Technical Report

1. Summary

Movie Review App is an Android application for movie discovery, review authoring, and personal watchlist tracking. The app combines:
- TMDB for public movie metadata.
- Firebase (Auth/Firestore/Storage) for user identity and app-owned social data.
- Room for local caching and reactive UI updates.

The implementation follows MVVM with repositories and Hilt-based dependency injection.

2. Introduction

- App Name: Movie Review App
- App Version: 1.0 (`versionCode=1` , `versionName="1.0"`)
- OS Target: Android (`minSdk=26` , `targetSdk=36` , `compileSdk=36`)
- Package: `com.csd3156.mobileproject.MovieReviewApp`

Short description:
Movie Review App lets users discover movies, inspect details (trailers/providers/reviews), write personal reviews with optional photos, maintain a watchlist, and manage account/profile settings.

3. System Scope And Use Cases

1. Authentication and account profile
2. Register/login/logout via Firebase Auth.
3. Persist profile metadata (name, bio, profile image URL) in Firestore.
4. Cache active account locally in Room.
5. Movie exploration
6. Browse home sections (popular/trending and extended list views).
7. Search/discover movies with sort and filters.
8. Paginate large lists.
9. Movie details and reviews
10. Open movie details, videos, providers, and aggregated review views.
11. Write review with optional image capture/upload.
12. View all reviews and review details pages.
13. Personal data surfaces
14. Profile summary (review count/watchlist count).
15. My review list.
16. Watchlist management.
17. Recommendation system
18. Content-based recommendation engine is implemented in `recommender/Recommender.kt` .
19. Model training builds movie feature vectors (text embedding + categorical vector) and persists them in a dedicated Room database.
20. Recommendation generation is wired to `RecommenderViewModel` , which observes watchlist updates and refreshes recommendations automatically.
21. Ranking uses cosine similarity over movie/user vectors with vote-aware weighting.

4. Technology Stack

- Kotlin, Coroutines, Flow/StateFlow
- Jetpack Compose + Material3
- MVVM + Repository pattern
- Hilt DI
- Retrofit + OkHttp + Moshi
- Room
- AndroidX DataStore (preferences)
- MediaPipe Tasks Text + EJML (recommendation embedding/scoring)
- Firebase Auth + Firestore + Storage
- Coil 3, ExifInterface, uCrop
- Android YouTube Player

5. Dependency Inventory (from Gradle)

Core Android

- `androidx.core:core-ktx:1.17.0`
- `androidx.appcompat:appcompat:1.7.1`
- `androidx.lifecycle:lifecycle-runtime-ktx:2.10.0`
- `androidx.activity:activity-compose:1.12.4`

UI and Navigation

- Compose BOM `androidx.compose:compose-bom:2026.02.00`
- `androidx.compose.material3:material3:1.5.0-alpha14`
- `androidx.navigation:navigation-compose:2.9.7`
- `androidx.hilt:hilt-navigation-compose:1.3.0`

Dependency Injection

- `com.google.dagger:hilt-android:2.59.1`
- `com.google.dagger:hilt-android-compiler:2.59.1` (KSP)

Networking and Serialization

- `com.squareup.retrofit2:retrofit:3.0.0`
- `com.squareup.retrofit2:converter-moshi:3.0.0`
- `com.squareup.okhttp3:logging-interceptor:5.3.2`
- `com.squareup.moshi:moshi-kotlin:1.15.2`
- `org.jetbrains.kotlinx:kotlinx-coroutines-android:1.10.2`

Local Persistence

- `androidx.room:room-runtime:2.8.4`
- `androidx.room:room-ktx:2.8.4`
- `androidx.room:room-compiler:2.8.4` (KSP)
- `androidx.room:room-paging:2.8.4`

Recommendation / ML

- `com.google.mediapipe:tasks-text:0.10.29`
- `org.ejml:ejml-simple:0.41`
- `androidx.datastore:datastore-preferences:1.1.1`

Media / Images

- `io.coil-kt.coil3:coil-compose:3.3.0`
- `io.coil-kt.coil3:coil-network-okhttp:3.3.0`
- `androidx.exifinterface:exifinterface:1.4.2`
- `com.github.yalantis:ucrop:2.2.11`
- `com.pierfrancescosoffritti.androidyoutubeplayer:core:13.0.0`

