Project Final Report

Lattice: A Hierarchical Schedule Manager

Yuwei Ye, Xian Gong
https://github.com/jamesye0003-blip/CS501-E1-Semester-Project

1 Overview of Features and Design

1.1 Core Features

Lattice implements a comprehensive hierarchical task management system with the following core features:

1.1.1 Task Management (CRUD Operations)

The application provides full Create, Read, Update, and Delete (CRUD) functionality for tasks. Users can create root tasks or nested sub-tasks with unlimited hierarchical depth. Each task supports rich metadata including:

- **Title and Description**: Textual content for task identification and detailed notes
- **Priority Levels**: Four-tier priority system (High, Medium, Low, None) with visual indicators
- **Time Management**: Optional due dates with support for all-day events or specific time points, including timezone awareness
- **Status Tracking**: Completion state (done/undone), postponement flags, and cancellation status
- **Attachments**: Support for file attachments including images, PDFs, and documents stored locally

1.1.2 Hierarchical Task Structure

The application's core design principle is unlimited hierarchical task decomposition. Tasks can be organized in a tree structure where:

- Each task can have multiple child tasks (sub-tasks)
- The hierarchy supports unlimited nesting levels (currently limited to 5 levels in the UI for usability)
- Tasks maintain parent-child relationships through `parentId` references
- The UI provides expand/collapse functionality for each node in the tree
- Cascade deletion: deleting a parent task automatically deletes all descendant tasks

The hierarchical view reduces cognitive load by allowing users to break down complex goals into actionable sub-tasks, improving focus and completion rates.

1.1.3 Speech-to-Text Integration

One of the key differentiators of Lattice is its voice-based task creation feature:

- **Microphone Recording**: Users can record audio directly within the task editor
- **Cloud Speech Recognition**: Integration with Google Cloud Speech-to-Text API for transcription
- **Flexible Input**: Recorded text can be assigned to either task title or description fields
- **Language Support**: Configurable language codes (e.g., "en-US", "zh-CN") for multilingual support
- **Error Handling**: Graceful handling of permission denials, recording failures, and API errors

This feature enables rapid task capture during commuting or micro-moments, addressing the fragmented "capture → organize → execute" workflow.

1.1.4 Task Filtering and Sorting

The application provides multiple ways to organize and view tasks:

- **Date-based Filtering**: Filter tasks by "Today", "This Week", "This Month", or "All"

- **Completion Filtering**: Toggle to show or hide completed tasks
- **Sorting Options**: Three sorting strategies:
  - **Title**: Alphabetical sorting by task title
  - **Priority**: Sorting by priority level (High ¿ Medium ¿ Low ¿ None)
  - **Time**: Sorting by due date and time
- **Layer-aware Sorting**: Sorting maintains hierarchical structure by applying sort order within each layer

1.1.5 Calendar View

A dedicated calendar screen provides a monthly view of tasks:

- **Monthly Grid**: Traditional calendar grid layout showing all days in the current month
- **Task Indicators**: Visual indicators on dates that have associated tasks
- **Date Navigation**: Month-by-month navigation with previous/next controls
- **Quick Task Creation**: Direct task creation from calendar dates
- **Task Editing**: Tap on dates to view and edit associated tasks

1.1.6 User Authentication and Multi-account Support

The application implements a complete authentication system:

- **Local Authentication**: Username/password-based authentication with local credential storage
- **Firebase Integration**: Optional Firebase Authentication for cloud-based account management
- **Offline Support**: Local authentication allows offline login using hashed password verification
- **Session Management**: Persistent sessions via DataStore, maintaining login state across app restarts
- **Multi-account Isolation**: User-specific task isolation ensuring data privacy

1.1.7 Cloud Synchronization

The application implements bidirectional synchronization with Firebase Firestore:

