Mobile Exam Preparation: Code Flow & Modification Analysis

January 22, 2026

Contents

Chapter 1

All Use Case Flows

1.1 Request Authentication Token Flow

```
User → LoginScreen (lines 27-31: Composable)
    ↓
User enters email → state at line 34
    ↓
Submit button (lines 55-77)
      launches coroutine (line 56)
      calls networkService.generateToken() (lines 59-61)
    ↓
NetworkService.kt (line 8: interface)
      Token endpoint (lines 10-14)
      sends to Lambda: egsbwqh7kildllpkijk6nt4soq0wlgpe
    ↓
Response received (lines 62-65)
      Success: TokenResponse (lines 11-14: code, message)
      navigates to TokenRoute (LoginScreen line 77)
    ↓
TokenScreen displays (lines 31-35: Composable)
```

1.2 Save Email/Token Pair Flow

```
TokenScreen (lines 31-35: Composable)
    ↓
User enters token (state at line 39)
    ↓
User clicks Save (lines 64-75)
      launches coroutine (line 68)
      dataStore.edit (lines 69-74)
         Saves EMAIL key (MainActivity line 19)
         Saves TOKEN key (MainActivity line 18)
    ↓
Navigation (line 80: navigateToHome)
      navigates to HomeRoute (Routes.kt line 9)
    ↓
MenuScreen displays (main menu ready)
```

1.3 Add Flashcard Flow

```
MenuScreen → Add Card button → AddCardRoute (Routes.kt line 12)
    ↓
Navigator.kt AddCardRoute (lines 137-151: composable)
      instantiates AddScreen (line 139)
      insertFlashCard callback (lines 140-147)
    ↓
AddScreen (lines 18-22: Composable)
    ↓
User enters English/Vietnamese text (state lines 23-24)
    ↓
User clicks Add button (lines 65-80)
      try-catch at line 67
      creates FlashCard object (line 68)
      calls insertFlashCard callback (line 68)
    ↓
insertFlashCard callback (Navigator lines 140-147)
      calls flashCardDao.insertAll() (line 143)
```

```
    ↓
FlashCardDao.kt (lines 34-38)
      @Insert(onConflict = IGNORE) (lines 34-35)
      inserts into Room database (MenuDatabase.kt lines 12-30)
    ↓
Response (lines 72-74: success or duplicate error)
```

1.4 Search Flashcards Flow

```
MenuScreen → Search Cards button → FilterRoute (Routes.kt line 38)
    ↓
Navigator.kt FilterRoute (lines 166-177)
      instantiates FilterScreen (line 167)
      onSearch callback (lines 169-176)
    ↓
FilterScreen (lines 26-29: Composable)
    ↓
User selects filters (state lines 30-33)
      English text + exact/partial toggle
      Vietnamese text + exact/partial toggle
    ↓
User clicks Search (lines 89-106)
      converts boolean to 0/1 (lines 94-95)
      calls onSearch callback (lines 97-98)
    ↓
onSearch callback (Navigator lines 169-176)
      navigates to SearchCardsRoute (line 176)
      passes 4 parameters
    ↓
SearchCardsRoute composable (Navigator lines 153-181)
      extracts route parameters (line 154)
      instantiates SearchScreen (line 158)
      onEdit callback (line 160)
      onDelete callback (line 165)
    ↓
SearchScreen (lines 93-106: Composable)
      performSearch lambda (lines 112-124)
      calls flashCardDao.getFilteredFlashCards() (line 116)
    ↓
FlashCardDao.kt filter query (lines 52-82)
      CASE WHEN SQL logic
      returns List<FlashCard>
    ↓
FlashCardList renders (lines 39-90)
      for each result: thumbnail, English, Vietnamese
      Edit button (lines 73-79): calls onEdit → EditCardRoute
      Delete button (lines 80-85): calls onDelete → flashCardDao.delete()
```

1.5 Edit Flashcard Flow

```
SearchScreen → Edit button → EditCardRoute (Routes.kt line 32)
    ↓
Navigator.kt EditCardRoute (lines 183-197)
      extracts cardId (line 184)
      instantiates EditScreen (line 187)
      passes networkService, flashCardDao (line 187)
      onCardUpdated callback (line 189)
    ↓
EditScreen (lines 43-48: Composable)
    ↓
Load flashcard (LaunchedEffect lines 60-88)
      fetches card by ID (line 62): flashCardDao.getById()
      checks audio cache (lines 69-74): Utils.kt cache check (lines 24-27)
```

```
    ↓
User edits text (state lines 53-58)
    ↓
User clicks Update button (lines 147-167)
      calls flashCardDao.update() (lines 151-155)
      SQL UPDATE: id, english, vietnamese (FlashCardDao lines 40-50)
    ↓
User generates audio (lines 235-289)
      loads credentials from DataStore (lines 240-243)
      calls networkService.generateAudio() (lines 250-253)
      receives AudioResponse (DataTypes.kt lines 23-26)
      decodes Base64 (line 256)
      saves with MD5 filename (lines 257-258)
        Utils.kt MD5 (lines 9-12) + Save (lines 15-21)
    ↓
ExoPlayer plays audio (lines 197-233)
      creates ExoPlayer instance (line 204)
      plays at lines 220-221
    ↓
Success → navigation callback (line 189: onCardUpdated)
```

1.6 Study Flashcards Flow

```
MenuScreen → Study Cards → StudyCardsRoute (Routes.kt line 15)
    ↓
Navigator.kt StudyCardsRoute (lines 124-135)
      instantiates StudyScreen in STUDY_SESSION mode
      passes networkService, coroutineScope
    ↓
StudyScreen (lines 60-68: Composable)
    ↓
Load data (LaunchedEffect lines 78-114)
      STUDY_SESSION mode (lines 92-105)
      fetches all flashcards: flashCardDao.getAll() (line 97)
      creates 3-card lesson (lines 94-98)
    ↓
Display card (STUDY_SESSION UI lines 289-358)
      shows English side (lines 294-316)
      user clicks flip (lines 299-304)
      shows Vietnamese side
    ↓
User clicks Play audio (lines 332-341)
      playAudio lambda (lines 116-203)
      cache check (lines 124-125): Utils.kt lines 24-27
      if cached: immediate playback (lines 126-149)
        ExoPlayer (lines 171-190)
      if NOT cached:
        load credentials (lines 150-157): DataStore
        call networkService.generateAudio() (lines 161-164)
        decode Base64 (line 169)
        save file (line 169): Utils.kt save (lines 15-21)
        play (lines 171-190)
    ↓
User clicks Next (lines 318-330)
      advances to next card
      repeats display/play cycle
    ↓
Lesson completes → back to MenuScreen
```

