Week 1- Foundations

Week 1 - System Architecture

Section 1 – Entity List

User

- name
- email
- location
- style_preference
- onboarding_date

WardrobeItem

- type
- color
- formality
- season
- weather_appropriate
- image_url

Outfit

- date
- weather_context
- items_included
- user_rating
- occasion

StyleProfile

- formality_preference
- color_preferences
- fit_preferences
- preferred_aesthetic

WeatherData

- date
- temperature
- precipitation
- wind
- weather_type

Section 2 - Why these entities exist

User

▼ Why this exists

Stores core identity and style preferences for each person using DressUp.

Needed for personalisation, weather alignment, and user-specific recommendations.

WardrobeItem

▼ Why this exists

Tracks every clothing item the user owns.
Needed so the recommendation engine only uses available items.

Outfit

▼ Why this exists

Represents a single recommended outfit.
Stores weather context, rating, and included items for feedback loops.

StyleProfile

▼ Why this exists

Captures deeper preferences that go beyond single outfits.
Enables consistent styling over time.

WeatherData

▼ Why this exists

Stores daily weather conditions.
Required to generate practical, weather-suitable outfits.

Section 3 – Entity Relationship Diagram (ERD v1)

ERD v1 – Final Diagram

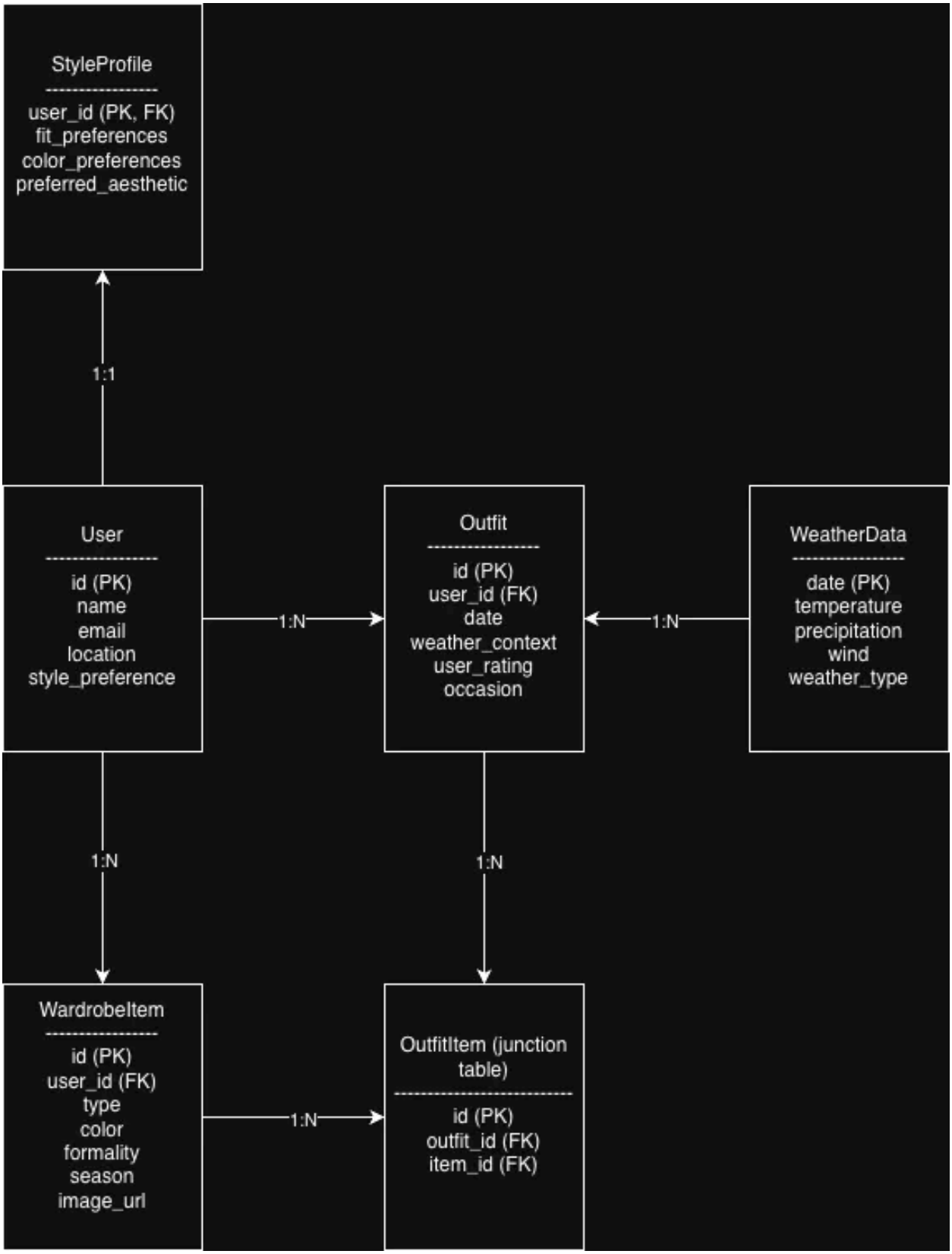
StyleProfile
user_id (PK, FK)
fit_preferences
color_preferences
preferred_aesthetic
1:1
User
id (PK)
name
email
location
style_preference
1:N
Outfit
id (PK)
user_id (FK)
date
weather_context
user_rating
occasion
1:N
WeatherData
date (PK)
temperature
precipitation
wind
weather_type
1:N
1:N
WardrobeItem
id (PK)
user_id (FK)
type
color
formality
season
image_url
1:N
OutfitItem (junction table)
id (PK)
outfit_id (FK)
item_id (FK)

