Drive foot traffic and in-store sales*. Google. Retrieved May 23, 2022, from https://www.thinkwithgoogle.com/consumer-insights/consumer-journey/increase-foot-traffic-and-in-store-sales/
4. Alsop, T. (2021, November 29). *Number of mobile augmented reality (AR) active users worldwide from 2019 to 2024*. Statista. Retrieved May 23, 2022, from https://www.statista.com/statistics/1098630/global-mobile-augmented-reality-ar-users/
5. Anonymous. *What are some alternatives to firebase authentication?* Stackshare. Retrieved June 9, 2022, from https://stackshare.io/firebase-authentication/alternatives
6. Isichko, D. (2021, October 19). *Downsides of firebase: Limitations to be aware of*. Medium. Retrieved June 9, 2022, from https://medium.com/moqod-software-company/downsides-of-firebase-limitations-to-be-aware-of-886ade5ae5a2
7. Rocca, B. (2019, June 2). *Introduction to recommender systems*. Towards Data Science. Retrieved June 8, 2022, from https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada
8. Rodreguaze, S. (2020, December 11). *Spring vs Django: Know the difference between the two*. eduwyre. Retrieved June 20, 2022, from https://eduwyre.com/article/spring-vs-django-differences-and-similarities
9. Bakshi, N. et al. (2020, June). *Spring Framework vs Django Framework: A Comparative Study*. IRJET. Retrieved June 20, 2022, from https://www.irjet.net/archives/V7/i6/IRJET-V7I61162.pdf
10. Turgay, Ç. (2017, December 11). *Comparing API Gateway Performances: NGINX vs. ZUUL vs. Spring Cloud Gateway vs. Linkerd*. Opsgenie Engineering. Retrieved June 20, 2022, from https://engineering.opsgenie.com/comparing-api-gateway-performances-nginx-vs-zuul-vs-spring-cloud-gateway-vs-linkerd-b2cc59c65369
11. Paialunga, P. (2022, January 16). *Hands-on content based recommender system using Python*. Medium. Retrieved June 22, 2022, from https://towardsdatascience.com/hands-on-content-based-recommender-system-using-python-1d643bf314e4
12. Aditya, P. H., Budi, I., & Munajat, Q. (2016, October). A comparative analysis of memory-based and model-based collaborative filtering on the implementation of recommender system for E-commerce in Indonesia: A case study PT X. In *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)* (pp. 303-308). IEEE.
13. He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T. S. (2017, April). Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web* (pp. 173-182).
14. Rendle, S., Krichene, W., Zhang, L., & Anderson, J. (2020, September). Neural collaborative filtering vs. matrix factorization revisited. In *Fourteenth ACM conference on recommender systems* (pp. 240-248).

15. Hssina, B., Grota, A., & Erritali, M. Building Recommendation Systems Using the Algorithms KNN and SVD.
16. MongoDB. (2022). *MongoDB Pricing*. Retrieved June 15, 2022, from https://www.mongodb.com/pricing
17. Crowell, W. (2021, June 24). *PostgreSQL vs. MongoDB: Features and Benefits Comparison*. OpenLogic. Retrieved June 25, 2022, from https://www.openlogic.com/blog/postgresql-vs-mongodb
18. Sharma, A. (2021, December 3). *Redis vs MongoDB: 10 Critical Differences*. Hevo. Retrieved June 15, 2022, from https://hevodata.com/learn/redis-vs-mongodb/#rs
19. Redis. (2022). *Redis Pricing*. Retrieved June 15, 2022, from https://redis.com/redis-enterprise-cloud/pricing/
20. Geekboots. (2019, June 20). *Advantages and disadvantages of JSON over SQL*. Retrieved June 25, 2022, from https://www.geekboots.com/story/advantages-and-disadvantages-of-json-over-sql
21. SF AppWorks. (2022, January 25). *Redis vs. MongoDB: Which One Should You Choose?* Retrieved June 25, 2022, from https://www.sfappworks.com/blogs/redis-vs-mongodb-which-one-should-you-choose
22. (n.d.). *Reddit*. Hugging Face. Retrieved March 13, 2023, from https://huggingface.co/datasets/reddit
23. (2023, February 19). *Horizontal Pod Autoscaling*. Kubernetes. Retrieved March 19, 2023, from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
24. Belagatti, P. (2022, June 16). Kubernetes Services Explained. Harness. Retrieved March 19, 2023, from https://www.harness.io/blog/kubernetes-services-explained