1.7 Single Card View (Show Card Flow)

```
[Alternative entry] → ShowCardRoute (Routes.kt line 29: cardId param)
    ↓
```

```
Navigator.kt ShowCardRoute (lines 210-227)
      extracts cardId (line 211)
      instantiates StudyScreen in SINGLE_CARD mode
      onCardDeleted callback (line 218)
    ↓
StudyScreen (lines 60-68)
      Load data: SINGLE_CARD mode (lines 82-91)
      fetches one card by ID (line 85)
      displays UI (lines 224-287)
      Delete button (lines 269-282)
         calls flashCardDao.delete()
         triggers onCardDeleted callback
      Play button (lines 284-288)
         same playAudio flow as STUDY_SESSION
```

1.8 Audio Cache & Regeneration Flow

```
[EditScreen or StudyScreen needs audio]
    ↓
playAudio lambda (StudyScreen lines 116-203)
      Extract word from card
      Compute MD5 hash (Utils.kt lines 9-12)
      Check cache (Utils.kt lines 24-27)
    ↓
If file exists in cache:
      Load directly (lines 126-149)
      ExoPlayer plays (lines 171-190)
    ↓
If NOT cached:
      Load email, token from DataStore (lines 150-157)
      Create AudioRequest (DataTypes.kt lines 16-20)
      Call networkService.generateAudio() (lines 161-164)
      Receive AudioResponse (DataTypes.kt lines 23-26)
      Decode Base64 MP3 (line 169)
      Save file with MD5 filename (Utils.kt save lines 15-21)
      ExoPlayer plays (lines 171-190)
    ↓
[Subsequent plays of same word use cache]
```

1.9 Navigation with Back Button Flow

```
Any non-HomeRoute screen
    ↓
TopBarComponent (lines 15-41)
      shows conditional back button (lines 20-30), calls showBack() (line 23)
    ↓
showBack() lambda (Navigator line 42)
      calls navigation.popBackStack(), returns to previous route
    ↓
BottomBarComponent (lines 12-22)
      displays status message (lines 16-22)
      message updated via changeMessage callback (Navigator lines 37-39)
    ↓
Scaffold structure (Navigator lines 41-246)
      TopBar at lines 42-48, BottomBar at line 43
      NavHost at lines 50-245
```

Chapter 2

Exam Questions

USE CASE 1: Request Authentication Token

2.1 Q1: When user requests a token, the system should store the request timestamp in a database table for audit purposes.

Answer:

- **Modify Class 1:** `AuthLog.kt` (new entity file)
- **Action:** Add entity definition containing id, email, timestamp, and success flag fields - `fun createAuthLogEntity(): Unit = null`
- **Modify Class 2:** `LoginScreen.kt` **lines 56–77**
- **Action:** Add at line 62 (after token response received): insert an AuthLog record with email, current timestamp, and success=true - `fun insertAuthLog(email: String, success: Boolean): Unit = null`
- **Explanation:** After token successfully received, capture timestamp immediately; pass email + timestamp to DAO insert; execute before navigating to TokenScreen.
- **Reason:** Audit trail for security compliance; timestamp captures exact request moment; database storage (like Cloud storing location) enables brute-force detection and forensic analysis.

2.2 Q2: When user fails authentication attempts, system should show a visual lock indicator on the login button after 3 failures.

Answer:

- **Modify Class 1:** `AuthLog.kt` **(already has success field from Q1)**
- **Modify Class 2:** `LoginScreen.kt` **lines 27–80**
- **Action:** Add at line 40 (state): create state to track failed attempt count - `val failedAttemptCount = remember mutableStateOf(0)`
- **Action:** Add at line 50 (in coroutine error handler): increment failed count when token request fails - `fun incrementFailedAtte Unit = null`
- **Action:** Add at line 72 (submit button): conditionally disable button and show lock icon when count reaches 3 - `@Composable fun LockIndicatorButton(failedCount: Int, isLocked: Boolean, onClick: () -> Unit): Unit = null`
- **Explanation:** Track failed attempts in mutable state; increment on each failed token request; disable button and display lock icon when count >= 3; show informational message.
- **Reason:** Display requirement (like Cloud showing metadata); provides visual feedback on account lockout; prevents brute-force attacks; similar to Cloud security patterns (display current state).

2.3 Q3: When user attempts token request, the system should lock the account after 5 failed attempts within 1 hour (similar to Cloud's delete authorization).

Answer:

- **Modify Class 1:** `AuthLog.kt` **(already has success field from Q1)**
- **Modify Class 2:** `LoginScreen.kt` **lines 50–68**
- **Action:** Add at line 51 (before Lambda call): query failed attempts for this email within the last hour - `fun getFailedAttempts String): Flow<Int> = null`

- **Action:** Add at line 52 (guard): if failed attempts are five or more, set error state and stop - `fun checkAccountLock(attempts: Int): Boolean = attempts >= 5`
- **Action:** Add at line 61 (on error): insert an AuthLog record with email, current timestamp, and success=false - `fun logFailedAttempt(email: String): Unit = null`
- **Explanation:** Before Lambda invocation, query failed attempts in last 1 hour; if >= 5, prevent call and show lock message; store all failed attempts with success=false.
- **Reason:** Authorization enforcement (like Cloud delete-only-by-owner); brute-force attack prevention; blocks further attempts; implements access control at app layer.