Figure 1: Entity Relationship Diagram (ERD v1) showing the core data model for DressUp, including Users, Wardrobe Items, Outfits, Style Profiles, Weather Data, and the OutfitItem junction table. The diagram establishes all primary relationships required for structured outfit recommendation logic.

Section 4 - DFD v1 – Level 0

Context Diagram

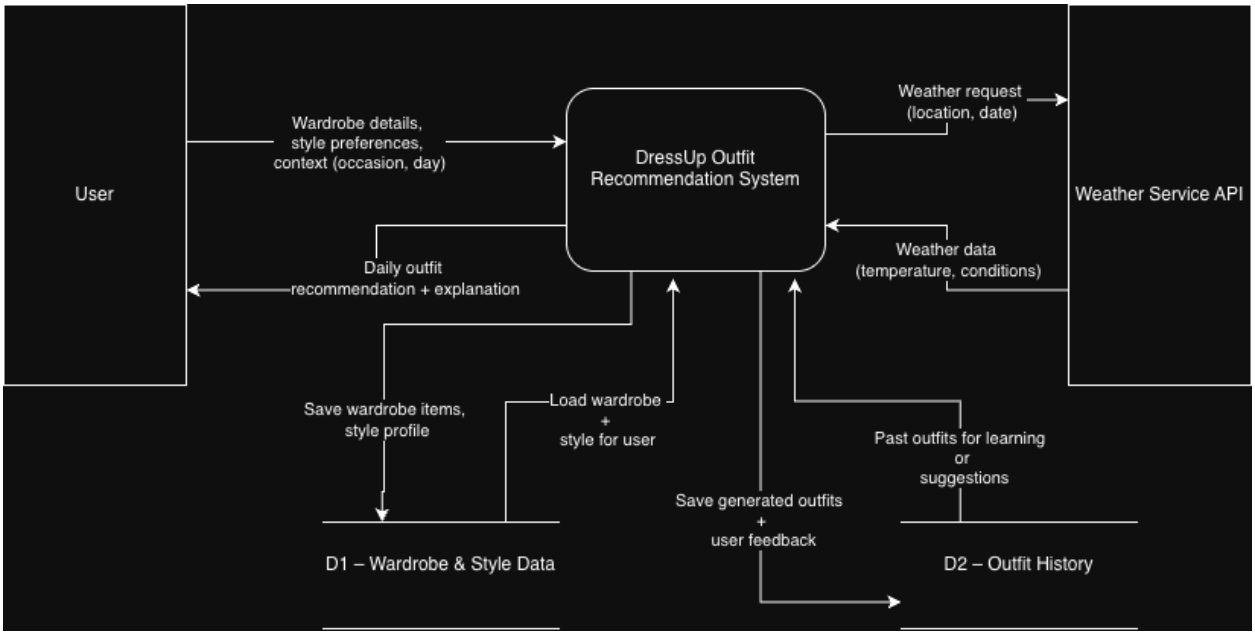

Figure 2: Data Flow Diagram Level 0 illustrating the high-level system context of DressUp. This diagram shows interactions between the User, the Weather Service API, the DressUp Recommendation System, and the core data stores for wardrobe, style data, and outfit history.

