==========

PART 1: AI GENERATION LOGIC (geminiService.ts)

This file handles interactions with the Google Gemini API to generate study materials,

analyze documents, and create quizzes.

==========

```
import { GoogleGenAI, Type, Schema, GenerateContentResponse } from "@google/genai";
import { StudyData, QuestionMode, QuizQuestion, TheoryQuestion, Formula, GeneratorParams, QuizConfig, InteractiveQuestion, QuizType } from "../types";
import { jsonrepair } from "jsonrepair";

const ai = new GoogleGenAI({ apiKey: process.env.API_KEY });

// --- HELPERS ---

// Retry wrapper for API calls
const withRetry = async <T>(fn: () => Promise<T>, retries = 3, delay = 1000): Promise<T> => {
  try {
    return await fn();
  } catch (error: any) {
    if (retries <= 0) throw error;

    // Log retry attempt
    console.warn(`API call failed. Retrying in ${delay}ms... (Attempts left: ${retries})`, error.message);
```

```
    await new Promise(resolve => setTimeout(resolve, delay));
    return withRetry(fn, retries - 1, delay * 2); // Exponential backoff
  }
};

const safeParseJSON = (text: string): any => {
  if (!text) return {};

  // 1. Basic cleanup: remove Markdown code blocks
  let cleaned = text.replace(/```json\s*/g, '').replace(/```\s*/g, '').trim();

  // 2. Robust extraction of JSON object
  // Find the first '{' and the last '}'
  const firstOpen = cleaned.indexOf('{');
  const lastClose = cleaned.lastIndexOf('}');

  if (firstOpen !== -1 && lastClose > firstOpen) {
    // Perfect case: we found both bounds
    cleaned = cleaned.substring(firstOpen, lastClose + 1);
  } else if (firstOpen !== -1) {
    // Truncated case: we have a start but no end. Take everything from start.
    cleaned = cleaned.substring(firstOpen);
  } else {
    // No JSON object found
    return {};
  }
```

```
 try {
   return JSON.parse(cleaned);
 } catch (e) {
   try {
     // 3. Use jsonrepair for common errors (missing quotes, trailing commas, unescaped quotes, truncation)
     const repaired = jsonrepair(cleaned);
     return JSON.parse(repaired);
   } catch (e2) {
     // 4. Last ditch effort: Escape backslashes that might be LaTeX and unescaped newlines
     try {
       // Replace literal newlines with \n if they are inside the string (heuristic)
       // and escape single backslashes that might be LaTeX commands
       let patched = cleaned.replace(/\\([^\\])/g, '\\\\$1');
       const repairedPatched = jsonrepair(patched);
       return JSON.parse(repairedPatched);
     } catch (e3) {
       console.error("JSON Parsing failed completely. Input sample:", cleaned.substring(0, 200));
       // Return empty object to prevent crashes
       return {};
     }
   }
 }
};

// Creates the content part for the API based on mime type
const getContentPart = (data: string, mimeType: string) => {
```

```
  if (mimeType === 'text/plain') {
    return { text: data };
  } else {
    return { inlineData: { data: data, mimeType: mimeType } };
  }
};

// --- SCHEMAS ---

const analysisResponseSchema: Schema = {
  type: Type.OBJECT,
  properties: {
    analysis: {
      type: Type.STRING,
      description: "A comprehensive study guide using Markdown headers (##), bullet points, and bold text. Detailed but concise.",
    },
    keyTopics: {
      type: Type.ARRAY,
      items: { type: Type.STRING },
      description: "A list of 5-8 main keywords/topics.",
    },
  },
  required: ["analysis", "keyTopics"]
};

const structuredDataSchema: Schema = {
  type: Type.OBJECT,
```

```
  properties: {
    formulas: {
      type: Type.ARRAY,
      description: "A list of 5-10 mathematical formulas or methods.",
      items: {
        type: Type.OBJECT,
        properties: {
          title: { type: Type.STRING },
          expression: { type: Type.STRING, description: "LaTeX format. Escape backslashes for JSON (e.g. '\\frac' -> '\\\\frac')." },