USE CASE 2: Save Email/Token Pair

2.4 Q4: When saving email/token to DataStore, the system should encrypt the token and store an encryption key.

Answer:

- **Modify Class 1:** `MainActivity.kt` **lines 17–22**
- **Action:** Add at line 20: define preference keys for encryption salt and encrypted token - `companion object  const val TOKEN`$_S ALT_K EY =$ "$token_s alt$"
- **Modify Class 2:** `TokenScreen.kt` **lines 64--75**
- **Action:** Add at line 68 (before dataStore.edit): generate a random salt value - `fun generateSalt(): ByteA = SecureRandom().generateSeed(16)`
- **Action:** Add at line 69: encrypt the token using the generated salt - `fun encryptToken(token: String, salt: ByteArray): String = null`
- **Action:** Replace line 72 (token save): persist both salt and encrypted token into DataStore - `suspend fun saveSaltAndToken(salt: ByteArray, encrypted: String): Unit = null`
- **Explanation:** Generate 16-byte random salt; encrypt token using AES cipher with salt; save both encrypted token and salt to DataStore for later decryption.
- **Reason:** Encryption prevents plaintext token compromise; salt prevents rainbow table attacks; similar to Cloud storing location; adds security layer beyond Android DataStore's built-in encryption.

2.5 Q5: The TokenScreen should display when the token expires (validity period).

Answer:

- **Modify Class 1:** `MainActivity.kt` **lines 17–22**
- **Action:** Add at line 21: define preference key for token creation timestamp - `const val TOKEN`$_C REATED_A T_K EY =$ "$token_c reated_a t$"
- **Modify Class 2:** `TokenScreen.kt` **lines 31--80**
- **Action:** Add at line 73 (dataStore.edit): store current time as token creation timestamp - `suspend fun storeTokenTimestamp(timestamp: Long): Unit = null`
- **Action:** Add at line 45 (display section): calculate expiration as creation time plus validity window and show the formatted value in UI - `fun calculateExpiration(createdAt: Long): String = SimpleDateFormat.form + 24*3600*1000)`
- **Explanation:** Store creation timestamp when token saved; calculate expiration by adding 24 hours; format timestamp and display in UI.
- **Reason:** Transparency shows token validity period (like Cloud showing photo metadata); helps user understa when re-authentication needed; prevents confusion about expiration.

2.6 Q6: When saving email/token pair, system should prevent save if token comes from a different email.

Answer:

- **Modify Class:** `TokenScreen.kt` **lines 68–75**
- **Action:** Add at line 69 (before dataStore.edit): decode token payload to read the embedded email - `fun decodeJWTEmail(token String): String? = null`
- **Action:** Add at line 70 (guard check): if token email differs from route email, show error and abort save - `fun validateTokenEmail(tokenEmail: String, routeEmail: String): Boolean = tokenEmail == routeEmail`
- **Action:** Line 72 (dataStore.edit) proceeds only when emails match
- **Explanation:** Decode JWT payload (base64-decode middle part); extract email claim from JSON; compare with email parameter from route; if mismatch, abort save.
- **Reason:** Authorization enforcement (like Cloud delete-only-by-owner); prevents token spoofing/swapping; ensures token authenticity; protects against session hijacking attacks.

USE CASE 3: Add Flashcard

2.7 Q7: When adding flashcard, system should store creation metadata (timestamp and creator email).

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add at line 17 (entity properties): include createdAt timestamp and createdBy email fields - `val createdAt: Long = System.currentTimeMillis()`
- **Modify Class 2:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 62 (inside try block): retrieve current user email and current timestamp - `suspend fun getCurrentUserEma String = dataStore.data.map it[EMAIL`$_{KEY}$`].`$first()$
- `Action: Replace line 63 (FlashCard creation): construct FlashCard with those metadata values included - fun buildFlashCard(english: String, vietnamese: String, email: String): FlashCard = null`
- `Explanation: Capture timestamp and email from DataStore before creating FlashCard; pass both as construct parameters; existing callback passes complete object to DAO insert.`
- `Reason: Metadata storage (like Cloud storing location); enables ownership tracking; similar to Cloud requ prepares for access control (only creator can modify).`

2.8 Q8: When listing flashcards after search, display should show creation date and creator email.

Answer:

- **Modify Class:** `SearchScreen.kt` **lines 38–90**
- **Action:** Add at line 65 (in FlashCardList row): display formatted createdAt date - `@Composable fun CreatedDateDisplay(ti Long): Unit = null`
- **Action:** Add at line 67 (in row): display creator email label - `@Composable fun CreatorEmailDisplay(email: String): Unit = null`
- **Explanation:** Access loaded FlashCard entity fields (createdAt, createdBy); format timestamp; display both in row below card text.
- **Reason:** Display requirement (like Cloud showing photo email and description); transparency shows card provenance; helps identify own cards; similar to Cloud metadata display.

2.9 Q9: When adding flashcard, system should prevent user from adding more than 1000 cards per day (quota enforcement similar to Cloud delete authorization).

Answer:

- **Modify Class:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 60 (before insert call): query today's insert count for this user - `fun getTodayCardCount(email: String): Flow<Int> = null`
- **Action:** Add at line 61 (guard check): if count reaches 1000, show error and stop - `fun checkQuotaExceeded(count: Int): Boolean = count >= 1000`
- **Action:** Line 63 (insert call) runs only when under quota
- **Explanation:** Query DAO for count of cards created by current user today; if >= limit, abort insertion and display error via BottomBarComponent.
- **Reason:** Authorization/quota enforcement (like Cloud delete-only-by-owner pattern); prevents resource exhaustion; protects database from abuse; implements access control.

