Problem Statement

Context

Video games are one of the most popular forms of entertainment worldwide, with a large and diverse player base constantly looking for new games to play and communities to engage with. However, many existing review platforms are either too commercialized, cluttered, or lack features that let everyday gamers express and share their opinions freely. This project exists in that context — an independent, community-driven review app designed for gamers.
The primary users are casual and enthusiast gamers who want to review games, read authentic feedback, and engage with a gaming community. Admin users are responsible for managing the integrity and moderation of the platform. The scope of the issue revolves around how game reviews are currently shared, discovered, and trusted in the gaming community.

Core Problem

There is no lightweight, community-focused platform dedicated solely to writing, browsing, and managing user-generated game reviews in a structured and meaningful way. Existing platforms are often bloated with ads, biased by sponsored content, or lack personalization and simplicity.
Users need a space where they can rate games, edit and delete their own reviews, and trust that the platform is moderated for fairness and safety.

Impact on Users

- **Gamers** face difficulty finding trustworthy, unbiased reviews from people like them. They're often overwhelmed by large platforms pushing editorial content or influencer opinions.

- **Reviewers** don't have a reliable space to share and manage their reviews, and may feel their voices are lost in noisy, commercial ecosystems.

- **Admins** don't have clear tools to maintain quality and enforce rules, which leads to inconsistent moderation and possible misuse.

This leads to missed opportunities for users to discover new games, make informed decisions, and contribute to a meaningful gaming community.

Opportunity for Solution

Solving this problem creates a dedicated space for community-driven, user-authored game reviews. It gives gamers the power to share their voice, discover others' opinions, and participate in a well-moderated, gamer-focused ecosystem.
By focusing on simplicity, authenticity, and usability, this platform will appeal to users who are disillusioned with mainstream review sites and want a cleaner, fairer, and more community-oriented experience.

Technical Solution

Overview of the Solution

We will build a web application that allows users to browse a curated list of games, write reviews, rate those games, and edit or delete their own reviews. Admin users will have access to tools for moderation and content management.
The application will provide a simple and intuitive interface, robust authentication and authorization through Supabase, and a scalable backend for long-term sustainability using Elastic Beanstalk and RDS.

Key Features and Functionalities

- **User Authentication (via Supabase):**
  Sign-up, log-in, and role-based access (user vs. admin).

- **Game Browsing:**
  View a list of available video games with their details (genre, platform, release date, etc.).

- **Review Management:**
  Users can write, update, and delete reviews; leave a rating.

- **Admin Controls:**
  Admins can moderate reviews, remove inappropriate content, and manage game listings.

- **Game Ratings Summary:**
  Aggregate scores based on all user reviews, shown on each game's page.

User Scenarios

Scenario 1: Casual Gamer

Kelly logs in using Supabase auth, browses for a game they just finished playing, and posts a review with a 7/10 rating. A few days later, they return to edit their review after playing more of the game. They also read what others had to say before purchasing their next game.

Scenario 2: Admin Moderator

McIvor is an admin who logs in to check for inappropriate content. They see a flagged review and remove it, ensuring the platform stays safe and respectful. They also add a new indie title to the list of games.

Technology Stack

- **Frontend**:
  React – for building a fast, interactive, and component-based UI.

- **Backend**:
  Spring Boot (Java) – for building a secure and scalable REST API.

- **Database**:
  MySQL (via Amazon RDS) – for storing users, reviews, games, and ratings.

- **Hosting/Deployment**:
  Elastic Beanstalk – to manage Spring Boot deployments with scaling and logging.
  RDS – managed MySQL database for reliable storage.

- **Authentication**:
  Supabase Auth – for easy integration of secure sign-up/login and role-based access.

Glossary

User

Anyone who signs up for an account. All users have a profile. Users can view game pages, where they can leave a review of the game and view other users' reviews of it. Users can perform CRUD methods on their own reviews.

Admin

A user with an administrator role. An admin is able to perform CRUD methods on any user review. All admins are users, but not all users are admins.

Profile

A page showing a user's basic information such as their favorite genre, favorite game, date joined, and a list of reviews they have made.

Game

There are 1000 games, each with a game page that lists basic information such as title, developer, publisher, genre, and ESRB rating. There is also a list of users reviews of the game on the page.

Game Review

A review left on a game page by a user. States the user's numerical rating of the game as well as their written review.

High Level Requirements

Manage 4-7 Database Tables (Entities) that are Independent Concepts
Plan to meet requirement:
The project will involve designing and implementing 4 independent database entities representing core concepts within the application. These entities will have unique attributes and relationships with each other. For example, the entities will include tables for users, games, reviews, and user profiles. These entities will be connected through appropriate relationships, ensuring a logical data structure.

