GIFT AI MVP - Technical Design Document

Introduction

This document outlines the technical design for the Minimum Viable Product (MVP) of the GIFT AI platform. The platform aims to help users discover and select curated gift recommendations for their friends and family based on personal preferences. This document covers system architecture, data models, technology stack recommendations, API design, integration points, and wireframes for key user flows.

GIFT AI MVP - System Architecture & Data Models

1. System Architecture

The GIFT AI MVP platform will adopt a modular, service-oriented architecture to ensure scalability and maintainability. The key components are:

1. **Frontend Application (Web App):**
    - The user interface where users register, manage profiles, add friends/family, input preferences, view recommendations, and click through to purchase products.
    - Built as a Single Page Application (SPA) for a responsive and interactive user experience.
2. **Backend API (RESTful API):**
    - The central nervous system of the platform, handling all business logic, data processing, and communication between the frontend and other backend services.
    - Exposes endpoints for user management, preference management, friend/family management, recommendation retrieval, and product interactions.

3. **Database:**
    - A relational or NoSQL database to store user data, friend/family profiles, preferences, product metadata, curated recommendations, and notification logs.
4. **Recommendation Engine (MVP - Rule-Based):**
    - A service or module responsible for generating gift recommendations.
    - For MVP, this will implement simple rule-based logic based on user/recipient preferences (demographics, interests from the quiz) and product metadata.
    - It will query the product metadata database and apply filtering rules.
5. **Product Data Ingestion Service:**
    - A background service or set of scripts responsible for populating and updating the product metadata database.
    - For MVP, this will support:
        - Manual entry/upload of curated product data.
        - Automated scraping of product information from specified e-commerce sites (e.g., Amazon, Etsy, eBay).
        - Integration with free-tier e-commerce APIs for product data retrieval.
6. **Notification Service:**
    - Responsible for sending email notifications to users about new recommendations.
    - Integrates with a third-party email service provider (e.g., SendGrid, Mailgun).

High-Level Interaction Flow:

- **User Onboarding & Preference Input:** User registers/logs in via Frontend -> Backend API stores user data in Database. User adds friends/family and their preferences (quiz results) via Frontend -> Backend API stores this in Database.
- **Product Data Population:** Product Data Ingestion Service scrapes/fetches product info -> stores in Product Metadata Database.
- **Recommendation Generation (Scheduled/Triggered):** Recommendation Engine (MVP) queries Database for user/recipient preferences and product metadata -> applies rules -> generates recommendations -> stores recommendations in Database.
- **Notification:** Recommendation Engine triggers Notification Service -> Notification Service sends email to user with a link to the dashboard.

- **Viewing Recommendations:** User clicks email link or logs into Frontend -> Frontend calls Backend API to fetch recommendations -> Backend API retrieves from Database and returns to Frontend.
- **Product Purchase:** User clicks "buy" link on Frontend -> Frontend redirects to external e-commerce site.

2. Data Models

Below are the core data entities for the MVP. Relationships and specific attributes will be further refined.

2.1. User

- `user_id` (Primary Key, UUID)
- `email` (String, Unique, Indexed)
- `password_hash` (String)
- `first_name` (String)
- `last_name` (String)
- `created_at` (Timestamp)
- `updated_at` (Timestamp)

2.2. Recipient (Friend/Family)

- `recipient_id` (Primary Key, UUID)
- `user_id` (Foreign Key to User, Indexed) - The user who added this recipient.
- `name` (String) - Name of the friend/family member.
- `relationship` (String, e.g., Friend, Spouse, Sibling, Parent, Child)
- `created_at` (Timestamp)
- `updated_at` (Timestamp)

2.3. Preference

- `preference_id` (Primary Key, UUID)
- `recipient_id` (Foreign Key to Recipient, Indexed)
- `preference_type` (String, e.g., "age_range", "interest", "occasion", "budget_min", "budget_max", "dislikes_categories")
- `preference_value` (String or JSON, flexible to store various preference data from the quiz)
  - Example for "interest": `["hiking", "reading", "tech gadgets"]`
  - Example for "age_range": `"30-40"`
- `created_at` (Timestamp)

- `updated_at` (Timestamp)

