Implementation Plan: V1 - Spaced Repetition App with Discord Integration

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

1. Overview & Goal Recap

This plan outlines the technical approach for building V1 of the web-based spaced repetition application with proactive Discord review capabilities. The goal is to implement the core features defined in the PRD V1.1, focusing on Google authentication, basic deck/card management, web-based study using SRS, and the unique Discord review flow using Levenshtein distance for answer checking.

2. Technology Stack (Confirmed)

* **Frontend:** React
* **Backend:** Node.js with Express.js
* **Database:** PostgreSQL
* **Discord Bot:** Node.js with `discord.js` library
* **Scheduler:** `node-cron` library (running within the backend process)
* **String Comparison Library:** `leven` (for Levenshtein distance)

(Added specific mention of `leven`)

3. Database Schema (PostgreSQL)

We will start with the following core tables. SQL types are illustrative; exact PostgreSQL types should be used (e.g., `VARCHAR`, `TEXT`, `INT`, `FLOAT`, `TIMESTAMPTZ`, `BOOLEAN`).

Table: `users`

* `id`: SERIAL PRIMARY KEY (or UUID)
* `google_id`: VARCHAR UNIQUE NOT NULL (ID from Google)
* `email`: VARCHAR UNIQUE NOT NULL
* `display_name`: VARCHAR
* `created_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP

Table: `decks`

* `id`: SERIAL PRIMARY KEY (or UUID)
* `user_id`: INT NOT NULL (Foreign Key referencing `users.id` ON DELETE CASCADE)
* `name`: VARCHAR NOT NULL
* `discord_review_enabled`: BOOLEAN DEFAULT FALSE
* `created_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
* `updated_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP

Table: `cards`

* `id`: SERIAL PRIMARY KEY (or UUID)

* `deck_id`: INT NOT NULL (Foreign Key referencing `decks.id` ON DELETE CASCADE)
* `front_text`: TEXT NOT NULL
* `back_text`: TEXT NOT NULL
* `created_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
* `updated_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
    * **SRS Fields:**
    * `interval`: INT DEFAULT 0 (Days until next review after rating)
    * `ease_factor`: FLOAT DEFAULT 2.5 (Multiplier for interval growth)
    * `next_review_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP (When the card is next due)

Table: `discord_links` (For connecting Web User to Discord User)

* `id`: SERIAL PRIMARY KEY
* `user_id`: INT UNIQUE NOT NULL (Foreign Key referencing `users.id` ON DELETE CASCADE)
* `discord_user_id`: VARCHAR UNIQUE NOT NULL (Discord User Snowflake ID)
* `discord_username`: VARCHAR (For display purposes, optional)
* `linked_at`: TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP

(Note: Add appropriate indexes on foreign keys and frequently queried columns like `google_id`, `discord_user_id`, `cards.next_review_at`)

4. Backend API (Node.js / Express)

4.1. Authentication Strategy

* **Login:** Use Google OAuth 2.0 flow.
    1. Frontend redirects to Google Consent Screen.
    2. User approves, Google redirects to backend callback URL (`/api/auth/google/callback`) with an authorization code.
    3. Backend exchanges code for Google tokens, fetches user profile (email, google_id, name).
    4. Backend finds existing user by `google_id` or creates a new user in the `users` table.
    5. Backend generates a JSON Web Token (JWT) containing `userId`.
    6. Backend sets the JWT in a secure, `HttpOnly` cookie sent back to the browser.
    7. Backend redirects frontend to the main application page (e.g., `/dashboard`).
* **Session Management:** JWT stored in `HttpOnly` cookie.
* **API Authentication:** Implement Express middleware to:
    1. Extract JWT from the request cookie.
    2. Verify the JWT signature and expiry using a secret key.
    3. If valid, extract `userId` and attach it to the `request` object (e.g., `req.user = { id: userId }`).
    4. If invalid or missing, reject the request (401 Unauthorized), unless the endpoint is public.

