Product Requirements Document: V1 - Spaced Repetition App with Discord Integration

Version: 1.1 (Updated for Levenshtein distance)
Date: April 2, 2025

1. Introduction & Goal

This document outlines the requirements for the first version (V1) of a web-based, Anki-style spaced repetition learning application. The primary goal of V1 is to provide core flashcard and deck management, a functional web-based study interface using a spaced repetition algorithm, and a unique, proactive Discord bot integration for reviewing cards via direct messages. The V1 should be simple enough for a beginner-intermediate developer to build but impressive enough to showcase the core value proposition.

2. Overall Scope (V1)

2.1. IN Scope for V1:

* **Authentication:** Google Social Login only.
* **Flashcards:** Front/Back text fields only. Manual creation + CSV import. Basic edit/delete.
* **Decks:** Private user decks. CRUD operations (Create, Rename, View, Delete) + adding cards.
* **Spaced Repetition System (SRS):** Implementation of SM-2 algorithm (or simpler fallback). Fixed default parameters. 4-button rating (Again/Hard/Good/Easy) *in the web app only*.
* **Web Study Interface:** Standard reveal/rate flow. Session summary page/modal.
* **Discord Account Linking:** Secure connection via Discord OAuth flow initiated from the web app.
* **Discord Bot Triggering:** Backend scheduler checks for due cards (in enabled decks) and triggers bot messages.
* **Discord Deck Configuration:** Users can enable/disable Discord reviews per deck via web app settings.
* **Discord Proactive Review:** Bot sends due card (Front) via DM.
* **Discord Correctness Check:** Bot checks user's reply answer using **string distance algorithms** (specifically Levenshtein distance, e.g., using the `leven` library) to determine similarity.
* **Discord SRS Update:** Bot provides Correct/Incorrect feedback in DM. Bot notifies web app backend API to update card's SRS state, assuming "Good" for correct and "Again" for incorrect.
* **Technology Stack:** React (Frontend), Node.js/Express (Backend), PostgreSQL (Database), discord.js (Discord Bot Library).

2.2. OUT of Scope for V1:

* Traditional email/password authentication, password reset, profile management.
* Advanced card formatting (images, audio, rich text, tags).
* Deck sharing, public decks, deck nesting (sub-decks).
* User configuration of SRS parameters.

* AI/LLM-based correctness checking for Discord replies.
* User input for SRS difficulty rating (Hard/Easy) via Discord.
* General LLM/chatbot conversational capabilities for the Discord bot.
* User commands to interact with the bot (e.g., fetch cards on demand).
* Detailed statistics or progress tracking beyond session summary.

3. Functional Requirements (V1)

3.1. User Authentication & Management

* **FR-AUTH-1:** System shall allow users to register and log in exclusively via Google Social Login (OAuth 2.0).
* **FR-AUTH-2:** Upon successful Google authentication, the system shall create a user account if one does not exist for the given Google ID, or log in the existing user.
* **FR-AUTH-3:** The system shall maintain user sessions after login.

3.2. Flashcards

* **FR-CARD-1:** System shall allow logged-in users to create flashcards with 'Front Text' and 'Back Text' within a specific deck.
* **FR-CARD-2:** System shall allow logged-in users to import flashcards into a specific deck from a CSV file.
    * **FR-CARD-2.1:** The CSV format shall be two columns (e.g., `Column A: Front Text`, `Column B: Back Text`). The system should handle or specify handling for a header row.
    * **FR-CARD-2.2:** The system shall provide feedback on the success or failure (with reasons, if possible) of the CSV import process.
* **FR-CARD-3:** System shall allow logged-in users to edit the Front and Back text of their existing flashcards.
* **FR-CARD-4:** System shall allow logged-in users to delete their existing flashcards.

3.3. Decks

* **FR-DECK-1:** System shall allow logged-in users to create new, private decks, requiring at least a Deck Name.
* **FR-DECK-2:** System shall display a list of the logged-in user's decks.
* **FR-DECK-3:** System shall allow logged-in users to rename their existing decks.
* **FR-DECK-4:** System shall allow logged-in users to delete their existing decks (this should likely delete all cards within that deck as well, or provide a warning).
* **FR-DECK-5:** All decks created by a user shall only be accessible by that user.

3.4. Spaced Repetition System (SRS)

* **FR-SRS-1:** The system shall implement the Anki SM-2 algorithm

(or a documented simpler alternative like exponential backoff) to schedule card reviews.
* **FR-SRS-2:** When a card is reviewed in the *web application*, the system shall use the user's rating (Again, Hard, Good, Easy) to calculate the next review date, interval, and potentially ease factor according to the chosen algorithm.
* **FR-SRS-3:** The SRS algorithm shall use hardcoded, sensible default parameters.
* **FR-SRS-4:** When a card is reviewed via *Discord*, the system shall update the card's SRS state based on a simple Correct/Incorrect outcome (mapping to "Good" / "Again" respectively).