          method: { type: Type.STRING, description: "Usage explanation." },
          section: { type: Type.STRING }
        },
        required: ["title", "expression", "method"]
      },
    },
    quiz: {
      type: Type.ARRAY,
      description: "5-10 Practice MCQs.",
      items: {
        type: Type.OBJECT,
        properties: {
          id: { type: Type.INTEGER },
          question: { type: Type.STRING },
          options: { type: Type.ARRAY, items: { type: Type.STRING } },
          correctIndex: { type: Type.INTEGER },
          explanation: { type: Type.STRING },
          section: { type: Type.STRING },
```

```
      },
      required: ["id", "question", "options", "correctIndex", "explanation"]
    },
  },
  theoryQuestions: {
    type: Type.ARRAY,
    description: "5-8 Open-ended theory questions.",
    items: {
      type: Type.OBJECT,
      properties: {
        question: { type: Type.STRING },
        answer: { type: Type.STRING, description: "The final short answer or result." },
        explanation: { type: Type.STRING, description: "Step-by-step method or detailed explanation." },
        section: { type: Type.STRING },
      },
      required: ["question"]
    }
  },
  flashcards: {
    type: Type.ARRAY,
    description: "10-15 Flashcards for key terms/concepts.",
    items: {
      type: Type.OBJECT,
      properties: {
        id: { type: Type.INTEGER },
        term: { type: Type.STRING, description: "The concept or term." },
        definition: { type: Type.STRING, description: "A concise definition or explanation." }
```

```
    },
    required: ["id", "term", "definition"]
   }
  },
  mindMap: {
   type: Type.OBJECT,
   description: "A hierarchical structure representing the document's topics. Root -> Children -> Grandchildren.",
   properties: {
    label: { type: Type.STRING, description: "The main topic." },
    children: {
     type: Type.ARRAY,
     items: {
      type: Type.OBJECT,
      properties: {
       label: { type: Type.STRING, description: "Subtopic or concept." },
       children: {
        type: Type.ARRAY,
        items: {
         type: Type.OBJECT,
         properties: {
          label: { type: Type.STRING },
          children: {
           type: Type.ARRAY,
           items: {
            type: Type.OBJECT,
            properties: {
             label: { type: Type.STRING },
```

```
                children: {
                  type: Type.ARRAY,
                  items: {
                    type: Type.OBJECT,
                    properties: {
                      label: { type: Type.STRING }
                    }
                  }
                }
              }
            }
          }
        }
      }
    },
    required: ["label"]
  }
 },
 required: ["formulas", "quiz", "theoryQuestions", "flashcards", "mindMap"]
};

const testPaperSchema: Schema = {
 type: Type.OBJECT,
```

```
 properties: {
  quiz: {
   type: Type.ARRAY,
   items: {
    type: Type.OBJECT,
    properties: {
     id: { type: Type.INTEGER },
     question: { type: Type.STRING },
     options: { type: Type.ARRAY, items: { type: Type.STRING } },
     correctIndex: { type: Type.INTEGER },
     explanation: { type: Type.STRING },
     section: { type: Type.STRING },
    },
    required: ["id", "question", "options", "correctIndex", "explanation"]
   },
  },
  theoryQuestions: {
   type: Type.ARRAY,
   items: {
    type: Type.OBJECT,
    properties: {
       question: { type: Type.STRING },
       answer: { type: Type.STRING, description: "The final short answer or result. If not applicable, leave empty." },
       explanation: { type: Type.STRING, description: "Step-by-step method or detailed explanation." },
       section: { type: Type.STRING },
    },
    required: ["question"]
```

```
    }
   }
  },
  required: ["quiz", "theoryQuestions"]