4.2. API Endpoint Definitions

(Base URL: `/api`)
(Auth: "Required" means valid JWT cookie needed)

Route	Method	Auth	Purpose	Request Body / Params / Query	Response Body (Success Example)
`/auth/google`	GET	None	Redirects user to Google OAuth consent screen.	-	Redirect
`/auth/google/callback`	GET	None	Handles Google redirect, logs in/signs up user.	Google `code` in query params	Redirect to frontend, sets JWT cookie
`/auth/logout`	POST	Required	Clears the JWT cookie.	-	`{ message: "Logged out" }`
`/auth/me`	GET	Required	Get current logged-in user info.	-	`{ id, email, displayName }`
`/decks`	POST	Required	Create a new deck.	`{ name: "Deck Name" }`	`{ id, userId, name, discordReviewEnabled, ... }`
`/decks`	GET	Required	Get all decks for the logged-in user.	-	`[ { id, name, cardCount*, dueCount* }, ... ]`
`/decks/:deckId`	PUT	Required	Rename a deck. Enable/disable Discord reviews.	`{ name?: "New Name", discordReviewEnabled?: true/false }`	Updated `{ id, userId, name, discordReviewEnabled, ... }`
`/decks/:deckId`	DELETE	Required	Delete a deck (and its cards).	`deckId` in path param	`{ message: "Deck deleted" }`
`/decks/:deckId/cards`	POST	Required	Create a new card in a deck.	`{ frontText: "Q", backText: "A" }`	`{ id, deckId, frontText, backText, ...srs fields }`
`/decks/:deckId/cards`	GET	Required	Get all cards in a deck (for management view).	`deckId` in path param	`[ { id, frontText, backText, ... }, ... ]`
`/decks/:deckId/cards/import`	POST	Required	Import cards from CSV file.	`deckId` in path param, `multipart/form-data` with CSV file	`{ successCount: N, errorCount: M, errors: [...] }`
`/cards/:cardId`	PUT	Required	Edit a card's text.	`{ frontText?: "New Q", backText?: "New A" }`	Updated `{ id, deckId, frontText, backText, ... }`
`/cards/:cardId`	DELETE	Required	Delete a card.	`cardId` in path param	`{ message: "Card

deleted" }` |
`/study/:deckId/next`	GET	Required	Get the next due card for a study session.	`deckId` in path param	`{ id, frontText, backText }` or `null` if none due
`/study/review`	POST	Required	Submit a review result from the web app.	`{ cardId: "...", rating: "Again"	"Hard"	"Good"	"Easy" }`	`{ message: "Review recorded" }`
`/integrations/discord/link`	GET	Required	Redirect user to Discord OAuth flow.	-	Redirect
`/integrations/discord/callback`	GET	Required	Handles Discord redirect, links accounts.	Discord `code` in query params	Redirect to frontend settings page, stores link
`/reviews/discord`	POST	**Bot Only**	Receives review outcome from Discord bot.	`{ cardId: "...", outcome: "correct"	"incorrect", botApiKey: "SECRET" }`	`{ message: "Discord review recorded" }`
`/internal/cards/:cardId`	GET	**Bot Only**	Get card details needed for bot DM.	`cardId` in path param, `botApiKey` header/query	`{ frontText, backText, deck: { name } }`

* * `cardCount`, `dueCount` can be calculated via DB query when fetching decks.
* **Bot Only Auth:** The `/reviews/discord` and `/internal/cards/:cardId` endpoints need protection. Use a simple shared secret API key (`botApiKey`) passed in a request header (e.g., `X-Bot-API-Key`) or body/query param that only the backend and bot know. Header is generally preferred.