2.4. Product

- `product_id` (Primary Key, UUID)
- `name` (String)
- `description` (Text)
- `source_url` (String, URL to the product on the e-commerce site)
- `image_url` (String)
- `price` (Decimal)
- `currency` (String, e.g., "USD")
- `source_site` (String, e.g., "Amazon", "Etsy", "Manual")
- `categories` (Array of Strings or JSON, e.g., `["electronics", "gifts_for_him"]`)
- `tags` (Array of Strings or JSON, for more granular filtering)
- `metadata` (JSON, for other relevant product attributes like brand, material, ratings, etc.)
- `is_active` (Boolean, for managing scraped/API products)
- `last_scraped_at` (Timestamp, if applicable)
- `created_at` (Timestamp)
- `updated_at` (Timestamp)

2.5. Recommendation

- `recommendation_id` (Primary Key, UUID)
- `user_id` (Foreign Key to User, Indexed) - The user for whom the recommendation is made.
- `recipient_id` (Foreign Key to Recipient, Indexed) - The friend/family member the gift is for.
- `product_id` (Foreign Key to Product, Indexed)
- `recommendation_score` (Float, MVP might be simple, e.g., 1.0 for direct match)
- `reasoning` (Text, MVP might be simple, e.g., "Matches interest: hiking")
- `status` (String, e.g., "new", "viewed", "approved", "dismissed")
- `generated_at` (Timestamp)
- `notified_at` (Timestamp, when email was sent)

2.6. NotificationLog

- `notification_id` (Primary Key, UUID)
- `user_id` (Foreign Key to User, Indexed)
- `type` (String, e.g., "new_recommendation_email")
- `status` (String, e.g., "sent", "failed")

- `sent_at` (Timestamp)
- `error_message` (Text, if failed)

This initial design for architecture and data models provides a foundation. The next step will be to recommend a technology stack and tools suitable for building these components.

3. Technology Stack & Tool Recommendations

Choosing the right technology stack is crucial for the MVP's success, balancing rapid development, scalability, cost-effectiveness, and the ability to integrate future AI/ML capabilities. The following recommendations are tailored to the GIFT AI platform's architecture and MVP requirements.

3.1. Frontend Application (Web App)

- **Framework/Library: React.js** or **Next.js**
  - **React.js:** A popular and robust library for building SPAs with a vast ecosystem of components and tools. Offers great flexibility.
  - **Next.js:** A framework built on top of React, offering features like server-side rendering (SSR), static site generation (SSG), routing, and API routes out-of-the-box. This can be beneficial for SEO (though less critical for a dashboard-heavy app initially) and provides a more structured development experience. For an MVP that includes a user dashboard and aims for good performance, Next.js is a strong contender.
  - **Recommendation: Next.js** for its comprehensive features, performance benefits, and ease of creating API routes if needed for simple backend-for-frontend (BFF) patterns, though the main backend API will be separate.
- **State Management:**
  - **Zustand** or **Redux Toolkit (RTK):** Zustand is lightweight and simple for managing global state. RTK is more powerful and structured, suitable if complex state logic is anticipated early on.
  - **Recommendation:** Start with **React Context API** for simpler global state (e.g., user authentication) and consider **Zustand** if more complex global state management becomes necessary. This keeps the initial setup lean.
- **UI Components:**
  - **Shadcn/ui** or **Material-UI (MUI)** or **Tailwind CSS with Headless UI**.
  - **Shadcn/ui:** Provides beautifully designed, accessible components that you copy and paste into your project, built on Tailwind CSS and Radix UI. Highly customizable.

    - **MUI:** Offers a comprehensive suite of pre-built React components following Material Design.
    - **Tailwind CSS:** A utility-first CSS framework for rapid UI development. Can be combined with Headless UI for accessible, unstyled components.
    - **Recommendation: Shadcn/ui** for its modern design, accessibility, and customizability, leveraging Tailwind CSS.
- **Styling: Tailwind CSS** (if not using a component library that dictates styling, or to complement it).
- **Form Handling: React Hook Form** with **Zod** for validation (as seen in your previous contact form example, this is a good choice).
- **Data Fetching: TanStack Query (React Query)** for managing server state, caching, and background updates.

3.2. Backend API (RESTful API)

- **Language/Framework:**
    - **Node.js with Express.js/NestJS:** JavaScript/TypeScript on the backend. Express.js is minimal and flexible. NestJS is a more opinionated, full-featured framework built with TypeScript, providing structure and scalability (good for long-term).
    - **Python with Django/FastAPI:** Python is excellent for data science and AI/ML, making future integration smoother. Django is full-featured. FastAPI is modern, high-performance, and easy to learn.
    - **Recommendation: Node.js with NestJS (TypeScript)**. It offers a good balance of structure, performance, a strong TypeScript ecosystem (aligning well with a potential Next.js frontend), and is well-suited for building robust APIs. Python with FastAPI is a strong alternative if the team has stronger Python expertise or wants to prioritize AI/ML integration from day one.

