Client Conversation: Movie and TV Review Platform

> **Project:** Full stack web application for a movie and TV review community
> **Prepared:** April 2026
> **Parties:** Client (Product), Buddy (Technical Lead)

The Problem

Client: We've seen an explosion of movie and TV content over the past few years. Streaming services, theaters, limited series, reboots. People want to find quality shows and movies but there's decision fatigue. IMDb exists, but we want to build something simpler and more community focused. Where users can quickly rate and review what they watch, then discover new recommendations from people with similar taste.

The core problem is this: How do we help users discover content through peer recommendations?

Buddy: I'm hearing two main pain points here. First, users need a way to record and share what they've watched and what they thought about it. Second, the product needs to aggregate those opinions so others can browse and discover.

That's a classic backend API plus frontend app situation. The backend stores reviews and ratings and serves search results. The frontend is the interface people actually use.

Client: Exactly. And here's the constraint. We can't build and maintain our own movie or TV database. That's a massive undertaking. We need to plug into an existing source.

Buddy: The solution is TMDB. The Movie Database. It's free, well-maintained, and has comprehensive metadata on movies and TV shows. We'll use their API as our source of truth for content. Our backend proxies requests to TMDB, so the frontend never needs an API key. And our database stores the unique stuff: who rated what, what reviews people wrote, and who has accounts on our platform.

Client: Perfect. So our database is lightweight. Just users, ratings, and reviews. TMDB handles the catalog.

Buddy: Right. And we get a nice architecture lesson out of it: the proxy pattern. It's how real companies work. You wrap third party APIs with your own logic, security, and caching.

User Experience and MVP Scope

Client: Walk me through what a user would do when they open the app for the first time.

Buddy: Okay. Landing page, users search for a movie or show by title. Results come from TMDB. No login required yet. Users pick a movie, see details like poster, synopsis, runtime. They also see the average rating and recent reviews from our community. This is the magic part. They can see what other users thought.

Then the first action. User wants to rate it or write a review. They hit a button, get prompted to sign up or log in. Email and password. This happens on a shared login server we're providing, not on the app itself. After login, they can rate it one through ten stars and write a text review. Users can see their own history, everything they've rated and reviewed. And if something's broken, there's a link to file a bug. This goes to the dev team dashboard.

Client: Nice. And the business model?

Buddy: For the MVP, this is free. No ads, no subscriptions. We're proving the concept. In the future, we add ads, premium features like curated lists and notifications, algorithmic discovery. All on top of the basic search and review platform.

Client: What about scope creep? I want to launch this quarter.

Buddy: We're intentionally strict. The MVP includes search and browse for movies and shows through TMDB. View ratings and reviews from our community. Register and log in. Submit your own rating and review. See your profile and history.

Stretch goals, if there's time: watchlists or want to watch lists. Filter reviews by rating. Sort search results. Featured or trending section.

But we don't ship without the core five working flawlessly.

Technical Architecture

Client: OK, so I heard the phrase backend and frontend. How are you splitting the work?

Buddy: The backend is an API. It's not a website, it's a web service. It has no HTML, no login form, no buttons. It's pure data. Here are the search results for Inception. Here's the average rating for that movie. User 42 submitted a nine star review. That kind of thing.

The frontend is the website or app that calls that API. Users see the search box, click buttons, type reviews. Every click triggers an API call to the backend.

Client: Why split them?

Buddy: Three big reasons. First, once the API exists, we can build an iOS app, an Android app, a smart TV app. All consuming the same backend. No code duplication. Second, a backend team can work independently of a frontend team. They communicate through the API spec, not through shared code. Third, we can update the backend without redeploying the frontend and vice versa. Fast iteration.

Client: And the data. Where does that live?

Buddy: PostgreSQL database. Stores users, ratings, reviews, and bug reports. Zero movie metadata. TMDB is the source of truth for that.

Client: What about security? How do we keep logins secure?