Section 5 – DFD v1 – Level 1 – Internal Processes

▼ **Purpose**

DFD Level 1 breaks down the high-level DressUp Outfit Recommendation System into four internal processes. This diagram shows how data flows between the user, the weather API, internal logic, and data stores. It provides a detailed view of how DressUp operates behind the scenes and supports the MVP architecture.

DFD Level 1 – Internal Processes

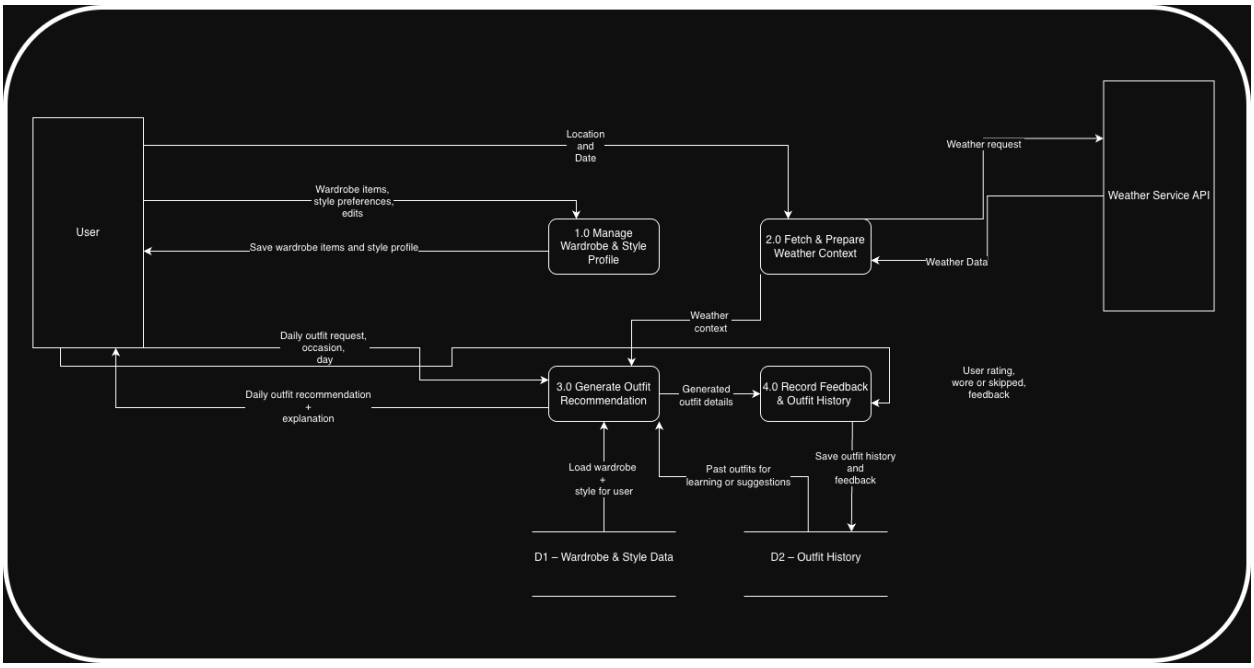

Figure 4: Data Flow Diagram Level 1 decomposing the DressUp Recommendation System into four internal processes: Manage Wardrobe and Style Profile, Fetch and Prepare Weather Context, Generate Outfit Recommendation, and Record Feedback and Outfit History. This diagram reveals the detailed operational workflow behind DressUp.

Process Breakdown

▼ 1.0 Manage Wardrobe and Style Profile

Handles user wardrobe items, wardrobe updates, and style preferences.