MySQL for Data Management
Plan to meet requirement:
MySQL will be used as the relational database management system for storing and retrieving application data. The database schema will be designed to maintain data integrity and optimize query performance. `JdbcTemplate` will be used to handle database interactions. This will allow for manual SQL queries and provide more control over database transactions compared to Spring Data JPA.

Spring Boot, MVC, JDBC, Testing, React

Plan to meet requirement:

The backend will be built using Spring Boot, leveraging the MVC (Model-View-Controller) pattern to separate concerns and organize the application. `JdbcTemplate` will be utilized for database connectivity, allowing efficient data retrieval and manipulation. React will be used to create the dynamic frontend, ensuring a responsive and interactive user interface. Unit and integration tests will be written for both the backend and frontend to verify the application's functionality.

An HTML and CSS UI Built with React

Plan to meet requirement:

The frontend UI will be created using React, incorporating modern web design practices with HTML and CSS. The UI will be fully responsive, ensuring a seamless experience across desktop and mobile devices. Users will be able to easily browse, rate, and review games, while also managing their profiles and viewing their review history.

Sensible Layering and Pattern Choices

Plan to meet requirement:

The application will follow best practices for software architecture, utilizing the MVC design pattern. The logic will be separated into distinct layers: controller, service, and repository. This structure will ensure that the code is maintainable and scalable. Additionally, `JdbcTemplate` will be used to handle database transactions, providing efficient data interaction.

A Full Test Suite that Covers the Domain and Data Layers

Plan to meet requirement:

A comprehensive test suite will be developed to cover both the domain and data layers. Unit tests will be written for the service and repository layers to verify business logic and data interactions. Integration tests will ensure that the backend, frontend, and database work together as expected.

Must Have at Least 2 Roles (Example: User and Admin)

Plan to meet requirement:

Role-based access control (RBAC) will be implemented using Supabase for authentication. Supabase will handle user sign-in and authentication, while the backend will use roles to differentiate between regular users and admins. Users will be able to review and rate games, while admins will have the ability to manage game entries, approve or delete reviews, and oversee user-generated content.

User Stories

Write a Review

Goal: As a user, I want to write a review and rate a game so I can share my opinion with others.

Plan to meet requirement:
I will implement a "Write Review" form on each game's page that allows logged-in users to write text feedback and provide a rating (e.g., 1 to 10). Users will submit their review, which will be stored in the database and linked to their account and the reviewed game.

Precondition:
User must be logged in with a USER role.
The game must exist in the database.

Post-condition:
The review is saved in the database, associated with the user and game, and becomes visible to other users on the game's page.

Edit a Review

Goal: As a user, I want to edit my own review in case I change my opinion or make a mistake.

Plan to meet requirement:
Users will see an "Edit" button next to their own reviews. Clicking it will open a pre-filled form allowing them to update the text or rating. The backend will validate that the user is the original author before allowing the update.

Precondition:
User must be logged in and must be the original author of the review.

Post-condition:
The review is updated in the database with the new content and timestamp. The updated review replaces the old version in the UI.

Delete a Review

Goal: As a user, I want to delete my own review if I no longer want it to be public.

Plan to meet requirement:
Users will be able to click a "Delete" button next to their reviews. A confirmation prompt will appear. Once confirmed, the backend will delete the review if the requesting user is its author.

Precondition:
User must be logged in and must be the original author of the review.

Post-condition:
The review is removed from the database and no longer appears on the game's review section.

Moderate a Review

Goal: As an ADMIN, I want to remove or flag inappropriate or offensive reviews to keep the platform safe and respectful.

Plan to meet requirement:
Admins will have a moderation dashboard that shows all reviews. Each review will have options to "Delete" or "Flag." Admins can remove reviews that violate community guidelines.

Precondition:
User must be logged in with an ADMIN role.

Post-condition:
The selected review is marked as removed or flagged and no longer appears publicly.

Browse Games and Reviews

Goal: As a user, I want to browse a list of games and read community reviews so I can decide what to play next.

Plan to meet requirement:
The app will feature a searchable and sortable game list. Each game page will show the average rating and a list of user reviews. Filtering by genre, platform, or rating will be supported.

Precondition:
None.

Post-condition:
The user views the list of games and is able to navigate to game-specific pages with reviews and ratings.

Login and Role-Based Access

Goal: As a user, I want to securely log in to access features like writing reviews and rating games.

Plan to meet requirement:
Supabase Auth will manage user registration and login. Role-based access (USER or ADMIN) will determine what actions are permitted (e.g., posting vs. moderating reviews).

Precondition:
User must have a valid Supabase account and be logged in.

Post-condition:
User gains access to functionality based on their role (e.g., post reviews, moderate content, etc.).