};

const diagramSchema: Schema = {
   type: Type.OBJECT,
   properties: {
      diagramSVG: { type: Type.STRING, description: "Valid SVG code string. Simple, black stroke, white fill." }
   }
};

const moreFormulasSchema: Schema = {
  type: Type.OBJECT,
  properties: {
   formulas: {
    type: Type.ARRAY,
    items: {
      type: Type.OBJECT,
      properties: {
        title: { type: Type.STRING },
        expression: { type: Type.STRING },
        method: { type: Type.STRING },
        section: { type: Type.STRING }
      },
      required: ["title", "expression", "method"]
```

```
    },
   }
  }
};

const moreQuestionsSchema: Schema = {
  type: Type.OBJECT,
  properties: {
    quiz: {
      type: Type.ARRAY,
      items: {
        type: Type.OBJECT,
        properties: {
          id: { type: Type.INTEGER },
          question: { type: Type.STRING },
          options: { type: Type.ARRAY, items: { type: Type.STRING } },
          correctIndex: { type: Type.INTEGER },
          explanation: { type: Type.STRING },
          section: { type: Type.STRING },
        },
        required: ["id", "question", "options", "correctIndex", "explanation"]
      }
    },
    theoryQuestions: {
      type: Type.ARRAY,
      items: {
          type: Type.OBJECT,
          properties: {
```

```
          question: { type: Type.STRING },
          answer: { type: Type.STRING },
          explanation: { type: Type.STRING },
          section: { type: Type.STRING },
        },
        required: ["question"]
    }
   }
 }
};

const topicsSchema: Schema = {
 type: Type.OBJECT,
 properties: {
  topics: {
   type: Type.ARRAY,
   items: { type: Type.STRING },
   description: "List of curriculum topics"
  }
 },
 required: ["topics"]
};

const customQuizSchema: Schema = {
 type: Type.OBJECT,
 properties: {
  questions: {
   type: Type.ARRAY,
```

```
    items: {
      type: Type.OBJECT,
      properties: {
        id: { type: Type.INTEGER },
        type: { type: Type.STRING, description: "One of: MULTIPLE_CHOICE, FILL_IN_THE_BLANK, OPEN_ENDED" },
        question: { type: Type.STRING, description: "For FILL_IN_THE_BLANK, use '[____]' to mark the blank." },
        options: { type: Type.ARRAY, items: { type: Type.STRING }, description: "Only for MULTIPLE_CHOICE" },
        correctIndex: { type: Type.INTEGER, description: "Only for MULTIPLE_CHOICE" },
        answer: { type: Type.STRING, description: "For FILL_IN_THE_BLANK (exact word) or OPEN_ENDED (model answer)" },
        explanation: { type: Type.STRING },
        keywords: { type: Type.ARRAY, items: { type: Type.STRING }, description: "List of 3-5 required keywords for checking OPEN_ENDED/FIB answers." }
      },
      required: ["id", "type", "question", "explanation"]
    }
  }
 },
 required: ["questions"]
};

// --- EXPORTS ---

export const processDocument = async (data: string, mimeType: string = 'application/pdf'): Promise<StudyData> => {
 try {
  const model = "gemini-2.5-flash";
```

```
  const contentPart = getContentPart(data, mimeType);

  // Request 1: Detailed Analysis
  // Adjusted prompt to be comprehensive but not force excessive length that causes truncation
  const analysisPromise = withRetry<GenerateContentResponse>(() => ai.models.generateContent({
    model: model,
    contents: {
     parts: [
      contentPart,
      { text: `You are an expert academic professor. Create a COMPREHENSIVE study guide.

      **Analysis Requirements**:
      - Write a detailed structured analysis.
      - Use Markdown Headers (##, ###) to separate sections.
      - Use Bullet points for lists.
      - Use **Bold** for emphasis.
      - Use LaTeX for ALL math. WRAP ALL MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$.
      - Cover major concepts effectively.

      **Output Format**:
      - Return ONLY valid JSON matching the schema.