Backend

- Firebase BOM `com.google.firebase:firebase-bom:34.9.0`
- `com.google.firebase:firebase-auth`
- `com.google.firebase:firebase-firestore`
- `com.google.firebase:firebase-storage`

Security / Utility

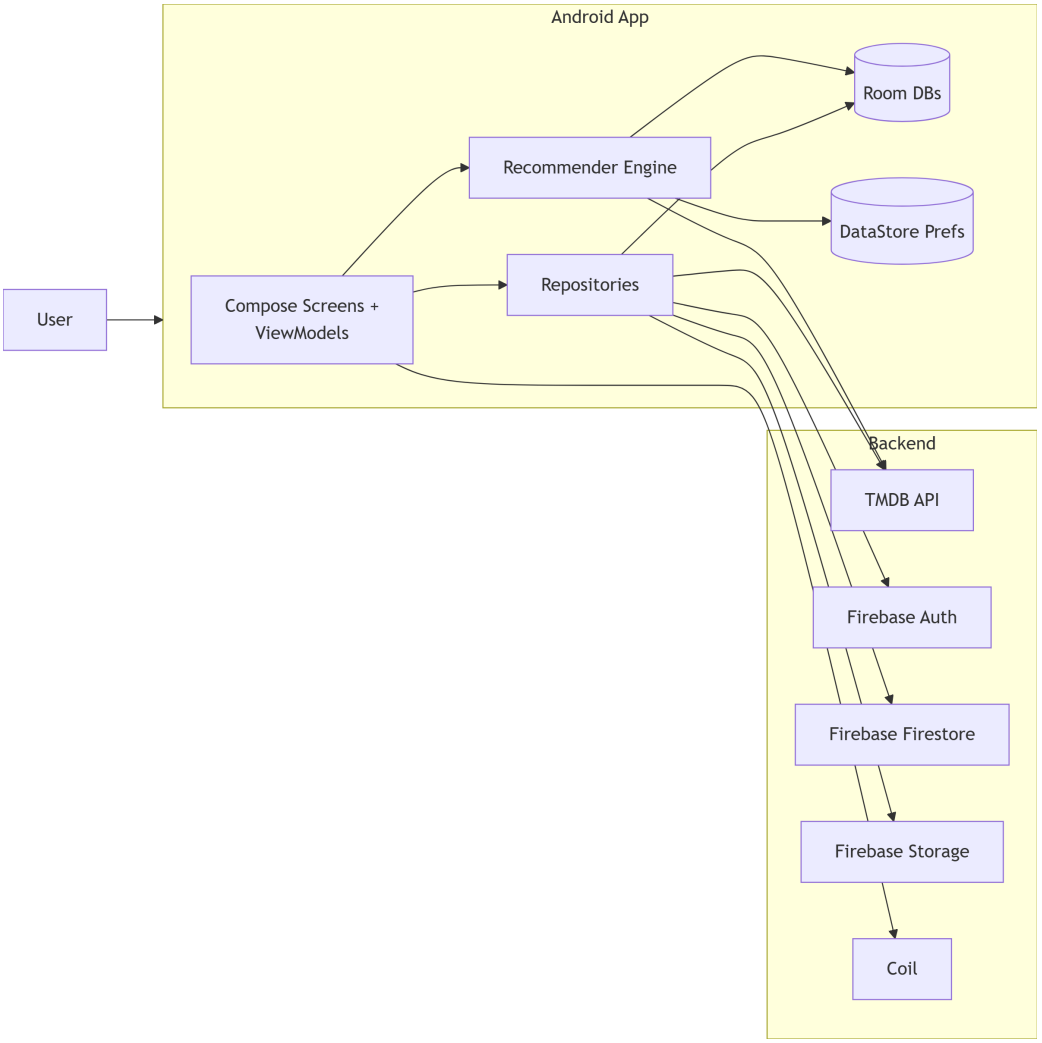
- `at.favre.lib:bcrypt:0.10.2`

6. Architecture Overview

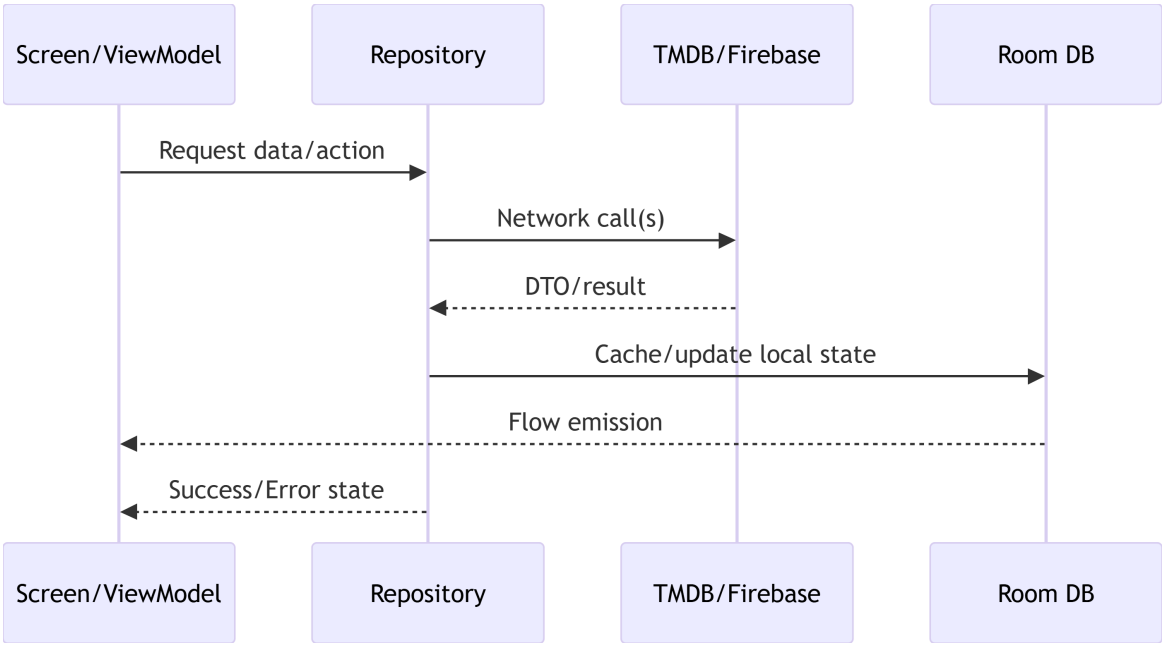
6.1 Layered Design

- UI layer: Compose screens + ViewModels.
- Domain layer: domain models and repository interfaces.
- Data layer: repository implementations coordinating local/remote data.
- Infrastructure: DI modules for network/firebase/database wiring.

6.2 Container Diagram



6.3 Runtime Data Paths



7. Project Structure Guide

Layer / Concern	Primary Path(s)	What belongs here
UI	`ui/**`	Compose screens, feature UI state, ViewModels, UI-only helpers/components
Domain	`domain/model`, `domain/repository`	Pure business/domain models and repository interfaces (platform-agnostic contracts)
Data Repositories	`data/repository/**`	Concrete repository implementations coordinating local + remote sources
Remote Integrations	`data/remote/api`, `data/remote/dto`	TMDB/Firebase service wrappers, request/response DTOs, mapping boundaries
Local Persistence	`data/local/**`, `data/local/database/**`	Room DB, entities, DAOs, converters, local-cache contracts
Recommendation Engine	`recommender/**`	Content-based model training/inference, vector persistence schema/DAO, recommendation ViewModel
Dependency Injection	`data/di/**`	Hilt modules for Retrofit/OkHttp, Firebase, Room, repositories, dispatchers
App Bootstrap	`application/**`	`@HiltAndroidApp` and app-wide initialization
App Shell / Navigation	`MainActivity.kt`	Root navigation host + top-level app shell wiring

Placement rules used in this project:
1. UI files should not directly call Firebase/TMDB SDKs; use ViewModel -> Repository.
2. Repository interfaces stay in `domain`, implementations stay in `data`.
3. DTO changes are isolated to `data/remote/dto` and mapped before reaching UI.
4. Room entities/DAOs remain in local data layer and are exposed via repository flows.