3.3. Database

- **Type:** Relational (PostgreSQL, MySQL) or NoSQL (MongoDB, DynamoDB).
    - The data model has clear relationships (User-Recipient, Recipient-Preference, Recommendation-Product), which lends itself well to a relational database.
    - However, `Preference.preference_value` and `Product.metadata` are designed to be flexible (JSON), which NoSQL handles natively, but modern relational databases also support JSON types well.
- **Specific Choice:**
    - **PostgreSQL:** A powerful open-source relational database with excellent support for JSONB, full-text search, and scalability. It can handle both structured and semi-structured data effectively.

    - **MongoDB:** A popular NoSQL document database, good for flexible schemas and rapid development, especially if the data structure is expected to evolve significantly.
- **Recommendation: PostgreSQL**. Its relational strengths combined with robust JSONB support make it a versatile choice for the defined data models and future growth. It provides strong data integrity.
- **ORM/Query Builder (for Backend):**
    - If using NestJS (Node.js): **TypeORM** or **Prisma**.
    - If using Python: **SQLAlchemy** (for Django/FastAPI) or **Prisma Client Python**.
    - **Recommendation (with NestJS): Prisma** for its type safety, auto-generated client, and ease of use.

3.4. Recommendation Engine (MVP - Rule-Based)

- **Implementation:** This will likely be a module within the **Backend API** for the MVP.
- **Logic:** Implemented in the backend language (e.g., TypeScript if using NestJS). It will involve querying the PostgreSQL database for preferences and product metadata and applying filtering logic.

3.5. Product Data Ingestion Service

- **Scraping:**
    - **Python with libraries like Scrapy, Beautiful Soup, Playwright/Puppeteer:** Python is very strong for web scraping.
    - **Recommendation: Python with Playwright or Puppeteer** for handling modern JavaScript-heavy websites. Scrapy is good for larger-scale, structured scraping.
- **API Integration:** Standard HTTP client libraries in the chosen backend language (e.g., `axios` for Node.js, `requests` for Python).
- **Scheduling:** Can be managed by cron jobs, a simple scheduler library, or a serverless function (e.g., AWS Lambda scheduled event).
- **Recommendation:** A separate **Python service/scripts** for scraping due to Python's excellent scraping libraries. This service can then push data to the main database or expose an internal API for the backend to consume.

3.6. Notification Service

- **Email Service Provider:**
    - **SendGrid, Mailgun, Amazon SES.** All offer robust APIs, good deliverability, and free tiers for MVPs.

    - **Recommendation: SendGrid** or **Amazon SES**. SendGrid is often praised for its ease of use and developer-friendly API. SES is cost-effective at scale.
- **Integration:** Via their SDKs in the backend language.

3.7. Hosting & Deployment

- **Frontend (Next.js):**
    - **Vercel:** Optimized for Next.js, offers seamless deployment, CI/CD, serverless functions, and a generous free tier.
    - **AWS Amplify Hosting:** Good for SPAs and Next.js apps, integrates with other AWS services.
    - **Netlify:** Similar to Vercel, excellent for static sites and SPAs.
    - **Recommendation: Vercel** for its superior Next.js support and developer experience.
- **Backend API (NestJS/Node.js or Python/FastAPI):**
    - **Serverless Functions:** AWS Lambda, Google Cloud Functions, Azure Functions. Cost-effective, auto-scaling.
    - **Containerization:** Docker with services like AWS Fargate, Google Cloud Run, or managed Kubernetes (e.g., Amazon EKS, Google GKE).
    - **PaaS:** Heroku, Render, Railway. Easier to manage than IaaS or containers but can be more expensive at scale.
    - **Recommendation (MVP): Serverless Functions (e.g., AWS Lambda with API Gateway)** for the backend API. This is cost-effective for starting, scales automatically, and reduces operational overhead. Alternatively, **Render** or **Railway** for a PaaS solution if a more traditional server setup is preferred with less AWS-specific lock-in.
- **Database (PostgreSQL):**
    - **Managed Database Services:** Amazon RDS for PostgreSQL, Google Cloud SQL for PostgreSQL, Supabase (provides PostgreSQL + auth + APIs), Neon (serverless PostgreSQL).
    - **Recommendation: Amazon RDS for PostgreSQL** if already in the AWS ecosystem, or **Supabase** if looking for a BaaS (Backend-as-a-Service) feel that includes a managed PostgreSQL database, authentication, and auto-generated APIs, which can speed up MVP development significantly.
- **Product Data Ingestion Service (Python Scripts):**
    - Can be run as scheduled **AWS Lambda functions** or on a small **EC2 instance/VPS** if long-running scraping tasks are needed.