2.10 Q10: When adding a flashcard, the user should also specify a category (e.g., "Vocabulary", "Grammar", "Idioms").

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add at line 17: include category field (String type) in the entity - `val category: String = ""`
- **Modify Class 2:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 28 (state): create a mutable state for category selection - `val selectedCategory = remember mutableStateOf("")`
- **Action:** Add at lines 40–45 (UI): add a dropdown or text input to let user select/enter category - `@Composable fun CategoryDropdown(selected: String, onSelected: (String) -> Unit): Unit = null`
- **Modify Class 3:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 63 (FlashCard creation): include category value from state - `fun buildFlashCardWithCategory(english: String, vietnamese: String, category: String): FlashCard = null`
- **Explanation:** Add category field to entity; create state for category input; display UI control for user to select category; pass category to FlashCard constructor when inserting.
- **Reason:** Data storage requirement (like Cloud adding location field); enables organization of flashcards; prepares for category-based filtering and display.

2.11 Q11: When adding a flashcard, the user should also specify a difficulty level (e.g., "Easy", "Medium", "Hard") to support progressive learning.

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add difficulty field to FlashCard entity as an enum or string (EASY, MEDIUM, HARD) - `val difficulty: String = "MEDIUM"`
- **Modify Class 2:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 32 (state): create state to hold selected difficulty level - `val selectedDifficulty = remember mutableStateOf("MEDIUM")`
- **Action:** Add at lines 45–55 (UI): render difficulty selection dropdown or buttons (Easy, Medium, Hard) - `@Composable fun DifficultySelector(selected: String, onSelect: (String) -> Unit): Unit = null`
- **Action:** Add at line 70 (before insertion): pass selected difficulty to FlashCard creation - `fun buildFlashCardWithDifficulty String, vietnamese: String, category: String, difficulty: String): FlashCard = null`

- **Explanation:** Add difficulty enum to entity; store selection in state; render 3 difficulty buttons in UI; pass difficulty to DAO insert with other fields.
- **Reason:** Data storage requirement; enables adaptive learning and progress tracking; prepares for difficulty-based filtering in study sessions; similar to Cloud storing metadata.

2.12 Q12: When adding a flashcard with difficulty level, system should support fine-grained difficulty (Easy/Difficult binary system) to enable streamlined learning paths.

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add a new binary difficulty field to FlashCard entity (isAdvanced: Boolean, defaulting to false for Easy) - `val isAdvanced: Boolean = false`
- **Modify Class 2:** `AddScreen.kt` **lines 17–88**
- **Action:** Add at line 33 (state): create state to hold binary difficulty selection (Easy or Difficult toggle) - `val selectedBinaryDi = remember  mutableStateOf(false) // false = Easy, true = Difficult`
- **Action:** Add at lines 56–65 (UI): render simple toggle button or radio buttons (Easy / Difficult) for quick selection - `@Composable fun BinaryDifficultyToggle(isAdvanced: Boolean, onToggle: (Boolean) -> Unit): Unit = null`
- **Action:** Add at line 71 (before insertion): pass binary difficulty flag to FlashCard creation - `fun buildFlashCardWithBinaryDi String, vietnamese: String, category: String, isAdvanced: Boolean): FlashCard = null`
- **Modify Class 3:** `FlashCardDao.kt`
- **Action:** Add query to fetch all Easy cards for beginner learners - `@Query("SELECT * FROM FlashCard WHERE isAdvanced = false ORDER BY RANDOM()")`
- **Explanation:** Add boolean isAdvanced field to entity; render toggle UI for Easy/Difficult selection; store binary value; enables faster filtering for beginner vs advanced paths.
- **Reason:** Simplified difficulty tracking requirement; streamlines learning path selection; faster query performance than 3-tier system; supports beginner-focused learning progression.

USE CASE 4: Search Flashcards

2.13 Q13: When user searches flashcards, system should store search queries in a SearchHistory table.

Answer:

- **Create:** New entity `SearchHistory.kt`: define fields for id, English query, Vietnamese query, exact flags, and timestamp
- **Modify Class:** `FilterScreen.kt` **lines 26–105**
- **Action:** Add at line 32 (after search button clicked): build a SearchHistory record using current filter inputs - `fun buildSearchHistory(english: String, vietnamese: String, exactFlags: Pair<Boolean, Boolean>): SearchHisto = null`
- **Action:** Add at line 33: insert the SearchHistory record into the database - `suspend fun insertSearchHistory(history: SearchHistory): Unit = null`
- **Explanation:** Before navigating to search results, construct SearchHistory record with all filter parameters; insert into database.
- **Reason:** Metadata storage (like Cloud storing location); enables history/analytics features; prepares for autocomplete and search suggestions.

2.14 Q14: The search UI should suggest previous search queries (autocomplete from SearchHistory).

Answer:

- **Modify Class:** `FilterScreen.kt` **lines 25–128**
- **Action:** Add at line 32 (state setup): create state holder for recent searches list - `val recentSearches = remember mutableStateOf<List<SearchHistory>>(emptyList())`
- **Action:** Add at line 33 (LaunchedEffect): query top recent searches and store in state - `fun getRecentSearches(limit: Int = 10): Flow<List<SearchHistory>> = null`
- **Action:** Add at line 50 (UI, after TextFields): show recent searches as selectable suggestions - `@Composable fun SearchSuggestions(searches: List<SearchHistory>): Unit = null`
- **Action:** Add at line 52 (click handler): when suggestion chosen, fill English/Vietnamese fields from it - `fun applySuggestion(h SearchHistory, onApply: (String, String) -> Unit): Unit = null`
- **Explanation:** LaunchedEffect fetches top 10 recent searches; display as clickable suggestions; clicking populates filter fields for quick re-search.
- **Reason:** Display requirement (like Cloud showing photo list); improves UX for power users; reduces typing for repeated searches.

2.15 Q15: When searching, system should prevent user from exceeding 100 searches per day (authorization enforcement similar to Cloud delete-only-by-owner).