8. MVVM Coverage By Feature

8.1 Screen to ViewModel to Data Mapping

Feature Area	Primary Screens	ViewModel(s)	Repository / Services	Data Sources
App root/navigation	`MainActivity` flows	`AppViewModel`	N/A	In-memory UI state
Home sections	`HomeScreen`, `MovieExtendedList`	`HomeScreenViewModel`, `MovieExtendedListViewModel`	`MovieRepository`	TMDB
Movie details + write review	`MovieDetailScreen`	`MovieListViewModel` (current shared VM)	`MovieRepository`, `ReviewRepository`, `WatchlistRepository`	TMDB + Room + Firebase
Reviews pages	`MovieReviewsScreen`, `ReviewDetailsScreen`, `ReviewList`	`MovieReviewsViewModel`, `ReviewDetailsViewModel`, `ReviewListViewModel`	`MovieRepository`, `ReviewRepository`	TMDB + Room + Firestore
Search/Browse	`BrowseScreen`	`browseMovieViewModel`	`MovieRepository`	TMDB
Watchlist	`WatchlistScreen`	`WatchlistViewModel`	`WatchlistRepository`, `WatchlistFirestoreService`	Room + Firestore

Feature Area	Primary Screens	ViewModel(s)	Repository / Services	Data Sources
Account auth	`AccountScreen`	`AccountViewModel`	`AccountRepository`, `FirebaseAuthService`, `AccountFirestoreService`	Firebase + Room
Profile + settings	`ProfileScreen`, `AccountSettings`	`ProfileViewModel`, `AccountSettingViewModel`	`AccountRepository`, `ReviewRepository`, `WatchlistRepository`, `ProfileStorageService`	Room + Firestore + Storage
Recommendations	`HomeScreen`, `MovieExtendedList` (`MOVIE_RECOMMENDED` section)	`RecommenderViewModel`	`Recommender`, `WatchlistRepository`, `MovieRepository`	Recommender Room DB + watchlist Room cache + TMDB details

8.2 Per-Feature Responsibility Notes

1. Auth and account
2. ViewModels orchestrate login/register/update actions.
3. Repository handles Firebase auth + Firestore metadata + Room active-account cache.
4. Movie listing and discovery
5. ViewModels manage query/filter/pagination state.
6. `MovieRepositoryImpl` maps TMDB DTOs into domain models with unified error/loading semantics.
7. Movie details and review submission
8. Details screen currently reuses `MovieListViewModel` (not a dedicated detail VM yet).
9. Review submission pipeline: create Firestore doc -> optional Storage upload -> update photo URL -> cache in Room.
10. Reviews aggregation and details
11. All-reviews screens combine remote TMDB reviews with app-local reviews.
12. Review detail first checks local cache identity before remote fallback where applicable.
13. Profile and media
14. Profile photo updates support crop flow in settings and compressed upload.
15. Missing profile URL paths use fallback initial avatar rendering in UI.
16. Recommendations
17. Implemented with `RecommenderViewModel` + `Recommender`; detailed in Section 9.

8.3 Detailed Implementation By Feature

1. Authentication (`AccountViewModel` + `AccountRepository`)
2. Register flow:
   - checks username uniqueness via Firestore metadata lookup,
   - creates Firebase Auth user,
   - creates Firestore account document,
   - hydrates Room active account cache.
3. Login flow:
   - finds account metadata by username,
   - signs in via Firebase Auth,
   - writes account into Room.
4. Logout flow clears local account cache then signs out auth session.
5. Design impact: local cache enables immediate UI continuity after auth transitions.