Saves structured wardrobe and style data into D1 for reliable access during recommendation generation.

▼ **2.0 Fetch and Prepare Weather Context**

Retrieves daily weather data from the Weather Service API.
Normalizes the raw weather information into a structured format that can be used by the recommendation engine.

▼ **3.0 Generate Outfit Recommendation**

Combines wardrobe data, user style preferences, weather context, and past outfit history to generate an optimized daily outfit along with an explanation.
This is the central decision making process within DressUp.

▼ **4.0 Record Feedback and Outfit History**

Stores the generated outfit along with user ratings, feedback, and wearing behavior into D2.
Creates the feedback loop required for future improvements and system learning.

Data Stores

▼ **D1 — Wardrobe and Style Data**

Contains structured wardrobe items and style preferences collected from the user.

Used by Processes 1.0 and 3.0.

▼ **D2 — Outfit History**

Holds generated outfit records, user ratings, and behavior signals.

Used by Processes 3.0 and 4.0.

Key Insights

- The single Level 0 process is decomposed into four smaller, logical processes which clarify system behavior.

- Weather processing is isolated in Process 2.0 for cleaner architecture and easier maintenance.
- Wardrobe and history data are separated into dedicated stores for accuracy and scalability.
- The diagram reveals a clear learning loop through Process 4.0 and Data Store D2.
- This structure directly supports the High Level System Architecture and ADR decisions in later sections.

Section 6 – High-Level System Architecture

System Architecture v1 – Final Diagram

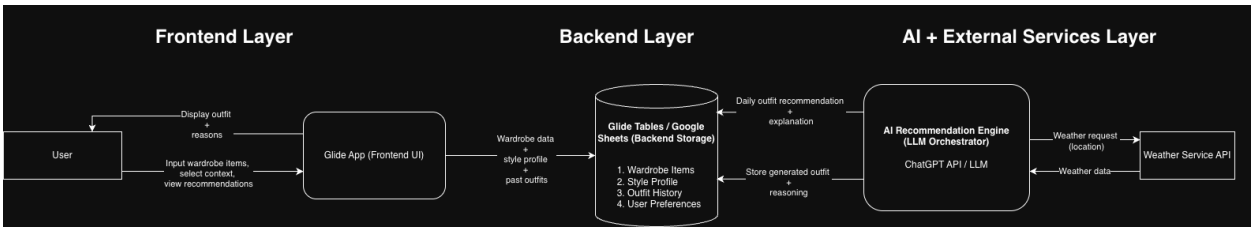

Figure 3: High-Level System Architecture for DressUp, displaying the three-layer structure consisting of the Frontend Layer (Glide App), Backend Layer (Glide Tables and Google Sheets), and the AI plus External Services Layer (LLM engine and Weather API). This establishes the MVP technical stack and integration points.

Section 7 – Architecture Decision Record (ADR v1)

ADR 001 — DressUp System Architecture v1

▼ **What is an ADR & why do we use it**

ADRs help track technical decisions, alternatives considered, and consequences.
They enable teams to revisit choices, understand the reasoning behind them, and intentionally evolve their architecture.

Status

Accepted - Week 1

Date

24 Nov 2025

Context

DressUp is an AI-powered outfit recommendation app that must:

- Be buildable by a solo founder without heavy coding
- Ship an MVP fast while still looking like a serious product
- Store structured wardrobe, style, and history data that LLMs can reliably use
- Integrate external weather data
- Be cheap or free to run during experimentation
- Be easy to extend later if the idea validates

You are working on a 2021 M1 MacBook Air with limited time and no desire to maintain servers or complex DevOps. The primary goal for this phase is to learn modern AI and no-code architecture and produce a portfolio-grade system, rather than optimising for massive scale.

Decision

1. **Frontend**

   Use **Glide App** as the primary user interface for DressUp.

   - Users will input wardrobe items, context, and see outfit recommendations inside Glide.

2. **Backend and Data Storage**

   Use **Glide Tables / Google Sheets** as the main backend storage layer.

- Core logical tables: User, WardrobeItem, Outfit, StyleProfile, WeatherData, OutfitItem (junction table).
- Data is modeled relationally with clear primary keys and foreign keys.