      - Do not add Markdown code blocks or text outside the JSON.` }
     ]
    },
    config: {
```

```
      responseMimeType: "application/json",
      responseSchema: analysisResponseSchema,
      temperature: 0.1,
    }
  }));

  // Request 2: Structured Data
  const dataPromise = withRetry<GenerateContentResponse>(() =>
ai.models.generateContent({
    model: model,
    contents: {
      parts: [
        contentPart,
        { text: `Extract structured study data from this document.

        **Rules**:
        - **Formulas**: Extract ALL mathematical formulas. Use LaTeX. WRAP MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$.
        - **CRITICAL**: Escape all backslashes in strings. e.g. "\\frac" must be "\\\\frac".
        - **Quiz**: Create 8 challenging MCQs.
        - **Theory**: Create 6 deep theory questions.
        - **Flashcards**: Create 15 key term/definition pairs.
        - **MindMap**: Create a deep hierarchical mind map structure starting from the main document topic.

        **Output Format**:
        - Return ONLY valid JSON matching the schema.
        - Do not add Markdown code blocks.` }
      ]
```

```
    },
    config: {
      responseMimeType: "application/json",
      responseSchema: structuredDataSchema,
      temperature: 0.1,
    }
  }));

  const [analysisResponse, dataResponse] = await Promise.all([analysisPromise,
dataPromise]);

  // Handle potentially missing text
  const analysisText = analysisResponse.text || "{}";
  const dataText = dataResponse.text || "{}";

  const analysisJson = safeParseJSON(analysisText);
  const dataJson = safeParseJSON(dataText);

  return {
    paperId: Math.random().toString(36).substring(7),
    analysis: analysisJson.analysis || "Analysis generation failed or was truncated.",
    keyTopics: analysisJson.keyTopics || [],
    formulas: dataJson.formulas || [],
    quiz: dataJson.quiz || [],
    theoryQuestions: dataJson.theoryQuestions || [],
    flashcards: dataJson.flashcards || [],
    mindMap: dataJson.mindMap || { label: "Root", children: [] }
  };
```

```
  } catch (error) {
    console.error("Error processing document:", error);
    throw error;
  }
};

// Replaces processPdf to support generic document processing
export const processPdf = processDocument;

export const getCurriculumTopics = async (grade: string, board: string, subject: string): Promise<string[]> => {
  try {
    const model = "gemini-2.5-flash";
    const response = await withRetry<GenerateContentResponse>(() => ai.models.generateContent({
      model: model,
      contents: {
        parts: [{
          text: `List the 8 most important chapter/topic names for Class/Grade: ${grade}, Board: ${board}, Subject: ${subject}.
          Return them as a simple JSON array of strings.`
        }]
      },
      config: {
        responseMimeType: "application/json",
        responseSchema: topicsSchema,
        temperature: 0.3
      }
```

```
  }));

  const json = safeParseJSON(response.text || "{}");
  return json.topics || [];
 } catch (error) {
  console.error("Error getting topics:", error);
  return [];
 }
};

export const generateTestPaper = async (params: GeneratorParams): Promise<{quiz: QuizQuestion[], theoryQuestions: TheoryQuestion[]}> => {
 try {
  const model = "gemini-2.5-flash";
  const contextPrompt = `Context: Class ${params.grade}, Board ${params.board}, Subject ${params.subject}, Topic: ${params.topic}.`;

  const mcqCount = params.mcqCount;
  const theoryCount = params.theoryCount;
  const difficulty = params.difficulty || "Variable (Mixed Difficulty)";

  // Adjust instructions based on source type
  const sourceInstruction = params.sourceType === 'NON_AI'
    ? "STRICTLY retrieve or simulate authentic past-paper style questions found in official exams for this curriculum. Avoid generic AI-generated questions. Format them exactly as they appear in standard exams."
    : "Generate high-quality, creative, and unique questions tailored to the specific topics. Focus on conceptual depth.";
```

```
  const response = await withRetry<GenerateContentResponse>(() =>
ai.models.generateContent({
    model: model,
    contents: {
     parts: [
      { text: `Create a test paper for: ${contextPrompt}.

