WSB University

Faculty of Applied Sciences

Field of study:

Computer science

Student's names:

ALI TEBOURBI 56908

Łukasz Woś 53925

Task 2: Design Your Own App & Plan AI-Based Implementation

Dąbrowa Górnicza 2025/2026

Write your idea (Concept Brief)

"Cloud Sync Puzzle Quest" is a casual mobile puzzle game designed as a practical implementation and demonstration vehicle for the thesis topic: "Developing a mobile game with cloud integration for user data synchronization." The primary purpose of this app is twofold: to offer engaging puzzle-based entertainment to casual gamers and, more significantly, to serve as a robust showcase of seamless, reliable cloud data synchronization capabilities. The game targets casual mobile players, particularly those who utilize multiple devices (smartphones and tablets) or who frequently upgrade their hardware, ensuring their gameplay progress is never lost and always accessible.

The core gameplay revolves around solving a series of progressively challenging spatial logic puzzles (e.g., connecting nodes without crossing paths, fitting geometric shapes into complex patterns). Players unlock new levels based on performance and earn star ratings (1-3 stars) reflecting their efficiency or score. A simple collectible system adds an extra layer of engagement, rewarding players with unique items for completing level sets or achieving specific milestones. The cornerstone feature, directly supporting the thesis, is the **robust cloud synchronization** implemented using **Firebase Firestore**. This system automatically and near-instantly syncs all pertinent user data—including level completion status, star ratings, high scores, unlocked collectibles, and user settings—across all devices associated with the player's authenticated account.

"Cloud Sync Puzzle Quest" directly addresses the common user frustration of losing game progress due to device failure, reinstallation, or the desire to switch between devices. For the thesis, it tackles the technical challenge of implementing a dependable synchronization mechanism. Consider this scenario: a user plays several levels on their smartphone during their commute, earning stars and unlocking a new collectible. Later that evening, they open the game on their tablet, already logged into the same account. Thanks to the automatic Firestore background synchronization, the game seamlessly loads their latest progress, reflecting the levels completed and collectibles earned hours earlier on the phone, allowing them to pick up exactly where they left off without any manual intervention.

While numerous mobile puzzle games incorporate some form of cloud saving, "Cloud Sync Puzzle Quest" differentiates itself through its deliberate **focus on the quality and reliability of the synchronization mechanism** itself. The implementation serves as a core technical demonstration for the thesis work, potentially exploring aspects like efficient data structuring for sync, real-time updates, and foundational strategies for handling offline scenarios or basic data conflict resolution. The game, therefore, is not just a playable product but a testament to a well-engineered cloud integration solution designed to provide a truly seamless multi-device user experience.

Group brainstorming (collaborative step): Share your idea early with a few teammates (in class or online):