- **Incremental Sync**: Cursor-based incremental pull mechanism to avoid redundant data transfer
- **Conflict Resolution**: Last-Write-Wins (LWW) strategy with support for local dirty state detection
- **Automatic Sync**: Background synchronization triggered on login and task observation
- **Manual Sync**: User-triggered synchronization from profile screen
- **Soft Delete Support**: Tombstone-based deletion tracking for proper remote synchronization
- **Batch Operations**: Efficient batch processing for large task sets (respecting Firestore's 500-item limit)

1.1.8 User Profile and Statistics

The profile screen provides user management and task statistics:

- **User Information**: Display of username and authentication status
- **Task Statistics**:
  - Today's task completion count
  - Lifetime task statistics (total tasks, completed tasks)
  - On-time vs. postponed completion tracking
- **Bulk Operations**:
  - Postpone all today's tasks to tomorrow
  - Manual synchronization trigger
- **Settings**: Dark mode toggle, logout functionality

1.2 Design Principles

1.1.9 User Experience Design

- **Material Design 3**: Modern Material Design 3 components and theming for consistent, accessible UI

- **Dark Mode Support**: System-aware dark mode with manual override option
- **Responsive Layout**: Adaptive UI for different screen sizes (phone vs. tablet)
- **Accessibility**: Semantic labels, proper text scaling, and screen reader support
- **Visual Hierarchy**: Clear visual indicators for task priority, completion status, and hierarchical relationships

1.2.2 Data Persistence Strategy

- **Local-First Architecture**: All data stored locally for offline functionality
- **Optimistic Updates**: Immediate UI updates with background persistence
- **Conflict-Free Replication**: Designed for eventual consistency with cloud sync
- **Soft Deletes**: Tombstone-based deletion for proper sync and recovery

1.2.3 Performance Optimization

- **Incremental Loading**: Only load and sync changed data
- **Efficient Queries**: Indexed database queries for fast task retrieval
- **Reactive Updates**: Flow-based reactive data streams for automatic UI updates
- **Background Processing**: All I/O operations performed on background threads

2 Architecture and Technologies Used

2.1 Architectural Pattern

Lattice follows a **Clean Architecture** approach combined with **MVVM (Model-View-ViewModel)** pattern, organized into distinct layers with clear separation of concerns:

2.1.1 Layer Structure

1. **UI Layer** (`ui/`):
   - Jetpack Compose screens and components
   - Navigation graph and routing
   - Theme definitions (Material 3)
   - Pure presentation logic, no business rules
2. **ViewModel Layer** (`viewModel/`):
   - `TaskViewModel`: Manages task list state, filtering, and CRUD operations
   - `AuthViewModel`: Handles authentication state and user session
   - `EditorViewModel`: Manages speech-to-text UI state for task editor
   - Exposes UI state as `StateFlow` for reactive updates
   - Coordinates between UI and Domain/Data layers
3. **Domain Layer** (`domain/`):
   - **Models** (`model/`): Pure Kotlin data classes (Task, User, AuthState, Attachment, TimePoint)
   - **Repositories** (`repository/`): Interface definitions for data access contracts
   - **Services** (`service/`): Domain service interfaces (e.g., SpeechToTextService)
   - **Business Logic** (`sort/`, `time/`): Pure functions for sorting, filtering, and time conversion
   - No Android dependencies, fully testable in isolation
4. **Data Layer** (`data/`):
   - **Repositories** (`DefaultTaskRepository`, `DefaultAuthRepository`): Concrete implementations of domain interfaces
   - **Local Storage** (`local/`):
     - `room/`: Room database for persistent storage
     - `datastore/`: DataStore for preferences and session management
   - **Remote Storage** (`remote/firebase/`): Firebase Firestore integration

- **External Services** (`speech/`): Google Cloud Speech-to-Text API integration

2.2 Technology Stack

2.2.1 UI Framework

- **Jetpack Compose**: Modern declarative UI framework
  - Material 3 components and theming
  - Compose Navigation for screen routing
  - State management with `remember` and `StateFlow`
  - Recomposition optimization for performance
- **Material Design 3**: Complete Material 3 theming system
  - Dynamic color support
  - Dark/light theme switching
  - Custom color schemes for priority indicators