      **Source Requirement**: ${sourceInstruction}

      **Requirements**:
      - Create exactly ${mcqCount} MCQs.
      - Create exactly ${theoryCount} Theory/Long Answer Questions.
      - **Theory**: Include a 'question', a short 'answer' (final result), and a detailed 'explanation' (method/steps) for each theory question.
      - **Difficulty Setting**: ${difficulty}. Adjust the complexity of questions accordingly.
      - **Formulas**: Use LaTeX. WRAP ALL MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$.
      - **CRITICAL**: Escape all backslashes in strings. e.g. "\\frac" must be "\\\\frac".

      **Validation**:
      - If the topic or subject is gibberish, unrelated to academics, or cannot be processed, return an empty JSON object with empty arrays for 'quiz' and 'theoryQuestions'.

      **Output Format**:
      - Return ONLY valid JSON matching the schema. ` }
     ]
    },
    config: {
```

```
      responseMimeType: "application/json",
      responseSchema: testPaperSchema,
      temperature: params.sourceType === 'NON_AI' ? 0.1 : 0.4, // Lower temp for factual past papers, higher for creative
    }
  }));

  if (!response.text) throw new Error("No response");

  const json = safeParseJSON(response.text);
  return {
    quiz: json.quiz || [],
    theoryQuestions: json.theoryQuestions || []
  };

 } catch (error) {
  console.error("Error generating test paper:", error);
  return { quiz: [], theoryQuestions: [] };
 }
};

export const generateDiagramForQuestion = async (question: string): Promise<string | null> => {
  try {
    const model = "gemini-2.5-flash";
    const response = await withRetry<GenerateContentResponse>(() => ai.models.generateContent({
      model: model,
      contents: {
```

```
            parts: [{ text: `Generate a valid SVG code string to visualize this academic problem.

            Question: "${question}"

            Requirements:
            - Return ONLY JSON with a single field 'diagramSVG'.
            - The SVG must be simple, clear, use black strokes, white/transparent fill.
            - Dimensions: ViewBox "0 0 400 300" or similar.
            - If the question DOES NOT need a diagram (e.g. pure algebra, definition), return empty string.` }]
        },
        config: {
          responseMimeType: "application/json",
          responseSchema: diagramSchema,
          temperature: 0.1
        }
      }));

      const json = safeParseJSON(response.text || "{}");
      return json.diagramSVG || null;
    } catch (error) {
      console.error("Diagram generation failed", error);
      return null;
    }
}

export const generateCurriculumStudyMaterial = async (params: GeneratorParams): Promise<StudyData> => {
```

```
  return processDocument(`Mock Context: ${JSON.stringify(params)}`, 'text/plain');
};

export const generateMoreQuestions = async (data: string, mimeType: string, mode: QuestionMode, existingCount: number, count: number = 5): Promise<{quiz?: QuizQuestion[], theoryQuestions?: TheoryQuestion[]}> => {
  try {
    const model = "gemini-2.5-flash";
    const contentPart = getContentPart(data, mimeType);

    let promptText = `Generate ${count} MORE distinct questions. Use LaTeX. WRAP ALL MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$. Double escape backslashes (\\) for JSON. Valid JSON only.`;

    if (mode === QuestionMode.MCQ) {
      promptText += ` Generate ${count} new MCQs. Start IDs from ${existingCount + 1}.`;
    } else {
      promptText += ` Generate ${count} new Theory Questions. Include answer and explanation for each.`;
    }

    const response = await withRetry<GenerateContentResponse>(() => ai.models.generateContent({
      model: model,
      contents: {
        parts: [
          contentPart,
          { text: promptText }
        ]
      },
```

```
    config: {
      responseMimeType: "application/json",
      responseSchema: moreQuestionsSchema,
      temperature: 0.2,
    }
  }));

  const text = response.text || "{}";
  return safeParseJSON(text);

 } catch (error) {
  console.error("Error generating more questions:", error);
  throw error;
 }
};

export const generateMoreFormulas = async (data: string, mimeType: string):
Promise<{formulas: Formula[]}> => {
  try {
    const model = "gemini-2.5-flash";
    const contentPart = getContentPart(data, mimeType);

    const promptText = `Extract 5 MORE unique mathematical formulas or methods.