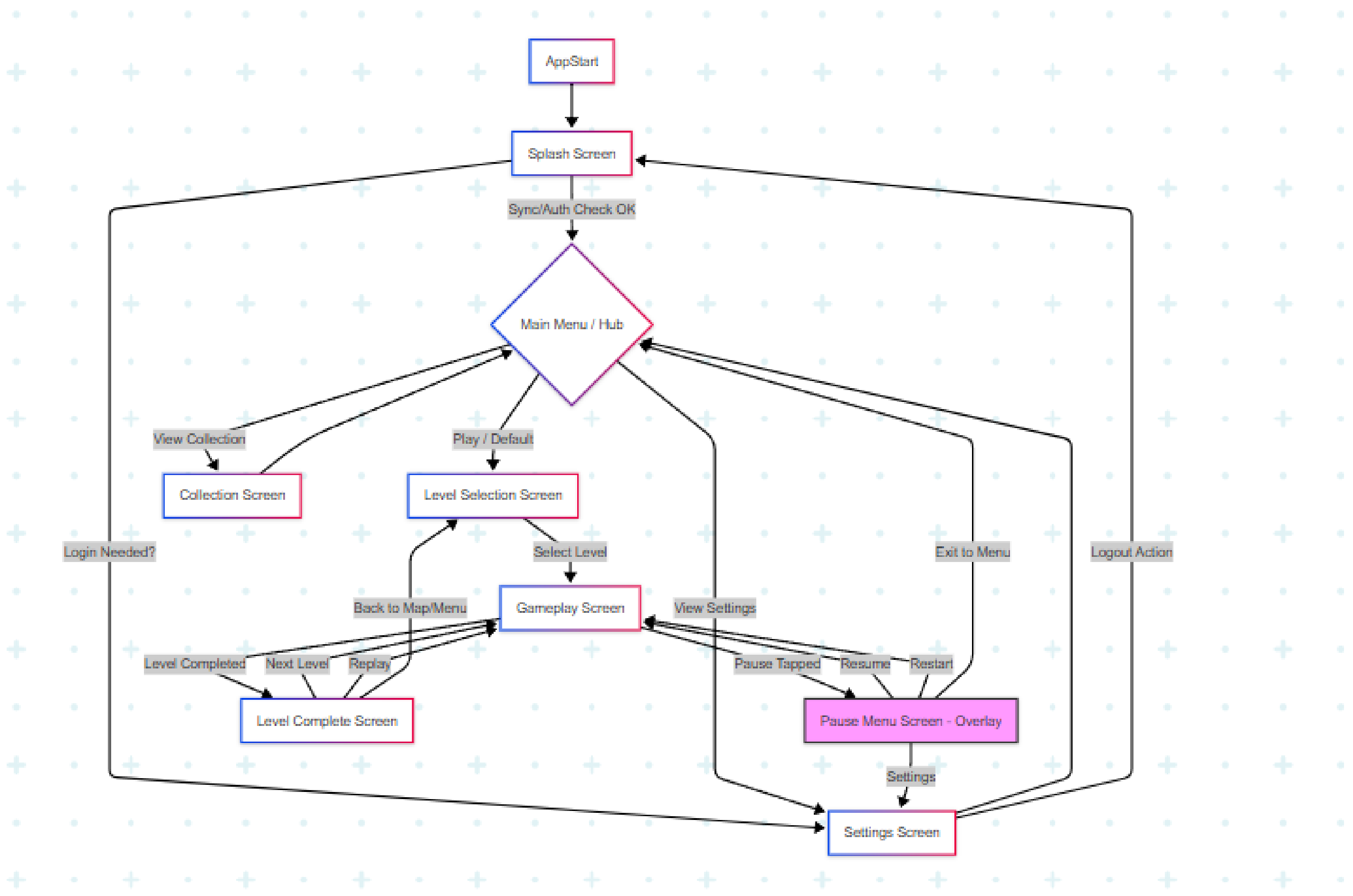
Opinion 1 (Gameplay): "So, the first person said, 'Yeah, the sync part makes sense for the thesis, but the puzzle game itself sounds a bit basic. Maybe spice it up? Like, add timed levels or let players unlock different looks for the puzzles with those collectibles you mentioned, just to make it more engaging.'"

Opinion 2 (Sync Tech): "Then the next classmate focused on the tech, asking, 'Okay, but what about when the sync gets tricky? Like, what happens if someone plays offline for a while? Or even plays offline on *two* different phones and then connects? How does it figure out which data is right?' They thought figuring out those 'what ifs' is pretty key for the thesis part."

Opinion 3 (Scope/Workload): "And the third person was thinking about the workload, basically saying, 'Nice idea, but making lots of puzzle levels, plus a collectible system, *plus* getting that cloud sync perfect... sounds like a *lot* of work for one thesis.' They suggested maybe starting with fewer levels or simpler collectibles, just to make sure you definitely nail the important sync feature.

Define architecture & complexity Plan your app in details :

Screen map + navigation (diagram).