6. Home and section pagination (`HomeScreenViewModel`, `MovieExtendedListViewModel`)
7. Maintains independent page cursors and end flags for popular/trending/discover.
8. Uses dedup by movie id when appending pages to avoid duplicates during partial refresh/reload.
9. Home refresh resets all section pagination states, then triggers first page loads.
10. Extended-list screen isolates pagination state per section for simpler UX and lower coupling.
11. Movie details + write review (`MovieDetailScreen` + `MovieListViewModel`)
12. Current architecture uses shared `MovieListViewModel` for detail data and review submission state.
13. Pulls details, videos, providers, and remote TMDB reviews.
14. Separately observes cached local reviews via Room stream.
15. Review draft/media:
    - stores `pendingCapturePath` and `reviewPhotoPath` in VM state,
    - handles camera success/failure cleanup,
    - tracks submit loading/error (`isSubmittingReview`, `reviewSubmitError`).
16. Review submit calls `ReviewRepository.addReview(...)` and updates UI status for uploading/saving.
17. Review surfaces (`MovieReviewsViewModel`, `ReviewDetailsViewModel`, `ReviewListViewModel`)
18. Movie all-reviews screen:
    - keeps separate `localReviews` and `remoteReviews`,
    - supports pull-to-refresh and page-based remote loading,
    - deduplicates incoming remote pages by review id.
19. Review details screen:
    - checks local cache first for target review id,
    - avoids remote fetch for `local-*` ids when missing,
    - otherwise fetches TMDB page and resolves review by id.
20. User review list screen:
    - displays cached user reviews from Room,
    - refresh operation resyncs user reviews from Firestore into Room.
21. Search/Browse (`browseMovieViewModel`)
22. Supports two paged data modes:
    - search mode (query present),
    - discover mode (query empty with sort/filter params).
23. Persists last discover filter set (`DiscoverParams`) so “load next” uses consistent criteria.
24. Tracks independent pagination progress and end-of-list flags for search and discover.
25. Watchlist (`WatchlistViewModel`)
26. Uses reactive stream for saved state and full watchlist list.
27. `toggle(movie, isCurrentlySaved)` delegates add/remove operations to repository.
28. Repository sync pattern combines local Room watchlist with Firestore-backed persistence.
29. Profile and settings (`ProfileViewModel`, `AccountSettingViewModel`)
30. Profile screen aggregates account info + review count + watchlist count.
31. Settings view model owns draft profile photo lifecycle:
    - pending capture path,
    - draft photo path,
    - remove-photo intent,
    - save loading/error state.
32. On save, repository handles compress/upload/update and then refreshes local account cache.

9. Recommendation/ML System

9.1 Objective and scope

- Provide personalized movie suggestions from local user behavior already available in app flow.
- Current signal source is watchlist content; recommendation results are rendered in home and extended-list recommendation surfaces.
- System is content-based and does not depend on collaborative filtering or external user-user graph data.

9.2 Model and feature engineering

- Model type: content-based ranking with cosine similarity.
- Two feature vectors are generated per movie:
  - text vector from overview using MediaPipe Text Embedder (`universal_sentence_encoder.tflite`),
  - categorical count vector built from genres + original language + adult tag + release-era tokenization.
- Categorical vector dimensions are kept stable by persisted vocabulary; vector values apply configurable repeat weighting (`MLReccConfig`).
- Final score combines weighted feature similarity and vote-aware rating adjustment.

9.3 Training pipeline

1. Startup gate
2. `MainActivity` checks `Recommender.IsTrained()` and triggers training only when needed.
3. Dataset collection
4. `EasyTrainModel()` fetches about 4000 discover-movie records from TMDB with controlled concurrency and retry handling.
5. Vectorization and persistence
6. `TrainModel(...)` rebuilds category vocabulary, marks training state false, clears existing feature rows, vectorizes movies in batches, inserts into `movie_features`, then marks state true.
7. Training state
8. Training metadata is persisted in DataStore (`is_trained`) to avoid retraining on every launch.

9.4 Inference and ranking pipeline

1. Trigger
2. `RecommenderViewModel` observes watchlist flow (`WatchlistRepository.getAllWatchlist()`), debounces, and starts recomputation.
3. User preference construction
4. Watchlist ids are resolved to full movie details via `MovieRepository.getMovieDetails(...)`.
5. User vectors are built by summing available vectors (reusing stored vectors when present, computing on-demand otherwise).
6. Candidate scoring
7. Recommender streams `movie_features` by paged DAO queries, computes batch cosine similarity for text and category matrices, combines scores by configured weights, then multiplies by weighted-rating term based on vote count.
8. Already watched ids are excluded; top-N is retained via priority queue.
9. Output
10. Returns ranked movie ids (up to configured max recommendation count).

9.5 Persistence and state management

- Room storage:
  - feature vectors and scoring metadata persist in `RecommenderDatabase` (`movie_features` table; full schema in Section 10.2).
- DataStore storage:
  - `is_trained` boolean,
  - category vocabulary set for stable category-vector dimensions.
- Runtime state:

- `RecommenderViewModel` uses recommendation-version guards to prevent stale async writes during rapid watchlist changes.

9.6 UI/ViewModel integration

- `RecommenderViewModel` publishes:
  - `recommendedMovies`,
  - loading flags (`isFetchingRecommendations`, `isLoadingMoreRecommendations`),
  - pagination completion state (`recommendationsEndReached`).
- Paging strategy:
  - id ranking is computed first, then movie cards are resolved and emitted in pages (`RECOMMENDATION_PAGE_SIZE = 12`, max 60 ids).
- UI consumers:
  - `HomeScreen` (`MOVIE_RECOMMENDED` row),
  - `MovieExtendedList` when section key is recommended.