3.8. Other Tools

- **Version Control: Git** (with GitHub, GitLab, or Bitbucket).

- **Project Management:** Jira, Trello, Asana, Notion.
- **API Design & Testing:** Postman, Insomnia.
- **Logging & Monitoring:** Sentry (for frontend/backend errors), CloudWatch (if using AWS), Prometheus/Grafana (more advanced).

This stack prioritizes modern technologies, developer productivity for MVP, and scalability for future enhancements, including more complex AI features.

4. API Design & Integration Points

This section outlines the RESTful API endpoints for the GIFT AI MVP backend, along with key integration points with other services. The API will serve as the primary interface for the frontend application.

Base URL: `/api/v1`

Authentication: For MVP, consider a simple JWT (JSON Web Token) based authentication for protected routes. Unprotected routes include user registration and login.

4.1. User Management Endpoints

- **`POST /auth/register`**
  - **Description:** Registers a new user.
  - **Request Body:** `json { "email": "user@example.com", "password": "string_password", "firstName": "John", "lastName": "Doe" }`
  - **Response Body (Success 201):** `json { "userId": "uuid", "email": "user@example.com", "firstName": "John", "lastName": "Doe", "token": "jwt_token" }`
  - **Response Body (Error 400/409):** Standard error object.
- **`POST /auth/login`**
  - **Description:** Logs in an existing user.
  - **Request Body:** `json { "email": "user@example.com", "password": "string_password" }`
  - **Response Body (Success 200):** `json { "userId": "uuid", "email": "user@example.com", "token": "jwt_token" }`
  - **Response Body (Error 400/401):** Standard error object.

- **`GET /users/me`** (Protected Route)
    - **Description:** Retrieves the profile of the currently authenticated user.
    - **Response Body (Success 200):** User object (excluding password).

4.2. Recipient (Friend/Family) Management Endpoints (Protected Routes)

- **`POST /recipients`**
    - **Description:** Adds a new friend/family member for the authenticated user.
    - **Request Body:** `json { "name": "Jane Doe", "relationship": "Spouse" }`
    - **Response Body (Success 201):** Recipient object.
- **`GET /recipients`**
    - **Description:** Lists all friends/family members for the authenticated user.
    - **Response Body (Success 200):** Array of Recipient objects.
- **`GET /recipients/{recipientId}`**
    - **Description:** Retrieves details of a specific friend/family member.
    - **Response Body (Success 200):** Recipient object.
- **`PUT /recipients/{recipientId}`**
    - **Description:** Updates details of a specific friend/family member.
    - **Request Body:** Partial Recipient object.
    - **Response Body (Success 200):** Updated Recipient object.
- **`DELETE /recipients/{recipientId}`**
    - **Description:** Deletes a specific friend/family member.
    - **Response Body (Success 204):** No content.

4.3. Preference Management Endpoints (Protected Routes)

- **`POST /recipients/{recipientId}/preferences`**
    - **Description:** Adds or updates preferences for a specific recipient (quiz results).
    - **Request Body:** Array of preference objects or a structured quiz result object. `json // Example: Array of preference objects [ { "preferenceType": "age_range", "preferenceValue": "30-40" }, { "preferenceType": "interest",`

`"preferenceValue": ["hiking", "tech"] } ] // Or a single object representing all quiz answers // { "ageRange": "30-40", "interests": ["hiking", "tech"], ... }`
    - **Response Body (Success 201 or 200):** Array of created/updated Preference objects or a confirmation.

- **`GET /recipients/{recipientId}/preferences`**
    - **Description:** Retrieves all preferences for a specific recipient.
    - **Response Body (Success 200):** Array of Preference objects.

4.4. Product & Recommendation Endpoints (Protected Routes)

- **`GET /recommendations`**
    - **Description:** Retrieves gift recommendations for the authenticated user (across all their recipients) or for a specific recipient if `recipientId` query parameter is provided.
    - **Query Parameters:** `?recipientId=uuid` (optional)
    - **Response Body (Success 200):** Array of Recommendation objects, potentially grouped by recipient, including associated Product details. `json [ { "recommendationId": "uuid", "recipient": { "recipientId": "uuid", "name": "Jane Doe" }, "product": { "productId": "uuid", "name": "Awesome Gift", "sourceUrl": "...", "imageUrl": "..." }, "reasoning": "Matches interest: tech", "status": "new" } ]`

- **`POST /recommendations/{recommendationId}/action`**
    - **Description:** Allows the user to act on a recommendation (e.g., approve, dismiss).
    - **Request Body:** `json { "action": "approved" // or "dismissed" }`
    - **Response Body (Success 200):** Updated Recommendation object.