Screen Flow Description:

1. **Splash Screen:**
    a. **Entry:** App launch.
    b. **Function:** Shows branding, checks authentication, performs initial cloud sync check.
    c. **Exits:** Automatically proceeds to the `Main Menu / Hub` after checks complete. May redirect to `Settings` (or a dedicated login flow) if user interaction is needed for login.
2. **Main Menu / Hub:**
    a. **Entry:** From `Splash Screen`. This is the central area, likely integrated with the Bottom Navigation Bar.
    b. **Function:** Provides primary navigation options.
    c. **Exits:**
        i. (Tap Play/Levels) -> `Level Selection Screen`
        ii. (Tap Collection) -> `Collection Screen` (Likely via Bottom Nav)
        iii. (Tap Settings) -> `Settings Screen` (Likely via Bottom Nav)
3. **Level Selection Screen:**
    a. **Entry:** From `Main Menu / Hub`. Accessible via Bottom Nav.

   b. **Function:** Displays the grid or list of puzzle levels, showing player progress (status, stars) loaded from synced data.
   c. **Exits:**
      i. (Tap a specific Level) -> `Gameplay Screen`
      ii. (Via Bottom Nav) -> `Collection Screen`, `Settings Screen`
      iii. (OS Back Button/Swipe) -> `Main Menu / Hub` (Typically)
4. **Gameplay Screen:**
   a. **Entry:** From `Level Selection Screen`.
   b. **Function:** Where the player actively solves the puzzle.
   c. **Exits:**
      i. (Tap Pause Button) -> `Pause Menu Screen` (Overlay)
      ii. (Puzzle Solved) -> `Level Complete Screen`
5. **Pause Menu Screen (Overlay):**
   a. **Entry:** From `Gameplay Screen` (appears on top).
   b. **Function:** Provides options during gameplay pause.
   c. **Exits:**
      i. (Tap Resume) -> `Gameplay Screen`
      ii. (Tap Restart) -> `Gameplay Screen` (reloads level)
      iii. (Tap Settings) -> `Settings Screen`
      iv. (Tap Exit/Quit) -> `Main Menu / Hub`
6. **Level Complete Screen:**
   a. **Entry:** From `Gameplay Screen` upon successful completion.
   b. **Function:** Shows results (stars, score), notifies about unlocked collectibles, triggers cloud sync of new progress.
   c. **Exits:**
      i. (Tap Next Level) -> `Gameplay Screen` (for the next level)
      ii. (Tap Replay) -> `Gameplay Screen` (reloads the same level)
      iii. (Tap Back to Map/Menu) -> `Level Selection Screen`
7. **Collection Screen:**
   a. **Entry:** From `Main Menu / Hub` (likely via Bottom Nav).
   b. **Function:** Displays all collectibles earned by the player, based on synced data.
   c. **Exits:**
      i. (Via Bottom Nav) -> `Level Selection Screen`, `Settings Screen`
      ii. (OS Back Button/Swipe) -> `Main Menu / Hub` (Typically)
8. **Settings Screen:**
   a. **Entry:** From `Main Menu / Hub` (via Bottom Nav) or `Pause Menu Screen`.
   b. **Function:** Contains options (sound, etc.), account management (login/logout), sync status information.
   c. **Exits:**
      i. (Via Bottom Nav) -> `Level Selection Screen`, `Collection Screen`

ii. (OS Back Button/Swipe) -> Previous Screen (e.g., `Main Menu / Hub` or `Pause Menu`)
iii. (Logout Action) -> Triggers navigation back to `Splash Screen` to handle re-authentication/sync state.

UserData Model

Represents the main document for each authenticated user in Firestore.

Field Name	Data Type	Description	Sync Notes
userId	String	Firebase Auth UID	Primary Key
displayName	String (Nullable)	User's display name	Syncs
lastLoginTimestamp	Timestamp	Server time of last login	Informational
lastSyncTimestamp	Timestamp	Server time of last successful write	Crucial for sync
totalStars	Number (int)	Aggregated stars earned	Syncs
settings	Map<String, dynamic>	User preferences (sound, etc.)	Syncs
levelProgress	Map<String, LevelProgressData>	User progress per level (key = levelId)	Core Sync Data
unlockedCollectibles	Map<String, Boolean>	Unlocked items (key = collectibleId, value=true)	Core Sync Data

LevelProgressData Model

Represents the nested data for a single level's progress within `UserData.LevelProgress`.