10. Database Design (Room)

Databases:
- `MovieReviewDatabase` (version 9), configured with `fallbackToDestructiveMigration(true)`.
- `RecommenderDatabase` (version 1), dedicated to recommendation feature vectors.

10.1 `MovieReviewDatabase` Schema Tables

`accounts`

Column	Type	Nullable	Key	Notes
`id`	`Long`	No	PK (auto-generate)	Local row identifier
`uid`	`String`	No	-	Firebase user id
`email`	`String`	No	-	User login/contact email
`username`	`String`	No	-	Public/app username
`name`	`String`	Yes	-	Display name
`profileUrl`	`String`	Yes	-	Cloud profile image URL
`bio`	`String`	Yes	-	User bio text
`createdAt`	`Date`	No	-	Account creation timestamp
`updatedAt`	`Date`	No	-	Last update timestamp

`movie_reviews`

Column	Type	Nullable	Key	Notes
`id`	`Long`	No	PK (auto-generate)	Local row identifier
`movieId`	`Long`	No	-	Related TMDB movie id
`author`	`String`	No	-	Display author name
`content`	`String`	No	-	Review text
`rating`	`Double`	Yes	-	User rating value

Column	Type	Nullable	Key	Notes
`createdAtMillis`	`Long`	No	-	Review creation time (epoch ms)
`photoPath`	`String`	Yes	-	Local path or cloud URL (depending on stage/source)
`reviewId`	`String`	Yes	-	Remote review document id
`userId`	`String`	Yes	-	Review author uid
`movieTitle`	`String`	Yes	-	Cached movie title

`watchlist_movies`

Column	Type	Nullable	Key	Notes
`movieId`	`Long`	No	PK	TMDB movie id
`title`	`String`	No	-	Movie title
`posterUrl`	`String`	No	-	Poster image URL
`releaseDate`	`String`	No	-	Release date text
`rating`	`Double`	No	-	Movie rating snapshot
`savedAt`	`Long`	No	-	Time when saved to watchlist
`firstGenres`	`String`	Yes	-	Primary genre label cache

DAOs:
- `AccountDAO`
- `ReviewDao`
- `WatchlistDao`

How Room is used:
- Local-first rendering for account/review/watchlist surfaces.
- `Flow` observers keep Compose UI updated automatically after inserts/deletes/refresh.
- Repository layer controls when remote snapshots are persisted.

10.2 `RecommenderDatabase` Schema

`movie_features`

Column	Type	Nullable	Key	Notes
`id`	`Long`	No	PK	TMDB movie id
`rating`	`Double`	No	-	Movie rating snapshot used for weighting
`voteCount`	`Int`	No	-	Vote count used for weighted rating term
`popularity`	`Double`	No	-	Reserved score metadata
`overviewTagVector`	`List<Float>`	No	-	Embedded overview/text feature vector
`categoryVector`	`List<Float>`	No	-	Encoded category feature vector

DAO:
- `RecommenderDao.insertAll(...)`, `insertMovie(...)`
- `RecommenderDao.deleteAll()`
- `RecommenderDao.exists(id)`, `getMovieById(id)`

- `RecommenderDao.getMoviesPaged(limit, offset)` for batch scoring
- `RecommenderDao.getAllMovies()` (flow exposure)

How recommender persistence is used:
- Training fully regenerates `movie_features` to keep vector schema consistent.
- Recommendation scoring reads rows in pages to bound memory usage during matrix operations.
- `List<Float>` vectors are serialized via Room `TypeConverter` (`listConverter`).
- DataStore (`recommenderCategoryCount`) stores recommender metadata:
- category vocabulary set used to keep category vector dimensions stable,
- `is_trained` flag for startup training guard.

11. Remote Integrations

TMDB

- Consumed via Retrofit service (`TmdbApiService`) with auth header injected by OkHttp.
- Used for movie lists, search/discover, movie details, providers, videos, and TMDB reviews.

