Technical Report

German Learning App

Technical Architecture and System Design

Author: Nico Abramowski
Co-creator: Tanya Nikolaeva Mitreva
Supervisor: Dr. Fatima Sapundzhi
Date: February 1, 2026

Abstract

This report describes the design and implementation of *German Learning App*, an offline-capable Android application for German vocabulary and grammar practice using algorithmic reinforcement. The app is built around a Spaced Repetition System (SRS) inspired by the SM-2 family of algorithms [7]. The implementation focuses on practical, day-to-day usability: a clean study flow, predictable review scheduling, and a daily limit for introducing new cards so study sessions remain manageable.

On the engineering side, the project follows Clean Architecture principles [6] and uses MVVM for UI state management [2]. The UI is implemented with Jetpack Compose [3]. Local persistence is handled via Room [5], and lightweight configuration is stored via Jetpack DataStore [1]. The result is a modular structure where the domain logic (card scheduling and session selection) remains testable and decoupled from Android-specific code, while the database layer provides reactive queries that keep dashboard counters and deck indicators consistent during study sessions.

Motivation

Most language apps either try too hard to be games or they are basically just digital word lists. Both approaches annoy me for the same reason: they do not respect the actual problem. Learning a language is not about being entertained for five minutes, it is about building long-term recall. If I do not see a word again at the right time, it disappears. If I do see it at the right time, it sticks.

German makes this even more obvious because it is not just vocabulary. The case system forces you to learn *structure* (articles, endings, sentence patterns) and not just translations. I wanted something that can support that kind of practice without relying on an internet connection, without distractions, and without turning every interaction into a gamified loop.

So this project became a practical tool and a software engineering exercise at the same time. It let me implement a real scheduling system, design a clean architecture, and build an app where data actually persists and the UI stays in sync with the database. The goal is not to compete with commercial platforms. The goal is to build something simple that I would personally trust for daily study.

Contents

1 Introduction

1.1 Context and Motivation

Learning a new language like German is difficult largely because long-term retention is difficult. Traditional approaches often fail when review is not structured: words are learned once and then forgotten. Spaced repetition systems address this by scheduling reviews at increasing intervals, adapting to the user's performance and the forgetting curve [7].

This project implements that idea as a focused Android application. The app is designed around daily study habits: it prioritizes due reviews, introduces a limited amount of new material per day, and works offline so that studying does not depend on network access or platform accounts.

1.2 Project Objectives

The primary goals of the project are:

- Implement a scientifically grounded SRS algorithm based on SM-2 principles.
- Provide structured content grouped into topics (e.g., Travel, Health) and sublevels that cover translations, nouns, sentences, and case-focused prompts.
- Keep the app fully offline-capable with local persistence.
- Ensure study sessions are predictable and bounded using a daily new-card limit (default 10, configurable per deck).

1.3 Scope of the Application

The app targets beginner to lower-intermediate learners (roughly A1/A2). It is a single-user Android app with a study mode, a dashboard for progress and workload, deck browsing by category, and a mixed study mode that can combine multiple decks into a custom session. The focus is stability, clean UX, and offline use rather than social features or cloud sync.

2 System Architecture

2.1 Architectural Overview

The app is built natively using Kotlin and Jetpack Compose [3]. The code is organized using Clean Architecture [6], splitting responsibilities into three layers:

- **Presentation:** Compose UI, ViewModels, and state handling.
- **Domain:** Business logic, domain models, and use cases.
- **Data:** Room database entities, DAOs, and repository implementations.

MVVM is used to connect the Presentation and Domain layers so that the UI observes reactive state rather than pulling data imperatively [2].