3.5. Web Application Study Interface

* **FR-WEBSTUDY-1:** System shall allow users to initiate a study session for a specific deck.
* **FR-WEBSTUDY-2:** During a study session, the system shall present one due card at a time, initially showing only the Front Text.
* **FR-WEBSTUDY-3:** The system shall provide a mechanism (e.g., button) for the user to reveal the Back Text of the current card.
* **FR-WEBSTUDY-4:** After revealing the Back Text, the system shall present four distinct options for the user to rate their recall difficulty: "Again", "Hard", "Good", "Easy".
* **FR-WEBSTUDY-5:** Upon the user selecting a rating, the system shall record the review, update the card's SRS state via FR-SRS-2, and fetch the next due card for the session.
* **FR-WEBSTUDY-6:** Upon completion of all due cards in the session (or user manually exiting), the system shall display a session summary screen indicating the number of cards reviewed.

3.6. Discord Integration

* **FR-DISCORD-1:** System shall provide an interface in the web application (e.g., Settings page) for users to initiate linking their Discord account via Discord OAuth 2.0.
* **FR-DISCORD-2:** System shall securely store the necessary Discord User ID upon successful OAuth linkage to associate the web app user with their Discord identity.
* **FR-DISCORD-3:** System shall include a backend scheduler process that periodically checks for cards due for review for users with linked Discord accounts.
    * **FR-DISCORD-3.1:** The scheduler shall only consider cards belonging to decks explicitly enabled for Discord review by the user.
* **FR-DISCORD-4:** System shall allow users to enable or disable Discord reviews on a per-deck basis through the web application interface.
* **FR-DISCORD-5:** When the backend scheduler identifies a due card for Discord review, it shall trigger the Discord bot component.
* **FR-DISCORD-6:** The Discord bot shall send a Direct Message (DM) to the linked user containing the Front Text of the due card (and potentially the Deck Name).
* **FR-DISCORD-7:** The Discord bot shall listen for replies to its

review DMs.
* **FR-DISCORD-8:** Upon receiving a reply, the bot shall normalize the user's answer text and the card's correct Back Text (e.g., lowercase, trim whitespace) and compare them using the **Levenshtein distance algorithm** (e.g., via the `leven` library). It shall determine correctness by checking if the **normalized similarity derived from the distance** (e.g., `1 - distance / max_length`) meets a predefined threshold (e.g., 0.8).
* **FR-DISCORD-9:** The Discord bot shall provide feedback in the DM based on the comparison:
    * If Correct (similarity meets or exceeds threshold): Reply with positive confirmation (e.g., "✅ Correct!").
    * If Incorrect (similarity below threshold): Reply indicating incorrectness and **show the correct Back Text**.
* **FR-DISCORD-10:** The Discord bot shall make an authenticated API call to the web application backend after processing the user's reply.
    * **FR-DISCORD-10.1:** The API call payload shall include the card identifier and the outcome (Correct/Incorrect).
* **FR-DISCORD-11:** The web application backend shall expose an API endpoint to receive review outcomes from the Discord bot and update the corresponding card's SRS state according to FR-SRS-4.

4. Key User Flows (V1)

4.1. New User Onboarding & First Deck/Card

1. **User:** Navigates to the application's homepage/landing page URL.
2. **System:** Displays the landing page, highlighting the app's features and a prominent "Login with Google" button.
3. **User:** Clicks the "Login with Google" button.
4. **System:** Redirects the user to Google's OAuth consent screen.
5. **User:** Logs into their Google account (if not already logged in) and approves the application's request for basic profile information.
6. **System:** Google redirects the user back to the application's specified callback URL, providing an authorization code.
7. **System (Backend):** Exchanges the authorization code, retrieves user info, creates/logs in the user, establishes a session.
8. **System (Frontend):** Redirects the user to their main dashboard page (likely empty initially).
9. **User:** Clicks a "Create New Deck" button.
10. **System:** Presents a simple form asking for the Deck Name.
11. **User:** Enters a Deck Name and submits.
12. **System (Backend):** Creates a new deck record associated with the user.
13. **System (Frontend):** Displays the newly created deck with options ("Add Card", "Study", etc.).
14. **User:** Clicks the "Add Card" button for the new deck.
15. **System:** Presents a form with "Front Text" and "Back Text" fields.
16. **User:** Enters text for Front and Back and submits.

17. **System (Backend):** Creates a new card record associated with the deck, initializing its SRS state.
18. **System (Frontend):** Confirms card creation.

4.2. Web App Study Session

1. **User:** Logs in, navigates to their list of decks.
2. **System:** Displays decks, possibly indicating due card counts.
3. **User:** Clicks "Study" on a deck with due cards.
4. **System (Backend):** Fetches the next due card for the session.
5. **System (Frontend):** Displays the study interface showing only the card's Front Text and a "Show Answer" button.
6. **User:** Reads prompt, recalls answer, clicks "Show Answer".
7. **System (Frontend):** Reveals the Back Text. Displays rating buttons: "Again", "Hard", "Good", "Easy".
8. **User:** Clicks the appropriate rating button.
9. **System (Backend):** Receives card ID and rating. Updates card's SRS data. Fetches the next due card.
10. **System (Frontend):** If another card is due, repeats from step 5.
11. **System (Frontend):** If no more cards are due, navigates to the Session Summary screen.
12. **System:** Displays summary (e.g., cards reviewed). Provides navigation options.