2.2.2 State Management

- **AndroidViewModel**: Lifecycle-aware ViewModels for state management
- **Kotlin Flow**: Reactive streams for asynchronous data
  - `StateFlow` for UI state
  - `Flow` for data streams from repositories
  - `flatMapLatest` for user-switching scenarios
- **rememberSaveable**: Compose state persistence across configuration changes

2.2.3 Local Data Persistence

- **Room Database** (v2.6.1):
  - SQLite-based local database
  - Type-safe DAOs (Data Access Objects)
  - Type converters for complex types (Instant, List¡Attachment¿)
  - Flow-based reactive queries
  - Foreign key constraints for data integrity
  - Indexed queries for performance
- **DataStore Preferences**:
  - Type-safe key-value storage
  - Used for user session (user ID)
  - Used for sync cursors (incremental sync state)
  - Used for user preferences (sort order, filter settings)

2.2.4 Remote Data and Synchronization

- **Firebase Authentication**:
  - Email/password authentication
  - User account management
  - Session persistence
- **Firebase Firestore**:
  - NoSQL document database
  - Real-time synchronization capabilities
  - Batch operations for efficiency
  - Incremental sync with cursor-based pagination
- **Sync Architecture**:

  - `FirebaseTaskSyncManager`: Dedicated sync manager for bidirectional synchronization
  - Conflict resolution: Last-Write-Wins with dirty state detection
  - Clock skew handling: 2-3 minute safety window
  - Mutex-based thread safety for concurrent sync operations

2.2.5 External API Integration

- **Google Cloud Speech-to-Text API**:
  - RESTful API via Retrofit
  - PCM audio encoding (16-bit, mono, 16kHz)
  - Base64 encoding for audio transmission
  - Language code configuration
  - Error handling and retry logic
- **Retrofit 2.9.0**: HTTP client library
  - Type-safe API definitions
  - Gson converter for JSON serialization
  - Coroutine support for asynchronous calls
- **OkHttp 4.12.0**: HTTP client implementation

2.2.6 Asynchronous Programming

- **Kotlin Coroutines**:
  - Suspend functions for asynchronous operations
  - `viewModelScope` for lifecycle-aware coroutines
  - `withContext(Dispatchers.IO)` for I/O operations
  - `flow` operators for reactive data transformation
- **Flow Operators**:
  - `flatMapLatest`: Switch to new user's data stream
  - `map`: Transform data between layers
  - `onStart`: Trigger side effects (e.g., auto-sync)
  - `collectAsState`: Convert Flow to Compose state

2.2.7 Platform Integration

- **Android Audio APIs**:
  - `AudioRecord` for microphone input
  - PCM audio capture and encoding
  - Permission handling for `RECORD_AUDIO`
- **File System**:
  - Internal storage for attachment files
  - FileProvider for secure file sharing
  - ContentResolver for file access
- **Content Providers**:
  - Image picker integration
  - File picker for document attachments
  - Camera integration for photo capture

2.3 Data Flow Architecture

2.3.1 Unidirectional Data Flow

The application follows a unidirectional data flow pattern:

1. **User Action** → UI triggers ViewModel method
2. **ViewModel** → Calls Repository interface
3. **Repository** → Coordinates local/remote data sources
4. **Data Source** → Updates local database or remote service
5. **Repository** → Emits updated data via Flow
6. **ViewModel** → Updates StateFlow
7. **UI** → Recomposes based on new state

2.3.2 Synchronization Flow

The sync mechanism operates as follows:

1. **Trigger**: User login, manual sync, or automatic background sync
2. **Pull Phase**:
   - Query Firestore for tasks with `updatedAt` ¿ cursor
   - Resolve conflicts using LWW strategy
   - Update local database
   - Advance sync cursor
3. **Push Phase**:
   - Identify dirty tasks (CREATED, UPDATED, DELETED status)
   - Batch upload to Firestore (respecting 500-item limit)
   - Mark tasks as SYNCED on success