    **Requirements**:
    - Use LaTeX. WRAP ALL MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$.
    - IMPORTANT: Escape backslashes for JSON strings (e.g. \\alpha becomes \\\\alpha).
    - Include usage method and section.
```

```
    - Output valid JSON only.` ;

    const response = await withRetry<GenerateContentResponse>(() =>
ai.models.generateContent({
      model: model,
      contents: {
        parts: [
          contentPart,
          { text: promptText }
        ]
      },
      config: {
        responseMimeType: "application/json",
        responseSchema: moreFormulasSchema,
        temperature: 0.2,
      }
    }));

    const text = response.text || "{}";
    return safeParseJSON(text);

  } catch (error) {
    console.error("Error generating more formulas:", error);
    throw error;
  }
};

export const askDocumentQuestion = async (data: string, mimeType: string, question:
string): Promise<string> => {
```

```
 try {
   const model = "gemini-2.5-flash";
   const contentPart = getContentPart(data, mimeType);

   const promptText = `Answer this question based on the document: "${question}"

   - Be concise but accurate.
   - Use Markdown.
   - Use LaTeX for math. WRAP MATH IN '$'.`;

   const response = await withRetry<GenerateContentResponse>(() =>
ai.models.generateContent({
    model: model,
    contents: {
     parts: [
      contentPart,
      { text: promptText }
     ]
    },
    config: {
     temperature: 0.1,
    }
   }));

   return response.text || "No response generated.";
 } catch (error) {
   console.error("Error asking question:", error);
   throw error;
```

```
 }
};

export const generateCustomQuiz = async (data: string, mimeType: string, config: QuizConfig): Promise<InteractiveQuestion[]> => {
 try {
  const model = "gemini-2.5-flash";
  const contentPart = getContentPart(data, mimeType);

  const promptText = `Create a custom quiz based on the document.

  **Configuration**:
  - Types: ${config.types.join(', ')}
  - Difficulty: ${config.difficulty}
  - Total Questions: ${config.count}

  **Requirements**:
  - **MULTIPLE_CHOICE**: Provide 'options' array and 'correctIndex'.
  - **FILL_IN_THE_BLANK**: The 'question' MUST contain '[____]' as the placeholder. Provide exact word in 'answer'.
  - **OPEN_ENDED**: Provide a detailed model answer in 'answer'.
  - **Keywords**: For FILL_IN_THE_BLANK and OPEN_ENDED, provide 3-5 acceptable keywords/synonyms in 'keywords' array to check the user's answer.
  - Include clear 'explanation' for all types.
  - Use LaTeX for math. WRAP ALL MATH IN '$' DELIMITERS. Example: $\\frac{1}{2}$.
  - Escape backslashes in JSON (e.g. \\frac -> \\\\frac).

  **Output**:
  - Return ONLY valid JSON matching the schema.`;
```

```
   const response = await withRetry<GenerateContentResponse>(() =>
ai.models.generateContent({
     model: model,
     contents: {
       parts: [
         contentPart,
         { text: promptText }
       ]
     },
     config: {
       responseMimeType: "application/json",
       responseSchema: customQuizSchema,
       temperature: 0.2,
     }
   }));

   const json = safeParseJSON(response.text || "{}");
   return json.questions || [];

  } catch (error) {
   console.error("Error generating custom quiz:", error);
   throw error;
  }
}
```

==========

PART 2: SPACED REPETITION LOGIC (srsService.ts)

This file implements the Spaced Repetition System (SM-2 Algorithm) to intelligently schedule reviews for flashcards and questions.