Answer:

- **Modify Class:** `FilterScreen.kt` **lines 25–128**
- **Action:** Add at line 91 (inside search button handler): query today's search count - `fun getTodaySearchCount(email: String): Flow<Int> = null`
- **Action:** Add at line 92 (guard check): if count reaches 100, show error and stop - `fun checkSearchQuotaExceeded(count: Int): Boolean = count >= 100`
- **Action:** Add at line 95 (if allowed): insert the search history entry - `suspend fun recordSearch(history: SearchHistory): Unit = null`
- **Action:** Add at line 96 (after insert): proceed with navigation to results - `fun navigateToResults(filters: SearchFilters) Unit = null`
- **Explanation:** Query SearchHistory count for today; if count ≥ 100, show error and return early; otherwise insert history and navigate.
- **Reason:** Authorization/quota requirement (like Cloud delete-only-by-owner); prevents daily search abuse; protects database from excessive queries.

2.16 Q16: When displaying search results, the app should show the category of each flashcard.

Answer:

- **Modify Class:** `SearchScreen.kt` **lines 38–90**
- **Action:** Add at line 70 (in FlashCardList row): display category label next to or below the English/Vietnamese text - `@Composable fun CategoryBadge(category: String): Unit = null`
- **Explanation:** Access category field from loaded FlashCard entity; display it as a badge or text label in each result row.
- **Reason:** Display requirement (like Cloud showing location); helps users quickly identify flashcard type; improves result readability.

2.17 Q17: When displaying search results, the app should allow filtering results by difficulty level to help users find practice materials at their current skill level.

Answer:

- **Modify Class 1:** `SearchHistory.kt`
- **Action:** Add difficulty field to SearchHistory to track difficulty filters used in searches - `val difficultyFilter: String? = null`
- **Modify Class 2:** `FilterScreen.kt` **lines 25–128**
- **Action:** Add at line 35 (state): create state to hold selected difficulty filter - `val selectedDifficultyFilter = remember mutableStateOf<String?>(null)`
- **Action:** Add at lines 50–60 (UI): render difficulty filter buttons (All / Easy / Medium / Hard) - `@Composable fun DifficultyFilterButtons(selected: String?, onSelect: (String?) -> Unit): Unit = null`
- **Action:** Add at line 100 (search execution): pass difficulty filter to DAO query - `fun performSearchWithDifficulty(english: String, vietnamese: String, difficulty: String?): Flow<List<FlashCard>> = null`
- **Explanation:** Add difficulty field to SearchHistory for tracking; render 4 filter buttons in UI; include difficulty in DAO query WHERE clause; return only matching difficulty cards or all if "All" selected.
- **Reason:** Display/filtering requirement; helps progressive learners focus on appropriate difficulty level; tracks user's preferred difficulty in search history; improves UX.

2.18 Q18: When searching for flashcards, system should support fine-grained filtering by easy/difficult level to enable beginner learners to find introductory content.

Answer:

- **Modify Class 1:** `SearchHistory.kt`
- **Action:** Add binary difficulty field to SearchHistory to track easy/difficult preference in searches - `val isAdvancedFilter: Boolean? = null`
- **Modify Class 2:** `FilterScreen.kt` **lines 25–128**
- **Action:** Add at line 36 (state): create state to hold binary difficulty filter (null, Easy, or Difficult) - `val selectedBinaryDifficulty = remember mutableStateOf<Boolean?>(null)`
- **Action:** Add at lines 61–70 (UI): render simple binary filter buttons (All / Easy / Difficult) for quick filtering - `@Composable fun BinaryDifficultyFilterButtons(selected: Boolean?, onSelect: (Boolean?) -> Unit): Unit = null`
- **Action:** Add at line 101 (search execution): pass binary difficulty filter to DAO query - `fun performSearchWithBinaryDifficulty(english: String, vietnamese: String, isAdvanced: Boolean?): Flow<List<FlashCard>> = null`
- **Modify Class 3:** `FlashCardDao.kt`
- **Action:** Add query method for easy-difficulty-only results - `@Query("SELECT * FROM FlashCard WHERE isAdvanced = false AND english LIKE :query ORDER BY RANDOM()")`
- **Explanation:** Add binary difficulty field to SearchHistory; render 3 filter buttons in UI (All / Easy / Difficult); include isAdvanced filter in DAO query WHERE clause; optimize for beginner paths.
- **Reason:** Simplified filtering requirement; streamlines UX for beginner learners; faster queries; tracks learning path preference; supports progressive difficulty progression.

USE CASE 5: Edit Flashcard

2.19 Q19: When user updates a flashcard, system should store update metadata (timestamp and who modified it).

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add at line 18: include nullable fields for updatedAt timestamp and updatedBy email in the entity - `val updatedAt: Long? = null, val updatedBy: String? = null`
- **Modify Class 2:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 160 (update button handler): capture current timestamp and current user email - `suspend fun captureUpdateMetadata(): Pair<Long, String> = null`
- **Action:** Add at line 165 (before DAO call): build an updated card copy that carries new text plus updated metadata - `fun buildUpdatedCard(card: FlashCard, newEnglish: String, newVietnamese: String, email: String): FlashCard = null`
- **Modify Class 3:** `FlashCardDao.kt` **lines 48–49**
- **Action:** Add in UPDATE statement at line 48: set updatedAt and updatedBy fields along with text update - `fun updateFlashCard(card: FlashCard): Unit = null`
- **Explanation:** On update click, gather timestamp and user email, create updated card with metadata, and persist via DAO in a single operation.
- **Reason:** Data storage requirement (like Cloud storing creation metadata); enables audit trail; tracks last editor and modification time.

2.20 Q20: The EditScreen should display card's creation and edit history (dates and who created/modified).

Answer:

- **Modify Class:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 125 (display section): show createdBy and createdAt as a formatted label - `@Composable fun CreationMetadataDisplay(email: String, timestamp: Long): Unit = null`
- **Action:** Add at line 130: show updatedBy (or N/A) and updatedAt as a formatted label - `@Composable fun UpdateMetadataD String?, timestamp: Long?): Unit = null`
- **Explanation:** After loading card, display metadata fields formatted as readable timestamps; createdBy and updatedBy show email; dates formatted using SimpleDateFormat.
- **Reason:** Display requirement (like Cloud showing metadata); transparency shows card lifecycle; helps user understand edit history; similar to Cloud requirement (show all metadata).

