Development Roadmap: Cinematic Wake

1 Project setup and foundations

- Choose tech stack
  - Kotlin, Android Jetpack, minSdk version, target SDK.
  - Likely choice: Kotlin + Jetpack (probability $\sim 90\%$).
- Create base project
  - Set up Gradle modules if needed (core, app, feature-animations).
  - Configure package name, signing configs (debug only for now).
- Define core data model
  - Entities: `ScheduledMoment` (time of day, enabled, linked animation), `Animation` (id, path, type), app settings.
  - Decide persistence: Room or DataStore. Room is more flexible for schedules (probability $\sim 70\%$).

2 Scheduling engine (time-based logic)

Goal: represent and store the times when animations should occur.

- Implement `ScheduledMoment` storage
  - CRUD operations: add, edit, delete, enable, disable.
  - Repository layer for schedules.
- Integrate `AlarmManager`
  - For each active `ScheduledMoment`, create an exact alarm (or as exact as allowed by API level).
  - Handle boot completed: reschedule alarms after device restart.
- Alarm receiver
  - BroadcastReceiver that receives alarm events and sets a flag like `pendingAnimationForId = X` in persistent storage.
- Edge cases
  - Multiple alarms close together: queue or merge.
  - Time zone changes and daylight saving: revalidate alarms when system time changes.

3 Hybrid trigger system (presence-based playback)

Goal: if a scheduled time is missed, play on next screen wake.

- Foreground service for event listening
  - Implement a lightweight foreground service that can react reliably to screen events on recent Android versions.
  - Show a minimal ongoing notification to respect system requirements.
- Screen-on detection
  - Register BroadcastReceiver for `ACTION_SCREEN_ON` and possibly `ACTION_USER_PRESENT`.
  - On screen-on:
    - Check `pendingAnimationForId`.
    - If not null, launch animation activity and clear the pending flag.
- Logic rules
  - Ensure only one animation triggers per wake event.
  - Avoid repeated playback due to multiple broadcasts.

4 Cinematic animation player activity

Goal: display the actual cinematic experience.

- Fullscreen activity
  - Immersive mode: hide system UI, navigation and status bars.
  - Use window flags: `FLAG_SHOW_WHEN_LOCKED`, `FLAG_TURN_SCREEN_ON` so the activity appears on top of the lock screen.
- Media engine
  - Integrate ExoPlayer (probability $\sim 80\%$ that this is the best choice).
  - Support local MP4 loops first; plan later for streaming or remote packs.
- UX behaviour
  - Fade in from black when the activity starts.
  - Duration options: fixed duration versus play-full-loop once.
  - Provide a simple gesture to dismiss or skip (for example tap or swipe).
- Handling device lock state
  - After the animation finishes, return to normal lock screen or previous state.
  - Never unlock the device programmatically.

5 Basic user interface (settings and configuration)

Goal: allow configuration of the experience.
Screens:

1. **Home / Dashboard**
   - List of scheduled moments (for example "Morning 07:00", "Evening 20:30").
   - Global enable / disable switch.
2. **Schedule editor**
   - Time picker for hour and minute.
   - Select associated animation.
   - Toggle active / inactive.
3. **Animation gallery**
   - Show available cinematic animations as thumbnails or short previews.
   - Allow selection of default animation for each schedule.
4. **Settings**
   - Toggle "play only when screen is turned on by the user" versus "auto wake at exact time" (if this option is kept).
   - Volume / mute preference.
   - Possibly a "test animation now" button.

The UI can be simple and functional in the MVP, with refined visual design later.

6 Persistence and state handling

- Implement repositories
  - `ScheduleRepository`, `AnimationRepository`, `SettingsRepository`.
- Storage choice
  - Room for schedules and animations metadata.
  - DataStore (preferences) for simple booleans (global enable, pending flags) (probability $\sim 75\%$ this split is a good balance).
- Pending animation state
  - Define a clear contract:
    - When alarm fires: set `pendingAnimationForId`.
    - When screen-on logic consumes it: clear it.
    - If the app crashes or device restarts, pending state should still be consistent.

7 Permissions, OS behaviour and battery

- Permissions handling
  - Exact alarms permission (Android 12+).
  - Foreground service usage and notifications.
  - Doze mode and battery optimisations: document that behaviour might be slightly device dependent.
- Battery impact minimisation
  - Keep foreground service work minimal.
  - Avoid unnecessary wakeups.
  - Reuse a single service for all screen events and internal tasks.

8 Testing and robustness

- Unit tests
  - Scheduling logic: next trigger time, rescheduling after time change, pending flag correctness.
  - Repository tests.
- Instrumented tests
  - Basic flows: create schedule, lock device, simulate alarm, then screen-on leading to animation playback.
  - Test across different API levels (at least one pre-Android 12 and one recent device).
- Manual device testing
  - Real devices from different manufacturers, as vendor customisations often affect lock screen behaviour.
  - Probability $\sim 90\%$ that this reveals at least one unexpected behaviour.

9 Pre-release, analytics and iteration

- Analytics (optional for MVP)
  - Log events such as animation played, skipped, schedule added or removed.
  - Helps understand real usage before monetisation.
- Crash and issue monitoring
  - Integrate a crash reporting tool.
  - Track rare edge cases, especially around lock screen and alarms.
- Beta release
  - Internal testing, then closed testing, then open release on Play Store.
- Refinement
  - Improve transitions, animation catalogue, and schedule UX based on feedback.
  - Introduce premium packs or subscriptions later, once engagement is validated.

9.1 Component grouping summary

UI layer

- **MainActivity** as navigation host
- **ScheduleListScreen** (list of scheduled moments)
- **ScheduleEditorScreen** (time picker, animation selection, enable or disable)
- **AnimationGalleryScreen** (select animation asset)
- **SettingsScreen** (global switches, sound, behavioural toggles)

Domain and logic layer

- **ScheduleUseCases** (create, update, delete, list schedules)
- **AlarmUseCases** (register, update, cancel alarms for schedules)
- **PendingAnimationUseCases** (set, clear, query pending animation based on alarms)

Data layer

- **ScheduleRepository** (CRUD for schedules in Room)
- **AnimationRepository** (metadata for animations, local asset mapping)
- **SettingsRepository** (DataStore based: global switches, pending animation id, last played times)
- Room database for structured data, DataStore for lightweight flags and settings

Background layer

- **AlarmScheduler** as abstraction over AlarmManager
- **AlarmReceiver** for alarm events that mark an animation as pending
- **ScreenEventService** as a small foreground service that stays eligible for screen on broadcasts
- **ScreenEventReceiver** listening to `ACTION_SCREEN_ON` and `ACTION_USER_PRESENT`, asking **PendingAnimationUseCases** whether an animation should start

Playback layer

- **AnimationPlayerActivity** in fullscreen immersive mode, with `FLAG_SHOW_WHEN_LOCKED` and `FLAG_TURN_SCREEN_ON` as needed
- Player implementation (likely ExoPlayer; estimated probability around 0.8 that this is the best choice)