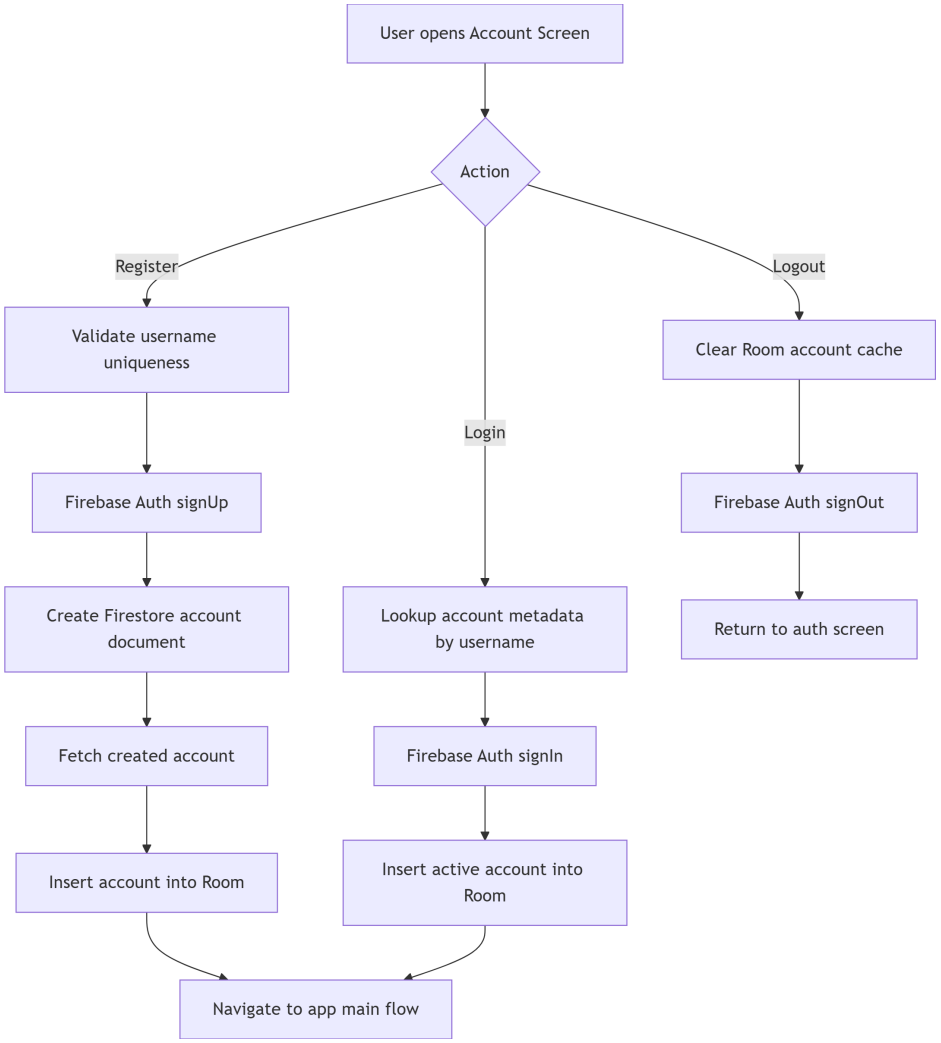
Firebase

- Auth: session and identity lifecycle.
- Firestore: account docs, review docs, watchlist docs.
- Storage: profile image binaries and review image binaries.

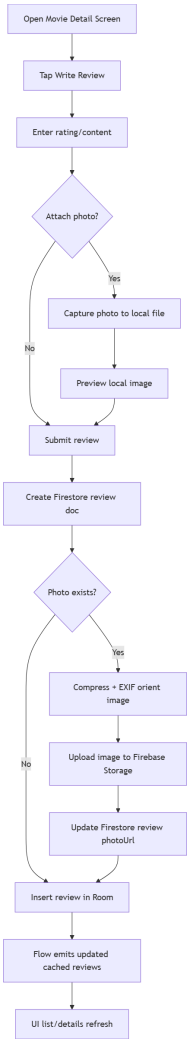
12. Key End-to-End Data Flows

1. Movie details load
2. Screen triggers details/videos/providers/reviews load.
3. Repository fetches remote TMDB data.
4. Local app reviews are observed from Room and merged at UI layer.
5. Add review with photo
6. Capture local file path -> compress/orient -> upload to Storage.
7. Update Firestore review doc with cloud URL.
8. Persist finalized review in Room for immediate local rendering.
9. Update profile photo
10. Capture + crop -> compress -> upload Storage -> update Firestore profile URL.
11. Refresh Room account cache -> Home/Profile displays updated image via Coil.
12. Recommendation generation and rendering
13. Watchlist updates trigger recommendation recomputation and paged UI updates (see Section 9).