2.21 Q21: When editing a flashcard, system should prevent modification if current user is NOT the card creator (authorization enforcement similar to Cloud delete-only-by-owner).

Answer:

- **Modify Class:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 66 (LaunchedEffect): read current user email from DataStore - `suspend fun getCurrentUserEmail(): String? = null`
- **Action:** Add at line 75 (after fetching card): compare card.createdBy to current user; if different, mark editing as disallowed - `fun isCardOwner(card: FlashCard, currentEmail: String): Boolean = card.createdBy == currentEmail`
- **Action:** Add at line 160 (update handler guard): if editing disallowed, show error and return - `fun guardUpdateAuthorization Boolean): Unit = if (!isAllowed) throw SecurityException()`
- **Explanation:** Verify creator match in LaunchedEffect; store in state variable; prevent update button execution if false; disable TextFields conditionally.
- **Reason:** Authorization enforcement (like Cloud delete-only-by-owner); protects user's study materials; prevents unauthorized edits; similar to Cloud access control pattern.

2.22 Q22: When editing a flashcard, the user should be able to change its category.

Answer:

- **Modify Class 1:** `FlashCard.kt` **lines 10–20**
- **Action:** Add at line 17: include category field (String type) in the entity - `val category: String = ""`
- **Modify Class 2:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 57 (state setup): create mutable state for category that loads from the current card - `val editableCategory = remember mutableStateOf("")`
- **Modify Class 3:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at lines 135–140 (UI): display category dropdown/input field - `@Composable fun EditableCategoryField(value String, onChange: (String) -> Unit): Unit = null`
- **Modify Class 4:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 165 (update handler): include new category value in the updated card object before calling DAO update - `fun applyEditedCategory(card: FlashCard, newCategory: String): FlashCard = null`
- **Explanation:** Load category into state when card is fetched; display UI control for category selection; pass updated category to DAO update call.
- **Reason:** Data modification requirement (like Cloud allowing edit); lets users recategorize cards; similar to other editable fields.

2.23 Q23: When editing a flashcard, the user should be able to change its difficulty level to adapt to improving skills.

Answer:

- **Modify Class 1:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 60 (state): create state to hold current difficulty for editing - `val editingDifficulty = remember mutableStateOf(currentCard.difficulty)`
- **Action:** Add at lines 120–130 (UI): render difficulty selector dropdown or buttons for editing - `@Composable fun EditableDifficultySelector(current: String, onSelect: (String) -> Unit): Unit = null`
- **Action:** Add at line 155 (update handler): pass new difficulty value to FlashCard update - `fun updateFlashCardDifficulty(c Int, newDifficulty: String): FlashCard = null`
- **Modify Class 2:** `FlashCardDao.kt`
- **Action:** Add update query method that modifies difficulty field - `@Query("UPDATE FlashCard SET difficulty = :difficulty WHERE id = :cardId")`
- **Explanation:** Load current difficulty from card; render editable selector (3 buttons); on save, update difficulty field in database via DAO query.
- **Reason:** Data modification requirement; lets users recategorize cards by difficulty as their skills improve; enables learning progression tracking.

2.24 Q24: When editing a flashcard, system should support updating with easy/difficult binary difficulty level to enable quick difficulty adjustments.

Answer:

- **Modify Class 1:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 61 (state): create state to hold binary difficulty value (isAdvanced: Boolean) - `val editingBinaryDifficu = remember mutableStateOf(currentCard.isAdvanced)`
- **Action:** Add at lines 131–140 (UI): render binary difficulty toggle (Easy / Difficult) for quick updates - `@Composable fun EditableBinaryDifficultyToggle(isAdvanced: Boolean, onToggle: (Boolean) -> Unit): Unit = null`
- **Action:** Add at line 156 (update handler): pass new binary difficulty to DAO update - `fun updateFlashCardBinaryDifficult Int, isAdvanced: Boolean): FlashCard = null`

- **Modify Class 2:** `FlashCardDao.kt`
- **Action:** Add update query method that modifies isAdvanced field - `@Query("UPDATE FlashCard SET isAdvanced = :isAdvanced WHERE id = :cardId")`
- **Explanation:** Load isAdvanced state when card is fetched; render 2-button toggle in UI (Easy / Difficult); on save, update binary difficulty in database via DAO query.
- **Reason:** Data modification requirement with simplified UI; streamlines difficulty updates; faster than 3-tier system; supports quick recategorization as skills progress.

USE CASE 6: Study Flashcards

2.25 Q25: When user studies a lesson, system should track study statistics (cards studied, time spent, replays).

Answer:

- **Create:** New Room entity
  texttttStudyLog.kt: define fields for id, cardId, timeSpent, replayCount, and timestamp - `@Entity data class StudyLog(val cardId: Int, val timeSpent: Long, val replayCount: Int, val timestamp: Long)`
- **Modify Class:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 95 (LaunchedEffect): generate a session identifier - `val sessionId = remember  UUID.randomUUID().to`
- **Action:** Add at line 96 (card state): track when the current card study begins - `val cardStartTime = remember mutableStateOf(System.currentTimeMillis())`
- **Action:** Add at line 110 (next card handler): compute time spent on the current card before advancing - `fun computeTimeSpent Long): Long = System.currentTimeMillis() - startTime`
- **Action:** Add at line 112: insert a StudyLog entry with card id, elapsed time, replay count, and current timestamp - `suspend fun recordStudySession(log: StudyLog): Unit = null`
- **Explanation:** Track session time per card; on card transition, calculate elapsed time and insert StudyLog record; ReplayCount incremented on each play button click (line 332).
- **Reason:** Data storage requirement (like Cloud storing metadata); enables analytics; tracks learning progress; similar to Cloud storing interaction data.