2.4 Key Design Decisions

2.4.1 Separation of Concerns

- **Domain Logic Isolation**: Business logic (sorting, filtering, time conversion) moved to domain layer
- **Repository Pattern**: Abstract data access behind interfaces for testability
- **Dependency Inversion**: High-level modules depend on abstractions, not concrete implementations

2.4.2 Time Handling

- **UTC Storage**: All timestamps stored as UTC `Instant` in database
- **Timezone Awareness**: Source timezone preserved for proper display conversion
- **All-day vs. Specific Time**: Boolean flag distinguishes all-day events from time-specific tasks
- **Time Conversion Utilities**: Centralized time conversion logic in `domain/time/`

2.4.3 Offline-First Design

- **Local Persistence**: All data stored locally in Room database
- **Offline Authentication**: Local password hashing for offline login
- **Optimistic Updates**: UI updates immediately, sync happens in background
- **Graceful Degradation**: App functions fully without network connectivity

2.4.4 Code Organization

- **Feature-based Structure**: UI components organized by feature (screens, components)
- **Domain Utilities**: Business logic utilities grouped by concern (sort, time)
- **Data Layer Modularity**: Clear separation between local, remote, and external services
- **Consistent Naming**: Clear naming conventions for entities, DAOs, and repositories

2.5 Development Tools and Libraries

- **Kotlin**: Primary programming language (v1.9+)

- **Gradle**: Build system with Kotlin DSL
- **KAPT**: Kotlin Annotation Processing Tool for Room
- **Coil**: Image loading library for attachment previews
- **JSON**: Native Android JSON library for type conversion

3 Testing Approach and Results

3.1 Testing Methodology

Given the project's development timeline and scope, we adopted a pragmatic testing approach focused on manual testing and runtime debugging rather than comprehensive automated test suites. The primary testing strategy involves:

- **Manual Functional Testing**: Hands-on testing of all user-facing features on both Android emulator (API 28+) and physical devices
- **Logcat-based Debugging**: Real-time error monitoring through Android Studio's Logcat to identify runtime exceptions, crashes, and unexpected behaviors
- **Iterative Bug Fixing**: Immediate identification and resolution of issues discovered during development
- **Regression Testing**: Re-running complete user flows after each bug fix to ensure no regressions

3.2 Tested User Flows

The following critical user flows were systematically tested:

1. **Authentication Flow**:
   - User registration with username and password
   - Login with valid credentials
   - Offline login using locally stored credentials
   - Logout and session persistence
2. **Task Management Flow**:
   - Creating root tasks and nested sub-tasks
   - Editing task details (title, description, priority, time)
   - Toggling task completion status
   - Cascade deletion of parent tasks and all descendants
   - Task filtering (Today, This Week, This Month, All)
   - Task sorting (by Title, Priority, Time)
3. **Speech-to-Text Flow**:
   - Microphone permission request and handling
   - Audio recording and transcription
   - Error handling for API failures and permission denials
   - Text assignment to title or description fields
4. **Synchronization Flow**:
   - Automatic sync on login
   - Manual sync trigger from profile screen
   - Conflict resolution between local and remote data
   - Offline operation and subsequent sync
5. **Calendar and Profile Flow**:
   - Calendar view navigation and task display
   - Profile statistics display
   - Bulk operations (postpone tasks, sync)

3.3 Issues Discovered and Resolved

During development, several issues were identified through Logcat monitoring and manual testing:

- **Null Pointer Exceptions**: Fixed null safety issues in authentication repository when accessing user entities
- **Type Conversion Errors**: Resolved Room type converter issues for complex types (Instant, List¡Attachment¿)
- **UI State Management**: Fixed state persistence issues across configuration changes using `rememberSaveable`
- **Flow API Warnings**: Resolved experimental coroutines API usage warnings with proper `@OptIn` annotations
- **Memory Leaks**: Prevented ViewModel scope leaks by ensuring proper coroutine cancellation