5. Core Logic Implementation Details

5.1. Spaced Repetition System (SRS)

* **Algorithm:** Based on SM-2 principles (or simplified exponential backoff if SM-2 seems too complex initially).
* **State Stored Per Card:** `interval` (int days), `ease_factor` (float, starts ~2.5), `next_review_at` (timestamp).
* **New Cards:** `interval = 0`, `ease_factor = 2.5`, `next_review_at = NOW()`.
* **Web Review Update (`POST /study/review`):**
    * Input: `cardId`, `rating` (Again, Hard, Good, Easy).
    * Logic: Implement SM-2 update rules based on rating to calculate new `interval`, `ease_factor`, and `next_review_at = NOW() + interval days`.
        * `Again`: Reset interval (e.g., to 0 or 1 day), decrease ease factor (min 1.3).
        * `Hard`: Increase interval slightly based on previous interval and ease factor. Small decrease in ease factor.
        * `Good`: Increase interval based on previous interval and ease factor. Ease factor unchanged.
        * `Easy`: Increase interval significantly based on previous interval and ease factor. Increase ease factor.

    * Reference: Find a clear SM-2 implementation guide online.
* **Discord Review Update (`POST /reviews/discord`):**
    * Input: `cardId`, `outcome` (correct, incorrect), `botApiKey`.
    * Logic: Map outcome to a fixed web rating and use the *same* SRS update logic function as above.
        * `correct` -> Simulate a `"Good"` rating.
        * `incorrect` -> Simulate an `"Again"` rating.
    * Update the card's `interval`, `ease_factor`, `next_review_at` in the database.

5.2. CSV Import (`POST /decks/:deckId/cards/import`)

* Use a library like `csv-parser` for Node.js.
* Expect a two-column CSV. Allow optional header row (detect or require specification).
* Column 1 -> `front_text`, Column 2 -> `back_text`.
* For each valid row, create a new card record associated with `:deckId`, initializing SRS fields to default values.
* Perform bulk insert if possible for efficiency.
* Track successful imports and any rows that caused errors (e.g., missing columns).
* Return a summary report.

5.3. Backend Scheduler (`node-cron`)

* **Schedule:** Run every 5 minutes (Cron pattern: `*/5 * * * *`).
* **Task:**
    1. Query the database: Find `cards` where `next_review_at <= NOW()`.
    2. Join with `decks` and `discord_links` tables.
    3. Filter for cards where `decks.discord_review_enabled = true` AND a corresponding `discord_links` entry exists for the `decks.user_id`.
    4. Group results by `discord_user_id`.
    5. For each user with due cards, trigger the Discord Bot logic (see section 6.3) with the `discordUserId` and the list of their due `cardId`s.

6. Discord Bot (`discord.js`)

6.1. Authentication

* Use a Bot Token obtained from Discord Developer Portal. Store securely (environment variable).
* Bot connects to Discord Gateway using `discord.js` client.

6.2. Account Linking

* The bot itself is not directly involved in the OAuth flow. The web backend handles linking user accounts via OAuth and stores the `discord_user_id` in the `discord_links` table.

6.3. Receiving & Handling Review Tasks

* The backend scheduler will likely call an exported function from the bot's code (if running in the same project/monorepo) or send a message (if separate processes). Assume direct function call for V1.
* Input to bot function: `discordUserId`, `dueCardIds: string[]`.
* **Queue/Throttling Logic (per user):**
    * Bot maintains an internal queue or list of due cards for each user (e.g., `userQueues = new Map<DiscordUserID, CardID[]>()`).
    * Bot uses the **in-memory map** `activePrompts = new Map<DiscordUserID, { cardId: string, backText: string, promptMessageId: string, timestamp: Date }>()`.
    * Bot has a function `sendNextPromptIfIdle(discordUserId)`:
        1. If user `discordUserId` has cards in their queue AND does *not* have an entry in `activePrompts`:
        2. Dequeue one `cardId`.
        3. Fetch card details (`frontText`, `backText`, `deckName`) from the backend API: `GET /api/internal/cards/:cardId` (authenticated with `botApiKey`).
        4. Fetch the Discord User object using `discordUserId` via `discord.js`.
        5. Send DM: `Time to review! Deck: [Deck Name]\n\nFRONT: [Card Front Text]`.
        6. Store `message.id` (the ID of the DM sent by the bot).
        7. Update map: `activePrompts.set(discordUserId, { cardId, backText, promptMessageId: message.id, timestamp: new Date() })`.