3. **AI Layer**

   Use a **hosted LLM (ChatGPT API or equivalent)** as the AI Recommendation Engine.

   - Glide or an automation layer will orchestrate calls to the LLM using structured prompts that pull from the backend tables.
   - No custom model training in v1.

4. **External Data**

   Use a standard **Weather Service API** (for example OpenWeather) to fetch daily weather context.

   - Weather data is not stored long term beyond what is required for outfits and history.

5. **Architecture Style**

   Adopt a **three layer architecture**:

   - Frontend layer: User, Glide App
   - Backend layer: Glide Tables / Google Sheets
   - AI plus External Services layer: LLM Orchestrator, Weather API

     Data flows are documented with ERD v1, DFD Level 0, and the High Level System Architecture diagram.

Rationale

Why Glide instead of a custom mobile or web app

- Much faster iteration for a solo builder
- Native support for lists, filters, relations, and basic logic
- No need to manage deployment, hosting, or app store pipelines

- Lets you focus on learning AI, data modeling, and product thinking rather than boilerplate UI code

Why Glide Tables / Google Sheets instead of a full database

- Spreadsheet like editing makes schema changes and test data easy
- Native integration with Glide reduces glue code and sync issues
- Good enough performance for an MVP and small user base
- Easy to export or migrate to a real database later if needed

Why a hosted LLM instead of building your own model

- You get state of the art language modeling without ML infrastructure
- You can iterate on prompt design and system behavior instead of training and tuning
- Cost scales with usage and can start near zero for development
- Perfect for the learning goals of this project: prompt engineering, chaining, and orchestration

Why an external Weather API

- Offloads the complexity of collecting and maintaining weather data
- Standard, well documented APIs already exist
- Keeps DressUp focused on its core value: style and personalization

Why relational modelling with a junction table

- Normalized entity design avoids duplication and inconsistency
- Many to many relationships between Outfit and WardrobeItem are handled cleanly via the OutfitItem junction table
- This structure maps well to any future migration to SQL, Airtable, or a more complex backend

Why a layered architecture

- Separates concerns: UI, data persistence, and AI logic are not tangled
- Makes it easier to swap components later

    - Replace Glide with a custom app
    - Replace Sheets with a database
    - Replace one LLM provider with another
- Easier to reason about data flow and debug issues

Alternatives Considered

1. **Custom React Native or web app**
    - Rejected for now because of higher build and maintenance cost, slower learning loop, and no immediate need for that level of control.
2. **Direct LLM only solution with no structured backend**
    - Rejected because wardrobe and history data would be hard to control, audit, and reuse.
    - Structured tables give better reliability and easier debugging.
3. **Full backend with Postgres plus server hosted API**
    - Rejected for MVP because infrastructure and DevOps overhead do not match current stage or goals.
    - Might be revisited if DressUp grows beyond Glide limitations.
4. **No weather integration in v1**
    - Rejected because weather awareness is a key differentiator for the app experience and is cheap to add through an API.

Consequences

Positive

- You can build a real, working DressUp prototype with minimal code.
- Architecture is clear enough that another engineer could join and understand it from the diagrams plus ADR alone.

- You can focus Week 1 to Week 3 on entity design, prompts, and flows rather than infrastructure.
- Migrating to a more advanced stack later is straightforward because the data model and flows are already designed cleanly.

Negative / Tradeoffs

- You are partially locked into Glide constraints for UI and logic in the short term.
- Glide Tables or Sheets may not scale to very large datasets or heavy concurrent usage.
- Latency and reliability depend on third party services: LLM provider and weather API.
- Some advanced behaviours will require creative use of Glide actions, automations, or an extra orchestration layer.

Future Changes and Open Questions

- At what traffic or user count would you need to migrate from Glide Tables to a dedicated database.
- Whether to introduce a separate orchestration layer later (for example a small backend service that handles LLM prompts, caching, and logging).
- How feedback loops from Outfit History will be used to improve prompts or add lightweight scoring logic.
- Which LLM provider and pricing tier will be best once you have real usage.

Prompt Engineering

🤖 Prompt Library – DressUp v1