2.2 High-Level Diagram

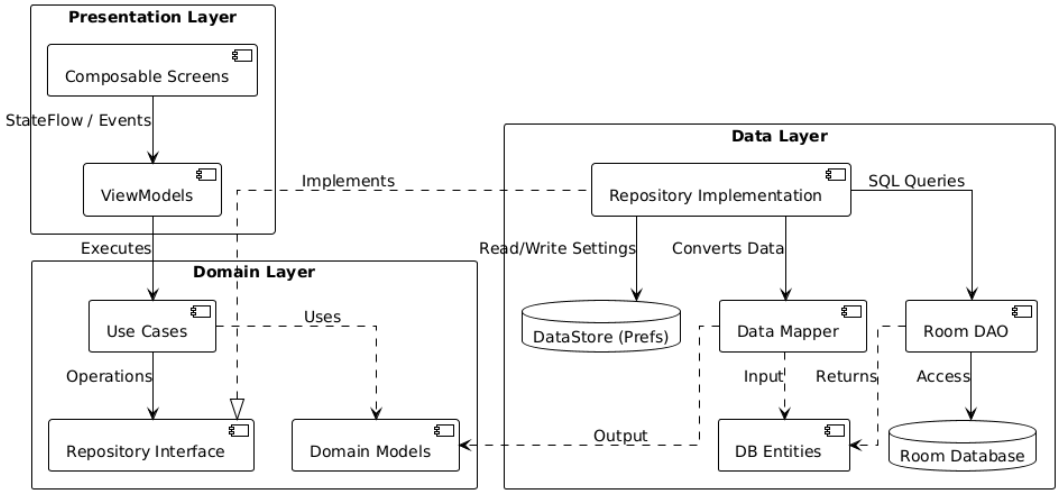

Figure 1: High-Level Architecture Diagram

2.3 Component Interaction

2.3.1 Presentation Layer

Compose screens render the UI and subscribe to ViewModel state exposed via `StateFlow`. For example, `StudyViewModel` holds the current card, session history (recent card IDs), and the session state. When the database changes (e.g., a card is rated and rescheduled), relevant counters update automatically because the UI collects reactive flows from the repository.

2.3.2 Domain Layer

The Domain layer defines the rules and is kept free of Android and database details. It contains the domain models (e.g., `Card`, `Deck`, `ReviewState`) and use cases that implement single responsibilities (e.g., "Get next due card" or "Rate card"). Repository interfaces define contracts for data access, allowing the domain logic to remain testable in isolation.

2.3.3 Data Layer

The Data layer implements the repository interfaces using Room [5]. DAOs provide the SQL queries that drive both the study session selection and the dashboard counters. Small settings (such as user preferences) are stored using Jetpack DataStore [1].

3 Domain Layer

3.1 Core Domain Models

- **Card:** Flashcard content (German/English prompts, type) associated with a deck.
- **Deck:** A group of cards with metadata like category and daily new-card limit.

- **ReviewState:** Scheduling state used by the SRS algorithm (due time, repetitions, interval, ease factor).

3.2 Repository Interfaces

The Domain layer defines interfaces such as `CardRepository` which expose operations like retrieving decks, obtaining candidate cards for sessions, updating review states, and logging study events. The Domain layer only depends on these contracts, not on Room or Android.

3.3 Use Cases (Interactors)

3.3.1 GetNextCardUseCase

The next-card logic prefers due reviews and avoids a strict deterministic order by shuffling among the top few candidates. A small look-ahead window is used (20 minutes) so that "Again" cards scheduled shortly ahead can re-enter the session smoothly.

Algorithm 1: Next Card Selection (Fuzzy Shuffle)

Input: deckId, recentCardIds
Output: Next Card to Study
candidates ← database.getDueCandidates(deckId, now + 20min);
available ← candidates.filter(id NOT in recentCardIds);
if *available is empty* **then**
  return null;
end
topCandidates ← available.take(5);
return topCandidates.shuffle().first();

3.3.2 RateCardUseCase

When the user rates a card (Again, Hard, Good, Easy), this use case retrieves the current `ReviewState`, computes the next state using the scheduler, persists it, and records a study log entry. The log is primarily used for statistics and for inferring how many new cards were introduced on a given day.

3.4 Spaced Repetition Scheduler

The scheduler is implemented in `CalculateNextReviewStateUseCase` and is inspired by SM-2. A key design choice is that "Again" does not erase all history permanently, but it reschedules the card immediately (e.g., now + 1 minute) so the user relearns it within the same session.

Algorithm 2: SRS Scheduling Logic (SM-2 Inspired)

```
Input: currentReviewState, rating (AGAIN, HARD, GOOD, EASY)
Output: newReviewState (dueAt, interval, easeFactor, repetitions)
repetitions ← currentReviewState.repetitions + 1;
if rating is AGAIN then
    interval ← 0;
    dueAt ← now + 1 minute;
    return newReviewState(interval, dueAt, repetitions,
     currentReviewState.easeFactor);
end
if repetitions == 1 (New Card) then
    if rating is GOOD then
        interval ← 1 day; dueAt ← now + 1 day;
    end
    if rating is EASY then
        interval ← 4 days; dueAt ← now + 4 days;
    end
end
else
    if rating is HARD then
        interval ← interval * 1.2; easeFactor ← easeFactor - 0.15;
    end
    if rating is GOOD then
        interval ← interval * easeFactor;
    end
    if rating is EASY then
        interval ← interval * easeFactor * 1.3; easeFactor ← easeFactor + 0.15;
    end
end
return newReviewState(interval, dueAt, repetitions, easeFactor);
```

4 Data Layer

4.1 Persistence Strategy

The app is offline-first, so local storage is the source of truth. Room [5] provides SQLite persistence with compile-time query validation. DataStore [1] is used for lightweight preferences.