4.3. CSV Card Import

1. **User:** Logs in, navigates to a specific deck's settings/management page.
2. **System:** Shows deck options, including "Import Cards from CSV". May show format instructions.
3. **User:** Clicks "Import Cards from CSV".
4. **System:** Presents a file upload input.
5. **User:** Selects a correctly formatted `.csv` file. Clicks "Upload" / "Import".
6. **System (Backend):** Receives file, parses CSV, creates new card records for valid rows, associates them with the deck, initializes SRS state. Handles errors.
7. **System (Frontend):** Displays feedback (success count, errors/skipped rows). Updates deck's card count.

4.4. Connecting Discord Account

1. **User:** Logs in, navigates to the web app's "Settings" / "Integrations" page.
2. **System:** Displays a "Connect Discord Account" button.
3. **User:** Clicks "Connect Discord Account".
4. **System:** Redirects user to Discord OAuth2 authorization URL with necessary scopes.
5. **User:** Sees Discord consent screen, clicks "Authorize".
6. **System (Discord):** Redirects user back to the web app's callback URL with an authorization code.
7. **System (Backend):** Receives code, exchanges it for tokens, fetches Discord User ID, securely stores User ID associated with the

web app user account.
8. **System (Frontend):** Displays confirmation (e.g., "Discord account connected as [Username]!") and potentially updates button text to "Disconnect...".

4.5. Enabling Discord Reviews for a Deck

1. **User:** Logs in, navigates to a specific deck's settings page.
2. **System:** Displays deck settings, including a toggle/checkbox "Enable Discord Reviews for this Deck" (may be disabled if Discord not linked).
3. **User:** Clicks the toggle/checkbox to enable.
4. **System (Frontend):** Updates UI state, sends API request to backend.
5. **System (Backend):** Receives request, updates deck record setting `discord_review_enabled = true`.
6. **(Later) User:** Clicks toggle/checkbox again to disable.
7. **System (Backend):** Updates flag to `discord_review_enabled = false`.

4.6. Discord Review Interaction

1. **System (Backend Scheduler):** Runs periodically. Queries DB for due cards for users with linked Discord and `discord_review_enabled=true` on the deck.
2. **System (Backend):** If due cards found, selects one or more per user, retrieves details (Front, Back, CardID, DeckName, DiscordUserID). Sends task(s) to Discord Bot service.
3. **System (Discord Bot):** Receives task(s). Finds user via Discord User ID. Manages sending prompts one at a time per user.
4. **System (Discord Bot):** Sends DM: `Time to review! Deck: [Deck Name]\n\nFRONT: [Card Front Text]`. Stores `card_id` and `Back Text` associated with this active prompt/user.
5. **User:** Receives DM, replies with answer text (e.g., `Vienna`).
6. **System (Discord Bot):** Receives reply. Retrieves stored `card_id` and `Back Text` for the active prompt.
7. **System (Discord Bot):** Compares user reply text with stored Back Text using the **Levenshtein distance algorithm to calculate similarity** (after normalization).
8. **System (Discord Bot - Correct):** If calculated similarity meets or exceeds threshold:
    * Sends DM: `✅ Correct!`
    * Calls backend API (`POST /api/reviews/discord`) with `card_id` and `outcome=correct`.
9. **System (Discord Bot - Incorrect):** If calculated similarity is below threshold:
    * Sends DM: `❌ Incorrect. The answer was: [Correct Back Text]`
    * Calls backend API (`POST /api/reviews/discord`) with `card_id` and `outcome=incorrect`.
10. **System (Backend):** API endpoint receives call. Updates card's SRS state based on `outcome` (mapping to "Good" or "Again"). Records review event.
11. **System (Discord Bot):** Clears state for the completed prompt.

If more cards are queued for the user, sends the next prompt.

5. Technology Stack (Proposed)

* **Frontend:** React
* **Backend:** Node.js with Express.js framework
* **Database:** PostgreSQL
* **Discord Bot:** discord.js library (for Node.js)
* **String Comparison Library:** `leven` (or similar maintained library for Levenshtein distance)
* **Deployment:** TBD (e.g., Heroku, Render, Vercel, AWS/GCP/Azure)

(Added specific mention of `leven` or similar library to tech stack)

6. Non-Functional Requirements (V1 Considerations)

* **NFR-1 (Usability):** Interfaces (web and bot interaction) should be clean, intuitive, and easy to navigate for core V1 features.
* **NFR-2 (Reliability):** The SRS scheduling and Discord notification mechanism should function reliably under typical load for a small initial user base. Basic error handling should be implemented.
* **NFR-3 (Security):** User data (Google profile info, Discord ID, card content) must be handled securely. Standard practices for web security (HTTPS, input validation, secure session management, secure API authentication between bot/backend) must be followed. Passwords are not stored due to Social Login only.
* **NFR-4 (Performance):** Basic operations (loading decks, studying cards, bot responses) should feel responsive. Database queries should be reasonably efficient.