==========

```
import { ReviewItem, QuizType } from '../types';

const STORAGE_KEY = 'aceai_srs_deck';

export const getDeck = (): ReviewItem[] => {
 try {
  const data = localStorage.getItem(STORAGE_KEY);
  return data ? JSON.parse(data) : [];
 } catch (e) {
  console.error("Failed to load SRS deck", e);
  return [];
 }
};

export const saveDeck = (deck: ReviewItem[]) => {
 localStorage.setItem(STORAGE_KEY, JSON.stringify(deck));
};

export const addToDeck = (question: string, answer: string, explanation: string = '', type: QuizType | 'GENERAL' = 'GENERAL') => {
 const deck = getDeck();

 // Prevent exact duplicates
```

```
  if (deck.some(i => i.question === question)) {
    return false; // Already exists
  }

  const newItem: ReviewItem = {
   id: Date.now().toString() + Math.random().toString(36).substr(2, 5),
   question,
   answer,
   explanation,
   nextReviewDate: Date.now(), // Due immediately
   interval: 0,
   easeFactor: 2.5,
   repetitions: 0,
   type
  };

  deck.push(newItem);
  saveDeck(deck);
  return true;
};

export const getDueItems = (): ReviewItem[] => {
  const deck = getDeck();
  const now = Date.now();
  // Filter items where nextReviewDate is in the past or now
  return deck.filter(item => item.nextReviewDate <= now);
};
```

```
export const getReviewCount = (): number => {
  return getDueItems().length;
}

/**
 * Process a review based on user rating.
 * Uses a simplified SM-2 Algorithm.
 * rating: 'AGAIN' | 'HARD' | 'GOOD' | 'EASY'
 */
export const processReview = (itemId: string, rating: 'AGAIN' | 'HARD' | 'GOOD' | 'EASY')
=> {
 const deck = getDeck();
 const index = deck.findIndex(i => i.id === itemId);
 if (index === -1) return;

 const item = deck[index];

 // Map rating to SM-2 quality (0-5)
 // Again: 0 (Complete blackout)
 // Hard: 3 (Difficult response)
 // Good: 4 (Correct response after hesitation)
 // Easy: 5 (Perfect recall)
 let quality = 0;
 switch (rating) {
  case 'AGAIN': quality = 0; break;
  case 'HARD': quality = 3; break;
  case 'GOOD': quality = 4; break;
  case 'EASY': quality = 5; break;
```

```
}

// Algorithm Logic
if (quality < 3) {
  // If forgotten, reset repetitions and interval
  item.repetitions = 0;
  item.interval = 1; // Technically 1 day, but we might want 'AGAIN' to show sooner?
  // Standard SM-2 resets to 1 day.
} else {
  // If remembered
  if (item.repetitions === 0) {
     item.interval = 1;
  } else if (item.repetitions === 1) {
     item.interval = 6;
  } else {
     item.interval = Math.round(item.interval * item.easeFactor);
  }
  item.repetitions += 1;
}

// Update Ease Factor (EF)
// EF' = EF + (0.1 - (5 - q) * (0.08 + (5 - q) * 0.02))
// EF cannot go below 1.3
item.easeFactor = item.easeFactor + (0.1 - (5 - quality) * (0.08 + (5 - quality) * 0.02));
if (item.easeFactor < 1.3) item.easeFactor = 1.3;

// Calculate Next Review Date
const DAY_MS = 24 * 60 * 60 * 1000;
```

```
  if (rating === 'AGAIN') {
    // If user clicked AGAIN, we technically want them to review it again *soon*.
    // For this app, let's just set it to tomorrow to avoid getting stuck in a loop during a single session,
    // OR set it to 1 minute from now if we want same-session functionality.
    // Let's stick to "Daily Review" concept -> Review tomorrow.
    item.nextReviewDate = Date.now() + DAY_MS;
  } else {
    item.nextReviewDate = Date.now() + (item.interval * DAY_MS);
  }

  saveDeck(deck);
};
```