Field Name	Data Type	Description	Sync Notes
levelId	String	Implicit Key in the map	
status	String	Locked', 'Unlocked', 'Completed'	Needs Sync
starsEarned	Number (int)	Stars achieved (e.g., 0-3)	Needs Sync
highScore	Number (int)	Highest score for the level	Needs Sync
lastPlayedTimestamp	Timestamp	When the level was last played/completed	Informational

Collectible Model

Represents static game configuration data, likely stored in a separate collection or bundled.

Field Name	Data Type	Description	Sync Notes
collectibleId	String	Unique ID for the collectible type	Primary Key
name	String	Display name	Static Data
description	String	Flavor text	Static Data
rarity	String	e.g., 'Common', 'Rare'	Static Data
imageUrl	String (Nullable)	URL for the collectible's image	Static Data

MVP Core Features & Complexity Estimation

Authentication & Splash Screen Logic:

Estimated Complexity: Medium. This involves handling the different authentication states (checking if logged in, loading, errors) using Firebase Auth, potentially showing a basic UI, and performing an initial check or fetch of basic user data from Firestore.

AI-Promptability: High. AI tools are generally good at generating boilerplate code for Firebase authentication flows and basic loading/splash screen UI elements.

Tech Needed: Flutter UI for the splash screen, State Management (like Bloc or Provider) to handle the authentication state across the app, Firebase Authentication SDK, and Firestore SDK for initial user data reads.

Potential Blockers/Notes: Key challenges include handling authentication errors smoothly for the user, optimizing the initial data load time, and deciding which authentication providers (Google, Email, etc.) to support.

Level Selection Screen:

Estimated Complexity: Medium. Requires building a UI (likely a grid or list) that dynamically displays level information (status like locked/unlocked, stars earned) based on the user's progress data fetched via the Cloud Sync Service. Needs to handle loading and error states during data fetching.

AI-Promptability: High. Generating UI lists or grids based on data held in a state management solution is a common task for AI code assistants. Integrating the specific state logic is moderately promptable.

Tech Needed: Flutter UI (ListView/GridView components), State Management (to hold and provide the `levelProgress` data), and integration with the Cloud Sync Service to get the data.

Potential Blockers/Notes: Ensuring the screen efficiently displays potentially many levels and accurately reflects real-time updates if cloud data changes via listeners.

Gameplay Screen & Logic:

Estimated Complexity: Medium-High. The complexity here is heavily dependent on the specific puzzle mechanics chosen. It involves building the interactive puzzle UI, managing the internal state of the game during play, implementing the core puzzle-solving logic, and calculating scores.

AI-Promptability: Medium. AI can help significantly with the basic UI layout (High) and standard state management patterns (Medium), but the unique core logic of your specific puzzle type will require more manual implementation (Low-Medium promptability).

Tech Needed: Flutter UI (potentially using custom widgets, Canvas, or game engines integrated with Flutter), State Management for the gameplay state, and custom Dart code for the game logic.

Potential Blockers/Notes: The inherent complexity of the chosen puzzle mechanics, ensuring good performance during gameplay, and robustly managing the game's state transitions.

Cloud Sync Service/Repository:

Estimated Complexity: High. This is the central component for your thesis. It involves designing the Firestore data models for efficient synchronization, implementing reliable functions to fetch and save user progress, setting up real-time listeners for live updates, potentially implementing basic offline data caching/queuing strategies, and defining a basic conflict resolution approach (e.g., using timestamps).

AI-Promptability: Medium. AI can generate basic Firestore CRUD (Create, Read, Update, Delete) operations and listener boilerplate. However, the overall architectural design, robust error handling, offline strategies, and specific conflict resolution logic require significant developer design and input.

Tech Needed: Dart language, Firebase Firestore SDK, integration with your State Management solution, potentially a local database or preferences library (like `sqflite` or `shared_preferences`) if implementing offline caching.

Potential Blockers/Notes: This is the **Core Thesis Challenge**. Key difficulties include handling network interruptions and errors reliably, designing a workable (even if basic) offline data strategy, defining how sync conflicts are handled (e.g., 'last write wins'), implementing appropriate Firestore security rules, and thorough error handling for sync operations.