6.4. State Management & Reply Handling

* Listen for `messageCreate` events, checking if the message is in a DM and not from the bot itself.
* `on messageCreate(message)`:
    1. Get `discordUserId = message.author.id`.
    2. Check if `activePrompts.has(discordUserId)`.
    3. If yes, retrieve `{ cardId, backText, promptMessageId } = activePrompts.get(discordUserId)`.
    4. **(Optional but Recommended):** Check if `message.reference?.messageId === promptMessageId` to ensure the user is replying *directly* to the prompt message. If not, ignore.
    5. Perform String Comparison Check (see 6.5).
    6. Determine if `isCorrect` based on the comparison result.
    7. Send Feedback DM (see 6.6).
    8. Call Backend API (see 6.7) with `cardId` and `outcome` (`correct` or `incorrect`).
    9. **Remove entry:** `activePrompts.delete(discordUserId)`.
    10. **Trigger next card:** Call `sendNextPromptIfIdle(discordUserId)` to potentially send the next queued card.

6.5. String Comparison Check (Levenshtein)

* Get user's reply text: `userAnswer = message.content`.
* Get correct answer: `correctAnswer = backText` (from the map).
* **Normalize both:** Define a normalization function (e.g., `normalize(str) => str.toLowerCase().trim().replace(/\s+/g, ' ')`). Apply it: `normalizedUserAnswer = normalize(userAnswer)`,

`normalizedCorrectAnswer = normalize(correctAnswer)`.
* **Calculate Distance:** Use the `leven` library: `distance = leven(normalizedUserAnswer, normalizedCorrectAnswer);`.
* **Calculate Normalized Similarity:**
    * `const maxLength = Math.max(normalizedUserAnswer.length, normalizedCorrectAnswer.length);`
    * `if (maxLength === 0) return distance === 0; // Handle empty strings`
    * `const similarity = 1 - (distance / maxLength);`
* **Compare:** `isCorrect = similarity >= 0.8;` (Using the 0.8 threshold).

6.6. Feedback Mechanism

* Based on the `isCorrect` value determined in 6.5:
    * **If `isCorrect` is true:** Send DM: `✅ Correct!`
    * **If `isCorrect` is false:** Send DM: `❌ Incorrect. The answer was: ${backText}`

6.7. Backend Communication

* After processing the reply:
    * Determine `outcome = isCorrect ? "correct" : "incorrect"`.
    * Make a `POST` request to the backend endpoint: `POST /api/reviews/discord`.
    * Request Body: `{ cardId: cardId, outcome: outcome, botApiKey: process.env.DISCORD_BOT_API_KEY }`. (Or send key in header).
    * Use `axios` or `node-fetch` for the API call. Ensure the `botApiKey` is stored securely as an environment variable.

6.8 Timeout Handling
* Implement a periodic check (e.g., using `setInterval` every hour) that iterates through `activePrompts`.
* If `Date.now() - entry.timestamp > TIMEOUT_DURATION` (e.g., 24 hours), remove the entry: `activePrompts.delete(userId)`. Call `sendNextPromptIfIdle(userId)` afterwards to potentially send the next card.

7. Frontend (React)

7.1. Authentication Flow

* `LoginPage`: Simple page with "Login with Google" button.
* Button click redirects browser to backend `/api/auth/google`.
* After Google login and backend processing, user is redirected to `/dashboard` (or similar). The JWT cookie is now set.
* Implement a way to check auth status on page load (e.g., call `/api/auth/me`) and redirect to login if not authenticated. Store user info (from `/api/auth/me`) in React state/context.

7.2. API Interaction

* Use `Workspace` API or install `axios`.
* Configure API client to automatically include credentials

(cookies) with requests: `Workspace(url, { credentials: 'include' })` or `axios.defaults.withCredentials = true;`.
* Create helper functions or hooks for common API calls (e.g., `useDecks`, `createDeck`, `addCard`, `getNexStudyCard`, `submitReview`).
* Handle API loading states and errors gracefully in the UI.

