# Design, Analysis and Implementation Decisions for CineFlow

## Initial Design Thoughts

CineFlow was designed as a modern, interactive movie discovery platform powered by a Django REST API and a React/Vite frontend — my first time building a project with this stack. The goal was to combine a cinematic look with smooth, app-like usability. The concept evolved around mood-based discovery: allowing users to browse movies that match a particular mood or emotion.

The interface uses glassmorphism, motion effects, and subtle gradients to create a sleek, immersive experience. The design had to stay consistent between the Django backend’s API structure and React’s dynamic frontend, ensuring that state management and data fetching felt seamless to the user.

The UI was built with reactivity and clarity in mind — light but expressive, and functional on both desktop and mobile screens.

## Home Page

The home page introduces CineFlow’s core idea through a bold hero section and integrated search bar. Users can search movies in real time with instant dropdown results that display as they type. To improve visibility, the search dropdown was given a higher stacking context (z-index) so it appears above call-to-action and movie cards.

CTAs below the hero section guide users to explore trending, now-playing, or mood-based movie collections. Each section is designed to flow visually, giving the experience of scrolling through a streaming service rather than a static site.

## Mood Dashboard

The Mood Dashboard is the central feature of CineFlow — an intelligent movie discovery page that re-ranks TMDB results by user “mood” categories like Feel-Good, Dark & Gritty, or Chill.

It supports pagination and infinite scroll, fetching from the /api/movies/mood/<mood\_key>/ endpoint. The layout dynamically adjusts to the viewport height to fill the user’s screen.

Provider filters (e.g., Netflix, Disney+) were initially implemented but later removed due to persistent backend integration issues with TMDB’s provider endpoints. The design still accommodates future reinstatement of filters when stable data sources are available.

User-rating controls were planned for this section but couldn’t be completed because rating data conflicted with the mood-ranking algorithm. This feature remains a potential enhancement.

## User Profile

The profile page allows users to update their details and upload an avatar. It connects directly to the Django REST API’s /api/me/profile/ endpoint.

Styling was kept minimal with “glass” panels and form elements consistent with the site’s overall theme. File uploads use Axios FormData requests, providing immediate local previews before saving. Error handling was refined to display clear feedback for invalid form fields and API errors.

An email verification banner and 2FA toggle were included to improve user trust and security awareness.

## Authentication & 2FA

CineFlow uses JWT authentication (via rest\_framework\_simplejwt) for secure token handling between frontend and backend. Email-based two-factor authentication (2FA) was integrated to add an extra layer of security. When 2FA is enabled, users receive a 6-digit code by email upon login, which must be verified before access tokens are issued.

A custom login flow was built in React to handle this: if an OTP is required, the user is prompted again without losing session context.

## Design Features

* Frontend: React + Vite (first project using this setup). Component structure was optimized for reusability and speed.
* - Backend: Django REST Framework providing secure, structured API endpoints.
* - Styling: Bootstrap CSS and custom stylesheets with a focus on gradients, blur effects, and consistent spacing.
* - Responsive Layout: The app adapts to different devices using flexible grid systems and dynamic container widths.
* - Data Integration: Real-time movie data fetched from TMDB API with caching on the server side.
* Deployment: Both frontend and backend deployed on Render, configured for cross-origin compatibility via CORS middleware.
* - Error Handling: Styled 404/500 error responses for both frontend and backend.

## My Development Process

I began by designing the layout in Canva and building static components in React. Once the UI framewas complete, I linked it to the Django API using Axios and JWT authentication.

As this was my first React/Vite project, one of the biggest learning curves was understanding async state handling and managing API calls efficiently. I also had to learn to work with FormData and CORS, especially for file uploads and third-party image requests.

The project was tested iteratively, both locally and in production, with fixes deployed continuously through Render. Each issue (like token refresh logic, avatar CORS errors, or provider filter bugs) led to a cleaner, more stable codebase.

## Challenges Faced

* Provider Filters: The TMDB provider API returned inconsistent data, so the filters were removed for stability.
* - User Ratings: Mood re-ranking logic conflicted with rating updates, so user ratings were postponed.
* - CORS Issues:TMDB image URLs blocked frontend previews — solved by using a Django proxy endpoint (/api/movies/poster\_palette/).  
  - Token Expiry Handling: Ensuring JWT refresh logic worked seamlessly without redirect loops.  
  - Render Deployment: Syncing frontend and backend origins while keeping authentication headers intact required repeated CORS debugging.

## Future Improvements

- Reintroduce provider filters once API consistency improves.  
- Implement local movie rating and review features.  
- Expand the profile system to allow public user pages.  
- Introduce caching and prefetching for smoother transitions.  
- Add accessibility refinements and dark/light theme switching.