Level Complete Screen & Sync Trigger:

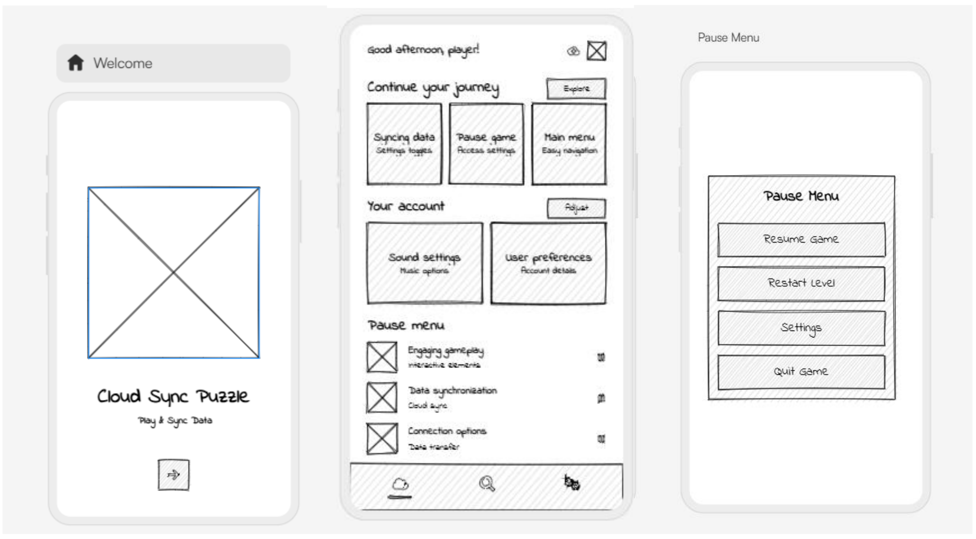
Estimated Complexity: Low-Medium. The UI itself is relatively simple (displaying stars, score, new collectibles). The main logic involves taking the results from the completed gameplay session and correctly triggering the `Cloud Sync Service` to save the updated progress to Firebase. It also needs to handle feedback from the sync service (success or failure).

AI-Promptability: High. Generating the UI to display results and writing the code to call a save function in your repository/service are tasks AI is well-suited for.

Tech Needed: Flutter UI, State Management (to receive results from gameplay and display sync status), and a dependency on the `Cloud Sync Service`.

Potential Blockers/Notes: Ensuring the correct data (score, stars, new collectibles, updated level status) is gathered and passed to the sync service. Providing clear feedback to the user if the synchronization fails and potentially offering a retry mechanism.

Possible Figma Prototype:



Prompt 1: UI Screen with Synced Data (Level Selection Screen)

Target: Flutter with Bloc state management. // Request: Generate a Flutter StatefulWidget called 'LevelSelectionScreen'. // // Requirements: // 1. Assume it receives a 'LevelSelectionBloc' instance via context (`context.read<LevelSelectionBloc>()`). //

2. Use a `BlocBuilder<LevelSelectionBloc, LevelSelectionState>` to react to state changes. //

3. Define a hypothetical `LevelSelectionState` with properties: // - `isLoading` (bool) // - `error` (String?) // - `levelProgress` (Map<String, LevelProgressData>?) - Keyed by levelId. // - Assume `LevelProgressData` has `status` (String: 'Locked', 'Unlocked', 'Completed') and `starsEarned` (int). //

4. Display a `CircularProgressIndicator` centered when `state.isLoading` is true. //

5. Display a centered error message (`Text`) when `state.error` is not null. //

6. When data is loaded (`state.levelProgress` is not null and not empty): // - Display levels using a `GridView.builder`. // - For each level (using `state.levelProgress.entries`), create a custom widget `LevelTile` (you can generate a basic placeholder for `LevelTile`). // - Pass the `levelId` (entry.key) and `progressData` (entry.value) to `LevelTile`. // - `LevelTile` should visually indicate: // - The level number/ID. // - If the level is 'Locked' (e.g., grayed out, lock icon). // - The stars earned (`progressData.starsEarned`) if 'Completed' (e.g., display 0-3 star icons). // - Make `LevelTile` tappable *only* if the status is not 'Locked'. On tap, it should add an event to the `LevelSelectionBloc`, e.g., `context.read<LevelSelectionBloc>().add(LevelSelectedEvent(levelId))`. //