- **Internal Endpoint (Not directly user-facing, for admin/curation):**
    - **`POST /products`**: For manual product data entry.
    - **`PUT /products/{productId}`**: For updating manually entered products.

4.5. Integration Points

1. **Product Data Ingestion Service & Backend API:**
    - **Method:** The Product Data Ingestion Service (Python scripts) will directly write to the shared **PostgreSQL Database** after scraping or fetching data from e-commerce APIs.

    - **Alternatively:** The ingestion service could expose a secure internal API endpoint that the main Backend API calls to trigger ingestion or receive data, but direct DB write is simpler for MVP if the service is trusted and runs in the same VPC/network.

2. **Recommendation Engine & Backend API:**
    - **Method (MVP):** The rule-based recommendation logic will be a module within the Backend API. It will be triggered periodically (e.g., by a scheduled job like a cron calling a specific internal API endpoint or a direct function call within the backend service) or on-demand (e.g., after new preferences are added).
    - **Data Flow:** Reads `User`, `Recipient`, `Preference`, and `Product` data from the Database. Writes generated `Recommendation` data back to the Database.

3. **Notification Service & Backend API:**
    - **Trigger:** After new recommendations are generated and stored, the Recommendation Engine (or a subsequent process in the Backend API) will trigger the Notification Service.
    - **Method:** The Backend API will make an API call to the chosen email service provider (e.g., SendGrid, Amazon SES) using their SDK.
    - **Payload:** User email, recipient name, link to the dashboard/recommendations page.
    - The Backend API will also log notification attempts and status in the `NotificationLog` table.

4. **Frontend & External E-commerce Sites:**
    - **Method:** Simple URL redirection. When a user clicks a "buy" link for a recommended product, the frontend will open the `product.source_url` in a new tab or redirect the current tab.

4.6. Error Handling

- APIs should use standard HTTP status codes (e.g., 200, 201, 400, 401, 403, 404, 500).
- Error responses should have a consistent JSON format: `json { "error": { "message": "A descriptive error message", "code": "ERROR_CODE_SLUG" // Optional error code } }`

This API design provides a starting point for the MVP. Endpoints and payloads can be refined as development progresses.

5. Wireframes for Key User Flows

This section provides low-fidelity wireframes for the core user interactions within the GIFT AI MVP platform.

5.1. User Onboarding (Sign Up & Log In)

Users will be able to sign up for a new account or log in to an existing account.

User Onboarding Wireframe

5.2. Adding Recipient & Preference Quiz

Users can add friends or family members and then complete a short preference quiz for each to help tailor gift recommendations.

Add Recipient and Preference Quiz Wireframe

5.3. Recommendations Dashboard

Users can view a dashboard of curated gift recommendations. They can filter recommendations by the specific friend or family member.

Recommendations Dashboard Wireframe

5.4. Email Notification

Users will receive email notifications when new gift recommendations are available for their recipients.

Email Notification Wireframe

5.5. Product Purchase Flow (Redirect)

When a user decides to purchase a recommended gift, they will be redirected to the external e-commerce site to complete the transaction.

Product Purchase Redirect Wireframe

6. Next Steps & Future Considerations

This technical design document provides a blueprint for the GIFT AI MVP. Key next steps include:

- **Detailed Task Breakdown:** Creating a detailed backlog of development tasks based on this design.
- **Prototyping & UI/UX Design:** Developing higher-fidelity mockups and interactive prototypes based on these wireframes.
- **Development Sprints:** Iterative development of the platform components.
- **Testing:** Implementing unit, integration, and user acceptance testing.

Future considerations beyond the MVP include: * **Advanced AI/ML Recommendation Engine:** Incorporating machine learning models to analyze broader data sets (e.g., social media trends, purchase history if available) for more personalized and predictive recommendations. * **Expanded Integrations:** Integrating with more e-commerce platforms, potentially for in-app purchasing or affiliate tracking. * **Social Features:** Allowing users to share wishlists or gift ideas. * **Automated Gift Messaging & Scheduling:** AI-curated gift messages and options to schedule gift purchases or reminders. * **Mobile Application:** Developing native mobile applications for iOS and Android.

This document should serve as a living guide and be updated as the project evolves.