4.2 Repository Implementation

`CardRepositoryImpl` coordinates between DAOs and the Domain layer. It maps Room entities to Domain models and exposes reactive flows to the ViewModels so UI counters remain consistent as the user studies.

4.3 Database Schema (Entity Models)

4.3.1 CardEntity (cards)

Stores the card content (front/back text) and its deck relationship.

4.3.2 ReviewStateEntity (review_states)

Stores scheduling data: due time, interval, ease factor, and repetitions. This is kept separate from card content so scheduling updates do not touch the static text data.

4.3.3 DeckEntity (decks)

Stores deck metadata such as name, category, and `daily_new_limit` (default 10).

4.3.4 StudyLogEntity (study_logs)

Stores study events as: `cardId`, `rating`, and `timestamp`. Deck membership is inferred via joins on the `cards` table when needed.

4.4 Data Access Objects (DAOs)

DAOs implement the core SQL used for:

- Selecting session candidates (due cards and new cards).
- Computing dashboard counters for due and new cards.
- Enforcing the daily new-card limit in a way that updates live during the session.

4.5 Daily New-Card Limit and Counter Logic

New cards are limited per deck per day. The dashboard "New" counter represents **remaining new cards for today**, not the total number of unseen cards in the deck.

Because the study log does not store an explicit "NEW" flag, the system infers "new introduced today" as: if a card's first-ever study log entry happens today, it counts as a new card introduced today. The remaining new cards are computed as:

$$remainingNew = \min(availableNew, \max(0, dailyLimit - introducedToday))$$

where `availableNew` is the number of cards with `repetitions = 0` in that deck.

Algorithm 3: Dashboard Counter Computation (Remaining New for Today)

Input: deckId, dailyLimit
Output: dueCount, remainingNew
availableNew ← COUNT(cards where repetitions == 0);
introducedToday ← COUNT(cards whose first log timestamp is today);
remainingNew ← $\min(availableNew, \max(0, dailyLimit - introducedToday))$;
dueCount ← COUNT(cards where repetitions > 0 AND dueAt within lookahead);
return (dueCount, remainingNew);

5 Core Algorithms and Data Flow

5.1 Dashboard Initialization

On startup, the app checks if the database is empty. If needed, a seeder populates starter decks and cards. The dashboard observes flows from the database, so counts update automatically when review states or logs change.

5.2 Study Session Orchestration

The `StudyViewModel` requests candidate cards for a deck and chooses the next card using the fuzzy shuffle strategy. Due cards are prioritized. New cards are only provided while `remainingNew` for the day is positive.

5.3 Mixed Study Session

Mixed study mode builds a candidate list from multiple decks. The selection prioritizes reviews over new cards and supports filtering by category and level. Daily limits are respected per deck through gating logic in the candidate queries.

Algorithm 4: Mixed Study Queue Builder (High-Level)

Input: MixedSessionConfig (Categories, Levels, Mode)
Output: Session Queue
targetDecks ← filterDecks(config.categories, config.levels);
candidates ← database.getMixedCandidates(targetDecks);
candidates.sortBy(reps > 0 DESC, dueAt ASC);
if *config.mode is NEW_ONLY* **then**
  candidates ← candidates.filter(reps == 0);
end
if *config.mode is DUE_ONLY* **then**
  candidates ← candidates.filter(reps > 0);
end
return candidates;

6 User Flow and Interaction

6.1 User Journey

The user starts on the Home screen, which provides a quick overview of categories and deck workload (due and remaining new for today). Selecting a deck starts a study session. A navigation drawer provides access to Mixed Study, Statistics, and Settings.

6.2 Flowchart Visualization

6.3 Interaction Design

The study screen is designed for low friction: show prompt, reveal answer, then rate using four clear buttons. Navigation follows standard Android patterns using Jetpack