3.4 Testing Results

Through systematic manual testing and Logcat monitoring:

- **All Core Features Verified**: All MVP features (task CRUD, hierarchy, speech-to-text, authentication) function correctly
- **No Critical Crashes**: Application runs stably without fatal crashes in normal usage scenarios
- **Offline Functionality Confirmed**: All local operations (task management, authentication) work without network connectivity
- **Sync Reliability**: Bidirectional synchronization successfully handles conflict resolution and data consistency
- **Performance Acceptable**: No noticeable performance issues with typical task loads (hundreds of tasks)

3.5 Limitations

The current testing approach has limitations:

- **Limited Automated Tests**: Only placeholder unit and instrumented tests exist; comprehensive test coverage would require significant additional development
- **Manual Testing Overhead**: Time-consuming manual testing process; automated UI tests would improve efficiency
- **Edge Case Coverage**: Some edge cases (e.g., extremely large task hierarchies, network timeout scenarios) may not be fully tested

4 Reflection on teamwork and Agile practices

4.1 Team member roles and responsibilities

- **Individual Feature Areas**
  - **Yuwei Ye.** UI framework & navigation, task hierarchy & list view, wireframes & interaction details, local data persistence
  - **Xian Gong.** Task editor, voice-based creation, calendar, settings & preferences, accessibility & multi-device adaptation, remote data persistence
- **Shared Responsibilities**: Naming & state conventions, privacy & permissions policy, documentation & demo

4.2 Agile (1-week sprints)

- Weekly planning + retrospective.
- Tools: GitHub Projects.
- Definition of Done: tests pass; smooth navigation; dark mode runs

4.3 Reflection

The clear division of responsibilities enabled efficient parallel development while maintaining consistency through shared conventions. The weekly sprint cycle with GitHub Projects provided excellent progress visibility and facilitated quick adaptation to technical challenges, such as Firebase sync implementation. We learned that features requiring deep integration between UI and data layers would benefit from more frequent pair programming to reduce merge conflicts and ensure architectural alignment.

5 AI Reflection

5.1 What tools were used?

We used ChatGPT and Gemini on Chrome.

5.2 What they were used for?

We asked ChatGPT how to improve code architecture and how to debug for specific problems shown by Logcat. We also asked ChatGPT how to write a correct and comprehensive commit after implementing features or satifying some other requirements such as architecuture refactoring. Moreover, we asked Gemini to help us adjust the theme, color, typography, and some other settings to beautify and polish the UI/UX.

5.3 Where AI was helpful vs. misleading.

For improving code architecture and debugging, AI is incredibly helpful. AI can also tell us how to write a beautiful screen and help us write comprehensive commits. However, for implementing a complex feature that cannot be easily described, AI cannot lend a hand. When I ask ChatGPT how to draw a flexible Gantt diagram with different types of calendars, it cannot give a perfect answer. The answer to our question used a variety of dependencies, and we could not understand the meaning of the code. Moreover, the code cannot run on my device, showing a huge number of bugs. Therefore, we gave up developing a calendar screen with a Gantt diagram. In the end, we chose to implement a simple diagram, only showing the tasks in one month.

5.4 How the team reviewed and refactored AI-generated code for quality, security, and Kotlin/Compose idioms.

If we would like to use AI to generate some code for implementing features, the most important thing is that the code should run correctly and as fast as possible. Some AI-generated code will use a great number of dependencies, but not all of them are useful, which means building the app may take a lot of time. Meanwhile, the developers may spend a lot of time on debugging. The chances are that the developers have debugged several times but cannot find the best solution.

In addition, AI-generated code may not focus on Kotlin/Compose idioms. They may use many outdated libraries to develop some components in the UI, which will lead to failure when building the app because some of the libraries have expired and cannot be used anymore. We must ensure that the libraries are the latest. If the AI-generated code uses some experimental feature, remember to add the 'Opt' tag.