12.1 Account Flow Chart



12.2 Movie Review Flow Chart



13. Code Snippets And Walkthrough

13.1 Auth registration and local sync

```
suspend fun registerAccount(email: String?, username: String, password: String, name: String? = null): RequestResult<String?
    val remoteAccount = findAccountByRemoteUser(username)
    if (remoteAccount != null) throw Exception("Username already exists")

    accountDao.deleteAll()
    val uid = firebaseAuthService.signUp(email ?: "${username}@moviereviewapp.default", password) {
        accountFirestoreService
    }

    val createRequestAccount = CreateAccountDto(uid = uid, email = email, username = username, name = name)
    when (val result = accountFirestoreService.createAccount(createRequestAccount)) {
        is RequestResult.Success -> {
            val accountDto = accountFirestoreService.getAccount(uid) ?: throw Exception("Unable to create account")
            accountDao.insert(accountDto.toDomain().toRoomEntity())
            return RequestResult.Success(null, uid)
        }
        is RequestResult.Error -> throw Exception(result.message)
    }
}
```

Why it matters:
- Keeps auth state and local app cache consistent after signup.

13.2 Review media pipeline

```
val reviewResult = reviewFirestoreService.createReview(movieId.toInt(), createDtoObj)
if (reviewResult is RequestResult.Success && !photoPath.isNullOrEmpty()) {
    val cloudData = localToCompressJPEG(photoPath) ?: throw Exception("Failed to compress image")
    val uploadResult = reviewCloudStoreService.uploadImage(userId, reviewResult.data, cloudData)
    if (uploadResult is RequestResult.Success) {
        reviewFirestoreService.updateReview(
            movieId.toInt(),
            reviewResult.data,
            ReviewFirebaseUpdateDto(photoUrl = uploadResult.data)
        )
    }
}
```

Why it matters:
- Decouples document creation from binary upload and minimizes payload size.

13.3 TMDB repository flow state

```
override fun getPopularMovies(page: Int): Flow<Resource<List<Movie>>> = flow {
    emit(Resource.Loading)
    val response = apiService.getPopularMovies(page = page)
    emit(Resource.Success(response.results.map { it.toDomain() }))
}.catch { throwable ->
    emit(Resource.Error(message = throwable.message ?: "Unable to load movies", throwable = throwable))
}.flowOn(ioDispatcher)
```

Why it matters:
- Standardized loading/success/error states simplify UI behavior.

13.4 Room reactive contract

```
@Query("SELECT * FROM movie_reviews WHERE movieId = :movieId ORDER BY createdAtMillis DESC")
fun observeMovieReviews(movieId: Long): Flow<List<ReviewEntity>>
```

Why it matters:
- Auto-refreshes the review UI whenever cached rows change.

14. Quality Attributes

- Performance: image compression and EXIF correction reduce upload size and display issues.
- Reliability: staged remote operations improve consistency but still rely on network quality.
- Security: token-based TMDB auth and Firebase-authenticated app operations.
- UX responsiveness: Room + Flow keeps UI fast even with intermittent remote latency.

15. References

1. Movie Review Android App (this project source and implementation).
2. Guide to app architecture: https://developer.android.com/jetpack/guide
3. Android developer documentation, recommendations for Android architecture: https://developer.android.com/topic/architecture/recommendations
4. Android developer documentation, compose UI architecture: https://developer.android.com/develop/ui/compose/architecture

5. Android developer documentation, ViewModel API reference: https://developer.android.com/reference/androidx/lifecycle/ViewModel
6. Dependency injection with Hilt: https://developer.android.com/training/dependency-injection/hilt-android
7. Android Room Architeture: https://developer.android.com/training/data-storage/room
8. Firebase User Authentication API: https://firebase.google.com/docs/auth/android/start
9. Firebase Firestore API: https://firebase.google.com/docs/firestore
10. Firebase Cloud storage API for the images: https://firebase.google.com/docs/storage
11. TMDB Developer Documentation and API: https://developer.themoviedb.org/docs
12. Coil documentation and API to load images: https://coil-kt.github.io/coil/network/
13. Coil (GitHub) project repository: https://github.com/coil-kt/coil
14. scikit-learn documentation. `cosine_similarity`: https://scikit-learn.org/1.1/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html