2.26 Q26: The StudyScreen should display study statistics (cards studied today, total time spent, progress bar).

Answer:

- **Modify Class:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 80 (state): hold today's study stats (card count and total time) - `val studyStats = remember mutableStateOf<Pair<Int, Long>>(0L to 0L)`
- **Action:** Add at line 85 (LaunchedEffect): fetch aggregated stats for today and store them - `fun getTodayStudyStats(email: String): Flow<Pair<Int, Long>> = null`
- **Action:** Add at line 90: update state with the retrieved count and total time - `fun updateStatsState(stats: Pair<Int, Long>, onUpdate: (Pair<Int, Long>) -> Unit): Unit = null`
- **Action:** Add at lines 260–270 (display section): show card count and formatted total time - `@Composable fun StudyStatsDisp Int, totalTime: Long): Unit = null`
- **Action:** Add at line 275: render progress bar for studied versus total cards - `@Composable fun StudyProgressBar(studied: Int, total: Int): Unit = null`
- **Explanation:** Query StudyLog with DATE filter; sum timeSpent and count records; display as formatted stats above $\text{STUDY}_SESSIONmodeUI; progressbarshowsstudied/totalratio$.
- **Reason:** Display requirement (like Cloud showing photo count); motivates user with progress visualization; similar to Cloud requirement (display aggregated data).

2.27 Q27: When studying, system should only show flashcards created by current user (authorization enforcement similar to Cloud delete-only-by-owner).

Answer:

- **Modify Class 1:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 100 (STUDY$_SESSIONmode, LaunchedEffect)$: $retrievecurrentuseremailfromDataStore$ – `suspend fun fetchCurrentUserEmail(): String? = null`
- **Action:** Add at line 105: fetch cards filtered by createdBy equal to that email - `fun getCardsForUser(email: String): Flow<List<FlashCard>> = null`
- **Modify Class 2:** `FlashCardDao.kt` **lines 15–66**
- **Action:** Add at line 45 (new query method): define DAO query returning only cards whose createdBy matches the supplied email, randomized and limited - `@Query("SELECT * FROM FlashCard WHERE createdBy = :email ORDER BY RANDOM() LIMIT 3")`
- **Explanation:** Query DAO with current user email parameter; filter by createdBy column; return only own cards; randomize order for variety.
- **Reason:** Authorization enforcement (like Cloud delete-only-by-owner, now study-only-own-cards); prevents studying other users' cards; protects privacy; similar to Cloud requirement (show only authorized data).

2.28 Q28: When studying, the user should be able to filter lessons by category (e.g., study only "Vocabulary" cards).

Answer:

- **Modify Class 1:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 77 (state): create state to hold selected category filter - `val selectedCategoryFilter = remember mutableStateOf<String?>(null)`
- **Modify Class 2:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 107 (query logic): fetch only cards where category matches the selected filter - `fun getCardsForUserAndCat String, category: String?): Flow<List<FlashCard>> = null`
- **Modify Class 3:** `FlashCardDao.kt` **lines 15–66**
- **Action:** Add new query method at line 50: define query filtering by both createdBy and category - `@Query("SELECT * FROM FlashCard WHERE createdBy = :email AND category = :category ORDER BY RANDOM()")`
- **Explanation:** Store selected category in state; pass it to DAO query; fetch only matching cards; if no category selected, fetch all.
- **Reason:** Display/filtering requirement; lets users focus learning on one category at a time; improves user experience.

2.29 Q29: When studying, the user should be able to filter study sessions by difficulty level (Easy, Medium, Hard) to focus on cards matching their current skill level.

Answer:

- **Modify Class 1:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 76 (state): create state to hold selected difficulty filter - `val selectedDifficultyFilter = remember mutableStateOf<String?>(null)`
- **Action:** Add at lines 88–100 (UI above lessons): render difficulty filter buttons (All / Easy / Medium / Hard) - `@Composable fun StudyDifficultyFilterButtons(selected: String?, onSelect: (String?) -> Unit): Unit = null`
- **Modify Class 2:** `StudyScreen.kt` **lines 54–358**

- **Action:** Add at line 110 (LaunchedEffect query): fetch cards filtered by current user, category, and difficulty - `fun getCardsForUserCategoryAndDifficulty(email: String, category: String?, difficulty: String?): Flow<List<F = null`
- **Modify Class 3:** `FlashCardDao.kt` **lines 15–66**
- **Action:** Add query method at line 55: define query filtering by createdBy, category, and difficulty - `@Query("SELECT * FROM FlashCard WHERE createdBy = :email AND (:category IS NULL OR category = :category) AND (:difficulty IS NULL OR difficulty = :difficulty) ORDER BY RANDOM() LIMIT 3")`
- **Explanation:** Add difficulty filter state alongside category filter; render 4 filter buttons in UI; pass both filters to DAO query; fetch only cards matching both criteria; reload lesson on filter change.
- **Reason:** Display/filtering requirement (enables adaptive learning); helps users progressively strengthen skills by difficulty level; supports scaffolded learning patterns; improves motivation and engagement.

2.30 Q30: When studying, system should support filtering by easy/difficult binary level to streamline beginner-focused learning paths.

Answer:

- **Modify Class 1:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 77 (state): create state to hold binary difficulty filter (null for All, false for Easy, true for Difficult) - `val selectedBinaryDifficultyFilter = remember  mutableStateOf<Boolean?>(null)`
- **Action:** Add at lines 88–100 (UI above lessons): render binary filter buttons (All / Easy / Difficult) for quick selection - `@Composable fun StudyBinaryDifficultyFilterButtons(selected: Boolean?, onSelect: (Boolean?) -> Unit): Unit = null`
- **Action:** Add at line 110 (LaunchedEffect query): fetch cards filtered by current user, category, and binary difficulty - `fun getCardsForUserCategoryAndBinaryDifficulty(email: String, category: String?, isAdvanced: Boolean?): Flow = null`
- **Modify Class 2:** `FlashCardDao.kt` **lines 15–66**
- **Action:** Add query method at line 56: define query filtering by createdBy, category, and isAdvanced - `@Query("SELECT * FROM FlashCard WHERE createdBy = :email AND (:category IS NULL OR category = :category) AND (:isAdvanced IS NULL OR isAdvanced = :isAdvanced) ORDER BY RANDOM() LIMIT 3")`
- **Explanation:** Add binary difficulty filter state alongside category filter; render 3 filter buttons in UI; pass binary filter to DAO query; fetch only cards matching all criteria; reload lesson on filter change.
- **Reason:** Simplified filtering requirement; enables progressive learning with binary levels (Easy for beginners, Difficult for advanced); faster query performance; supports beginner-first learning strategies.