7.3. Component Structure (High-Level)

* **Routing:** Use `react-router-dom`.
* **Pages:** `LoginPage`, `DashboardPage` (main view after login), `DeckViewPage` (viewing/managing single deck), `StudyPage`, `SettingsPage`.
* **Components:** `Navbar`, `DeckList`, `DeckListItem`, `CreateDeckForm`, `CardList`, `CardListItem`, `CardForm`, `CSVImportForm`, `StudyCard`, `StudyRatingButtons`, `SessionSummary`, `DiscordLinkButton`, `DeckSettingsForm`.

7.4. State Management

* Start with React's built-in `useState`, `useReducer`, and `useContext` for managing local and shared state (like auth status, user info, maybe list of decks).
* Consider a lightweight global state library like Zustand later if prop drilling or context management becomes too complex.

7.5. Web Study Interface Flow (`StudyPage`)

1. Fetch next due card using `GET /study/:deckId/next`.
2. Display `frontText` and "Show Answer" button.
3. On button click, reveal `backText` and show rating buttons ("Again", "Hard", "Good", "Easy").
4. On rating button click:
    * Send review using `POST /study/review` with `cardId` and `rating`.
    * Fetch the next card.
    * If no more cards, show `SessionSummary` component.

8. Development & Deployment Considerations

* **Version Control:** Use **Git** from the start. Initialize a repository.
* **Environment Variables:** Use `.env` files (and add `.env` to `.gitignore`) to store sensitive information:
    * `DATABASE_URL`
    * `GOOGLE_CLIENT_ID`
    * `GOOGLE_CLIENT_SECRET`
    * `GOOGLE_CALLBACK_URL`
    * `JWT_SECRET`
    * `DISCORD_BOT_TOKEN`
    * `DISCORD_CLIENT_ID`
    * `DISCORD_CLIENT_SECRET`
    * `DISCORD_CALLBACK_URL`
    * `DISCORD_BOT_API_KEY` (shared secret between backend/bot)

    * `FRONTEND_URL` (for redirects)
    * `PORT`
* **Deployment:** TBD (as per PRD). Consider platforms like Render, Heroku, or Fly.io which handle Node.js/React/Postgres reasonably well. Need separate processes/services for Backend API and Discord Bot.

9. Implementation Order / First Steps

It's best to build incrementally:

1. **Project Setup:** Initialize Git repo, set up Node.js backend project (Express), React frontend project, install core dependencies (`express`, `pg`, `jsonwebtoken`, `passport`, `passport-google-oauth20`, `node-cron`, `discord.js`, `leven`, `react`, `react-router-dom`, `axios`/`Workspace`, `csv-parser`). Set up `.env`.
2. **Database Setup:** Create initial tables (`users`, `decks`, `cards`, `discord_links`) using SQL scripts or a migration tool (like `node-postgres-migrate`).
3. **Authentication:** Implement Google OAuth backend flow and JWT cookie setup. Implement basic React login page and `/api/auth/me` check.
4. **Deck CRUD:** Implement backend API endpoints for decks (Create, Read, Rename, Delete). Implement basic frontend UI to list, create, rename, delete decks.
5. **Card CRUD:** Implement backend API endpoints for cards (Create, Read, Edit, Delete within a deck). Implement frontend UI (e.g., in `DeckViewPage`) to manage cards.
6. **Web Study Flow:** Implement backend (`/study/...` routes) and frontend (`StudyPage`) for the web-based review cycle, including SRS updates.
7. **Discord Linking:** Implement backend (`/integrations/discord/...`) and frontend settings UI for linking Discord account.
8. **Discord Bot - Core:** Set up basic `discord.js` bot, connect to gateway. Implement backend scheduler (`node-cron`) to find due cards. Create internal API endpoint for bot to fetch card details.
9. **Discord Bot - Review Flow:** Implement the bot receiving tasks, queueing, sending DMs, handling replies (state map, Levenshtein comparison), providing feedback, and calling the backend `/reviews/discord` endpoint (including bot auth). Implement timeout handling.
10. **CSV Import:** Implement backend API and frontend UI for CSV uploads.
11. **Refinement & Testing:** Thoroughly test all flows, refine UI/UX, fix bugs.