7. Handle the case where `state.levelProgress` is null or empty after loading (e.g., display "No levels found"). //

8. Include basic padding around the GridView.

Prompt 2: Cloud Data Synchronization Logic (Save Progress Function)

// AI Prompt for GitHub Copilot/Cursor: // Target: Dart function within a Repository or Bloc class using Firebase Firestore. // Request: Generate an asynchronous Dart function `saveLevelCompletion`. // // Requirements: //

1. Function Signature: `Future<bool> saveLevelCompletion({required String userId, required String levelId, required int starsEarned, required int score, List<String>? newCollectibles})`. Return true on success, false on failure. //

2. Get a `FirebaseFirestore.instance`. //

3. Define the path to the user's document: `FirebaseFirestore.instance.collection('users').doc(userId)`. //

4. Prepare the data for Firestore update using a Map<String, dynamic>. The update should: // - Set `levelProgress.{levelId}.status` to 'Completed'. // - Set `levelProgress.{levelId}.starsEarned` to the provided `starsEarned` (consider logic for only updating if higher than existing, though simple set is okay for prompt). // - Set `levelProgress.{levelId}.highScore` to the provided `score` (consider logic for only updating if higher). // - Set `levelProgress.{levelId}.lastPlayedTimestamp` to `FieldValue.serverTimestamp()`. // - Atomically increment the `totalStars` field using `FieldValue.increment(starsEarned)`. // - If `newCollectibles` is not null and not empty, iterate through it and add entries to the `unlockedCollectibles` map, e.g., `unlockedCollectibles.{collectibleId}` set to `true`. Use dot notation for map fields. // - **Crucially, update the top-level `lastSyncTimestamp` field to `FieldValue.serverTimestamp()`**. //

5. Perform the update using the `update()` method on the `DocumentReference`. //

6. Wrap the Firestore call in a `try-catch` block. //

7. In the `catch` block, log the error (e.g., `print('Error saving progress: $e')`) and return `false`. //

8. If the update succeeds, return `true`.

// AI Prompt for GitHub Copilot/Cursor:

// Target: Dart logic within a Repository or Bloc class using Firebase Firestore.

// Request: Generate the setup for a real-time Firestore listener for a specific user's data.

//

// Requirements:

Prompt 3: Real-time Sync Listener Setup

// 1. Create a function or stream getter that returns a `Stream<UserData>`. Signature example: `Stream<UserData> getUserDataStream(String userId)`.

// 2. Inside the function/getter:

//   - Get a `FirebaseFirestore.instance`.

//   - Define the `DocumentReference` for the user: `FirebaseFirestore.instance.collection('users').doc(userId)`.

//   - Call the `.snapshots()` method on the `DocumentReference` to get a `Stream<DocumentSnapshot>`.

//   - Use the `.map()` method on the stream to transform `DocumentSnapshot` into `UserData` objects.

//   - Inside the `.map()` callback:

//   - Check if `snapshot.exists` and `snapshot.data()` is not null.

//   - If data exists, parse it into a `UserData` object. Assume a factory constructor `UserData.fromFirestore(Map<String, dynamic> data, String documentId)` exists. Handle potential parsing errors gracefully within the mapping if possible, or ensure `fromFirestore` is robust.

//   - If data doesn't exist or is null, handle appropriately (e.g., return a default `UserData` object, or signal that the user data doesn't exist - depends on app logic, maybe throw specific error).

// 3. Implement basic error handling for the stream itself (e.g., using `handleError` or `catchError` on the stream, or letting the Bloc's `addError` handle it upon subscription). Log errors encountered on the stream.

// 4. (Context for Bloc Usage - describe how it would be used): This stream would typically be subscribed to within a Bloc (e.g., in its constructor or an initialization method). When new `UserData` arrives from the stream, the Bloc would emit a new state containing the updated `UserData`. Ensure the `UserData` model includes all fields necessary (`userId`, `displayName`, `levelProgress`, `unlockedCollectibles`, etc.).

https://github.com/LukaszWoss/CloudSyncPuzzleQuest-DesignPlan