USE CASE 7: Audio Cache Management

2.31 Q31: When caching audio files, system should store cache metadata (filename, file size, last accessed, user email).

Answer:

- **Create:** New Room entity `AudioCache.kt`: define fields for id, filename, fileSize, lastAccessed timestamp, and userEmail - `@Entity data class AudioCache(val filename: String, val fileSize: Long, val userEmail: String)`
- **Modify Class 1:** `EditScreen.kt` **lines 43–293**
- **Action:** Add at line 260 (after saving audio file): locate the saved file to read its size - `fun getAudioFileSize(filename: String): Long = null`
- **Action:** Add at line 265: insert AudioCache metadata including filename, file size, current timestamp, and user email - `suspend fun recordAudioCache(cache: AudioCache): Unit = null`
- **Modify Class 2:** `StudyScreen.kt` **lines 54–358**
- **Action:** Add at line 145 (cache hit in playAudio lambda): update lastAccessed for that filename to current time - `suspend fun updateCacheAccessTime(filename: String): Unit = null`
- **Explanation:** After saving audio file to disk, insert metadata record with current timestamp; on replay (cache hit), update lastAccessed for LRU tracking.
- **Reason:** Data storage requirement (like Cloud storing metadata); enables cache management; prepares for LRU eviction and quota features; similar to Cloud requirement (store metadata for management).

2.32 Q32: When clearing audio cache, system should only delete current user's cache files (authorization enforcement similar to Cloud delete-only-by-owner).

Answer:

- **Modify Class 1:** `AudioCacheDao.kt` (DAO file)
- **Action:** Add at line 68 (new delete method): define DAO delete query that removes cache rows for a given user email - `@Query("DELETE FROM AudioCache WHERE userEmail = :email")`
- **Modify Class 2:** `MenuScreen.kt` **lines 80–95**
- **Action:** Add at line 85 (clear cache handler): read current user email from DataStore - `suspend fun readUserEmailForClear( String? = null`
- **Action:** Add at line 86: invoke DAO deletion for that user email - `suspend fun clearUserAudioCache(email: String): Unit = null`
- **Action:** Add at line 87: remove physical audio files in the user-specific cache directory - `fun deletePhysicalAudioFiles(emai String): Unit = null`
- **Action:** Add at line 88: show success feedback and refresh displayed cache stats - `fun showCacheClearSuccess(): Unit = null`
- **Explanation:** Query AudioCache with WHERE userEmail = current user; delete all matching records and physical files in user-specific directory.
- **Reason:** Authorization enforcement (identical to Cloud delete-only-by-owner pattern); prevents accidental deletion of other user's cache; protects privacy; similar to Cloud requirement (enforce user ownership).

USE CASE 8: Navigation History & Current Route Management

2.33 Q33: When user navigates between screens, system should store the navigation history with timestamps for analytics and debugging purposes.

Answer:

- **Create:** New Room entity `NavigationHistory.kt`: define fields for id, routeName, timestamp, and userEmail - `@Entity data class NavigationHistory(val routeName: String, val userEmail: String, val timestamp: Long)`
- **Modify Class 1:** `Navigator.kt` **lines 41–246**
- **Action:** Add at line 50 (NavHost setup): create NavigationHistoryDao instance - `val navigationHistoryDao = remember context.getDatabase().navigationHistoryDao()`
- **Action:** Add at line 52 (before route composition): capture current route name and timestamp - `fun captureNavigationEvent String): NavigationHistory = null`
- **Action:** Add at line 53 (after route changes): insert navigation history record into database - `suspend fun recordNavigationE NavigationHistory): Unit = null`
- **Explanation:** Before displaying each route, extract route name from NavGraphBuilder; capture current timestamp and user email; insert as NavigationHistory record for analytics.
- **Reason:** Data storage requirement (like Cloud storing metadata); enables usage analytics and debugging; tracks user journey; prepares for session reconstruction and crash analysis.

2.34 Q34: When displaying navigation screens, system should show the current active route and provide feedback about navigation state (loading, success, error).

Answer:

- **Modify Class 1:** `TopBarComponent.kt` **lines 15–41**
- **Action:** Add at line 20 (state setup): create state to hold current active route name - `val currentRoute = remember mutableStateOf("")`
- **Action:** Add at line 25 (display section): show current route as title in the top bar - `@Composable fun CurrentRouteTitle(rou String): Unit = null`
- **Modify Class 2:** `BottomBarComponent.kt` **lines 12–22**
- **Action:** Add at line 15 (state): create state for navigation status (IDLE, LOADING, SUCCESS, ERROR) - `val navigationStatus = remember mutableStateOf<NavigationStatus>(NavigationStatus.IDLE)`
- **Action:** Add at line 18 (display): show colored indicator and status message reflecting current navigation state - `@Composable fun NavigationStatusIndicator(status: NavigationStatus): Unit = null`
- **Modify Class 3:** `Navigator.kt` **lines 41–246**
- **Action:** Add at line 100 (route callback): update currentRoute state whenever navigation occurs - `fun updateCurrentRoute(r String): Unit = null`
- **Action:** Add at line 105 (error handler): update navigationStatus to ERROR with error message on navigation failure - `fun updateNavigationStatus(status: NavigationStatus, message: String?): Unit = null`
- **Explanation:** Extract active route from NavController; update top bar title in real-time; display status with visual feedback (color-coded); update BottomBarComponent with current state.
- **Reason:** Display requirement (like Cloud showing metadata); transparency shows user which screen they are on; status feedback enables debugging; similar to Cloud requirement (show current state and feedback).