Buddy: We're using a centralized login service. You can think of it like Sign in with Google, but for our platform. Users register once, get a token called a JWT, and present that token to the API for authenticated requests. The token proves they're logged in and which user they are.

Client: Where's this login service hosted?

Buddy: Instructor hosted. For this course, it's provided. In production, companies either build it themselves like Auth0 or Okta or use a third party.

Team Structure and Handoff

Client: How many people are building this?

Buddy: For the MVP, we're organizing it like a real tech company. A backend team of about four to five people builds the API, connects to PostgreSQL and TMDB, and writes docs. A frontend team of about four to five people builds the website or app and calls the backend API.

One team builds the backend first, or at least they ship faster. The frontend team consumes that API.

Client: What if the backend API isn't ready when the frontend team needs it?

Buddy: That's why the backend team writes API documentation first. It's a contract. The frontend team reads it and can start building with mock data before the real API is done. Once the API is ready, integration is just swapping out the mock server for the real one.

Client: And if the backend API is broken or inconsistent?

Buddy: The frontend team files a bug report with details. Something like Endpoint X returns an empty array when it should return 10 results, or the timestamp format doesn't match the

documentation. The backend team sees this on a dashboard, reproduces it, fixes it, and updates the status. Real world workflow.

Client: I like that. Keeps both teams honest.

Technical Constraints and Stack

Client: What tech are we using?

Buddy: Backend is Node.js and Express. That's a JavaScript or TypeScript framework for building APIs. We use PostgreSQL for the database. Frontend is React and Next.js, which is JavaScript or TypeScript for building web interfaces. Authentication uses JWT tokens and the OAuth2 protocol. And we call the TMDB REST API for movie and TV data.

All modern, industry standard tech. Anything your team learns here transfers directly to real jobs.

Client: Why these specifically?

Buddy: They're the popular, approachable stack right now. Express is simple to learn. React is the most used frontend framework. PostgreSQL is rock solid and free. TMDB has great documentation and a generous free tier.

Client: Are we managing servers?

Buddy: No. We're using platform as a service. Render for the backend, they host and scale the API. Vercel for the frontend, they host and deploy the website. Push code, they deploy automatically. No server management.

Client: What about the database?

Buddy: Render also provides managed PostgreSQL. We don't SSH into a server. We use a connection string and let Render manage backups, patches, and scaling.

Data Model at a Glance

Client: How do you actually store the data?

Buddy: Four main tables. Users table has ID, email, display name, and created date. Nothing sensitive. Passwords are hashed and stored on the login service, not here. Ratings table has user ID, TMDB ID for which movie or show, score from one to ten, and timestamp. Simple and fast. Reviews table has user ID, TMDB ID, review title, text body, and timestamp. Allows richer feedback than just a score. Issues table stores bug reports from frontend users. Title,

description, which endpoint was broken, reproduction steps, and status whether it's open, in progress, or resolved.

That's it. No movies table, no TV shows table. We get that from TMDB.

Client: What about watchlists?

Buddy: That's a stretch goal. Adds another table. For MVP, we scope it out.

API Design Philosophy

Client: How do I know the API will work with my frontend?

Buddy: We write the spec first. For example, GET slash movies slash search with q equals inception returns a list of matching movies with fields like id, title, release date, poster URL, and overview. Or GET slash reviews slash 13 returns all reviews for the movie with TMDB ID 13.

The spec is the contract. If the frontend needs a field that's not in the spec, we negotiate and update the spec before coding.

Client: Who writes the spec?

Buddy: The backend team. They own the API design. But they write it before shipping code and they share it with the frontend team for feedback.

Client: What if the frontend team doesn't like it?

Buddy: They say so and they iterate. It's a conversation. The spec isn't handed down. It's agreed on. That's professionalism.

Development Approach

Client: When do I see working code?

Buddy: We work in sprints. Each sprint has specific requirements and deliverables. Your instructor will provide sprint documents with detailed requirements and due dates.