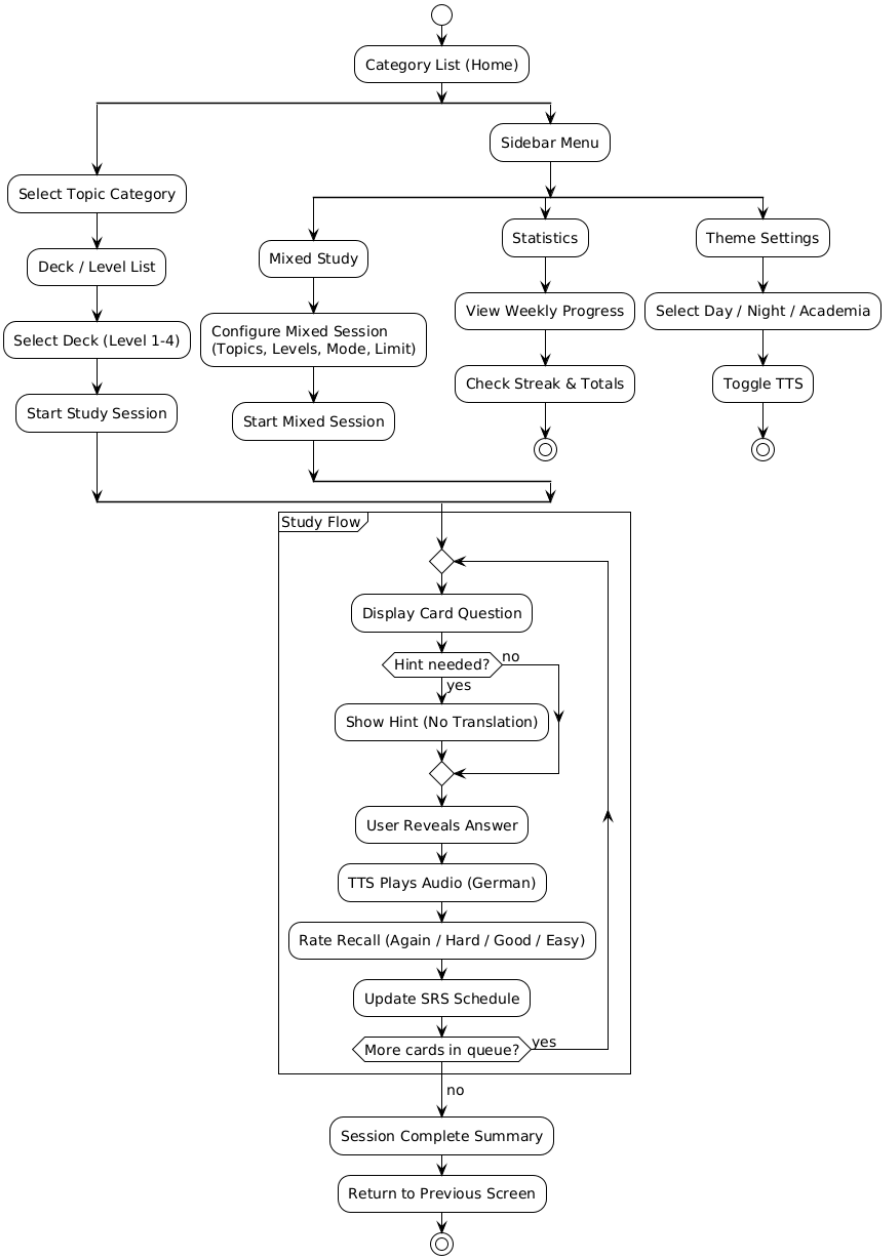

Figure 2: User Interaction Flowchart

Navigation [4]. The main goal is staying fast and predictable so daily study does not feel like a project.

7 User Interface (UI)

7.1 Navigation Structure

The app uses a single `NavHost` to switch screens. A drawer provides global navigation. The UI is theme-aware and supports day and night modes. Compose allows state-driven updates so the user sees counter changes immediately during a study session.

8 Conclusion and Future Work

The project results in a functional offline Android app with spaced repetition scheduling, deck organization by category, a bounded daily new-card system, and reactive counters that stay consistent during study. Clean Architecture keeps the codebase structured while features improve.

Future work includes:

- **Custom content:** Creating and editing decks/cards inside the app.
- **Cloud sync:** Optional syncing across devices.
- **Audio:** Pronunciation support and listening-based cards.
- **More analytics:** Better statistics and progress trends.

References

[1] Google Developers. Datastore. https://developer.android.com/topic/libraries/architecture/datastore, 2023. Accessed: 2023-10-26.

[2] Google Developers. Guide to app architecture. https://developer.android.com/topic/architecture, 2023. Accessed: 2023-10-26.

[3] Google Developers. Jetpack compose documentation. https://developer.android.com/jetpack/compose, 2023. Accessed: 2023-10-26.

[4] Google Developers. Navigation with compose. https://developer.android.com/jetpack/compose/navigation, 2023. Accessed: 2023-10-26.

[5] Google Developers. Save data in a local database using room. https://developer.android.com/training/data-storage/room, 2023. Accessed: 2023-10-26.

[6] Robert C Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[7] Piotr A Wozniak. Optimization of learning. *Master's Thesis, University of Technology in Poznan*, 1990. Retrieved from https://www.supermemo.com/en/archives1990-2015/english/ol/sm2.