The general flow is: start with foundations and infrastructure, then build out the TMDB proxy, then add user-generated content like ratings and reviews, then build the bug tracker and cross-group integration.

Client: And deployment?

Buddy: The backend goes live early on a hosting service. The frontend follows later. Both teams get working, deployable code in front of users.

Client: What about when one team's code is buggy?

Buddy: That's where the structured bug report system comes in. Frontend team hits a bug, files it with reproduction steps. Backend team sees it on a dashboard, fixes it, and updates the status. Frontend team can track progress. No lost issues. No Slack chaos.

Intentional API Divergence

Client: Earlier you said the backend team owns the API design. Does that mean every backend team designs differently?

Buddy: Yes, intentionally. For search, pagination, filtering, and the TMDB proxy layer, your team makes your own choices. One team might do GET slash movies slash search with q equals inception and page equals one. Another might do GET slash api slash search with type equals movie and query equals inception and limit equals twenty and offset equals zero. Both are valid.

But routes for ratings, reviews, user profiles, and bug reports are standardized across all groups. That's non negotiable. Frontend teams need consistency there.

This teaches students that there's no one right way to design an API. Trade offs exist. And when the frontend team has to integrate with a different API, they learn the real world skill of adapting to divergent APIs.

Client: That seems chaotic.

Buddy: It is, in the best way. It mirrors production. Your company uses Stripe's API, Google's API, your own internal API. They all have different conventions. Engineers learn to read docs and adapt. That's more valuable than everyone does it the same way.

Open Questions and Decisions

Client: What are the unknowns?

Buddy: Watchlists. Do we scope this in as required or defer to stretch goals? Search filtering. Can users filter by release year, genre, rating threshold? User profiles. Public or private? Can you see another user's reviews? Review moderation. Can admins delete inappropriate reviews? Can users flag reviews?

These are all reasonable features but every one adds complexity. My recommendation is start minimal, ship, then add.

Client: What's your timeline for decisions?

Buddy: Sprint planning is weekly. Next sprint requirements drop on Sunday. Teams have a week to build. Decisions on scope happen in retrospectives and planning sessions.

Success Criteria

Client: How do we know this is working?

Buddy: Technically, API documentation is complete and accurate. Frontend can call the API without hacks or workarounds. Cross team bug reports are filed and resolved in under 48 hours.

For the product, users can complete the core flow. Search for movies, view details and community reviews, register, submit their own review.

For quality, no unplanned downtime. Response times under 500 milliseconds for searches. API is 99 percent available.

For the team, both teams ship on time without dependencies blocking them. Communication is clear. No surprises.

Next Steps

Client: So what happens now?

Buddy: We finalize the tech stack and hosting platforms. The instructor provides the centralized auth service, which is the login API, and TMDB credentials. The backend team gets the starter repository with boilerplate like Express, Prisma, and middleware examples. The frontend team gets the starter repository with Next.js, auth setup, and API client examples. Both teams read the API specification, which will be written in sprint zero. Then sprint zero begins with foundations, infrastructure, and the first integration test.

Client: And the budget and timeline?

Buddy: This is for a college course. Timeline is ten weeks. Budget is free because we're using open source tech and free tier services. Constraints are about eight to ten people on the team, learning as they go. Given that, the MVP scope is realistic.

Client: Sounds solid. Let's build it.

Appendix: Key Assumptions

- **TMDB is the source of truth** for media metadata. Our database never stores movie or show data, only user generated content.
- **Authentication is federated.** One login service, multiple APIs trust its tokens.
- **The API is the contract.** Frontend depends on API documentation. If the API changes, the spec changes first.
- **Divergence is intentional.** For search, pagination, filtering, and proxy design, teams make their own choices. Routes for ratings, reviews, user profiles, and bug reports are standardized across all groups.
- **Bug tracking is structured.** Not Slack, not email, but a dedicated dashboard.
- **Scope is intentionally tight.** MVP first, stretch goals second.