What specific knowledge or skill do you aim to gain from this project that you haven't yet learned? Why is it meaningful to you or the project?

I aim to learn how to deploy and scale a production-ready full-stack web application using AWS Elastic Beanstalk for the backend and Amazon RDS for the database. While I have experience building full-stack apps locally, I haven't yet deployed one using AWS services in a way that ensures scalability, high availability, and cloud-based persistence.
This skill is meaningful because it bridges the gap between local development and real-world deployment, helping me understand how cloud infrastructure supports modern applications. It also improves the professionalism and reliability of my project.

Describe how this new knowledge or skill will be used within the project. What specific part of the application or feature will rely on it?

This knowledge will be used to **host the Spring Boot backend on AWS Elastic Beanstalk** and to **manage the MySQL database using Amazon RDS**. It is essential for ensuring that the backend services are accessible to users and can scale based on demand. The deployed backend will connect to RDS to store and retrieve user reviews, ratings, and game data.
Additionally, Elastic Beanstalk will manage environment variables, deployments, and autoscaling—critical for maintaining a stable application.

List initial resources you plan to use to understand this concept (e.g., official documentation, tutorials, or textbooks). Are there any third-party libraries or tools? Will you need an API key or extra setup?

- **AWS Elastic Beanstalk Documentation**
- **AWS RDS Documentation**
- YouTube tutorials like "Deploying Spring Boot apps to Elastic Beanstalk" and "Connecting Spring Boot to Amazon RDS"
- **Baeldung guides** for deploying Java apps
- **AWS Free Tier account** for setup and experimentation

What challenges do you anticipate in learning and applying this skill? How do you plan to address them?

- **Challenge:** Configuring environment variables and database connections securely in the AWS environment
  Solution: Use Elastic Beanstalk's environment settings and practice with test data before production deployment
- **Challenge:** Handling deployment errors and logs on AWS
  Solution: Learn how to monitor logs using Elastic Beanstalk and CloudWatch; run through deployment cycles with dummy builds
- **Challenge:** Ensuring security of the RDS instance (e.g., proper VPC and security group setup)
  Solution: Follow best practices from AWS docs and tutorials, especially regarding public access and port management
- **Challenge:** Cost management
  Solution: Use free-tier resources and shut down environments when not in use

---

How will you measure whether you have achieved your learning goal? What will the successful implementation of this skill or technology look like in your project?

I will know I've achieved this learning goal when:

- My **Spring Boot backend is successfully hosted on Elastic Beanstalk**
- The backend can **reliably connect to a live Amazon RDS MySQL instance**
- Users can log in, submit reviews, and interact with the database in real-time through the hosted application
- I can monitor logs, view deployments, and manage environments through AWS
  A **fully functional live URL** accessible to other users will be the final proof of success.

Task List with estimated completion times
In this section, you will provide a detailed breakdown of the tasks required to complete your project (any task over 4 hours must be broken down further). List each task involved in building and implementing the system, from setting up the database to creating the UI and testing the functionality. For each task, estimate how much time you expect it to take to complete, based on its complexity. This will help you manage your time and ensure that the project stays on track. Tasks should be organized logically, and you should be as detailed as possible, covering all the components required for your application.

Model "Layer" (Total Estimated Time: ~2-3 hours)

- In IntelliJ, create model object Java classes for each element in our project: User, Game, GameReview, and Profile. Create getters and setters per field per object. There is additionally a Region enum consisting of NA, EU, JP, and OTHER. (Estimated completion time: 15-20 minutes)
- In MySQL Workbench, create database tables for each object, following along with our already-made ERD. Define primary and foreign keys, determine which fields are non-nullable, and ensure field types and sizes are reasonable. (Estimated completion time: 30-45 minutes)
- In MySQL Workbench, use the data import wizard to populate the Game table, ensuring any invalid rows are excluded, and basic queries can be ran on the table. (Estimated completion time: 15-20 minutes)
- Create and insert data for the User and Profile tables, ensuring foreign key constraints are respected. Users are required to have a profile, but not all profile elements are required. (Estimated completion time: 30-45 minutes)
- Create and insert data for the GameReview table, ensuring foreign key constraints are respected. (Estimated completion time: 15-20 minutes)

- Take a portion of the “Production data” and create a second copy of the schema to use as a testing database. Include a “set_known_good_state” function. (Estimated completion time: 30-45 minutes)

Data Layer (Total Estimated Time: ~5-6 hours)

- Create the interfaces for each element’s repository: User, Game, GameReview, and Profile. Include all required CRUD methods for each. (Estimated completion time: 30 minutes)
- Create mapper classes for each element’s MySQL table: User, Game, GameReview, and Profile. Each foreign key reference to an id becomes the object it’s referencing on the “Java side of things” (see assessment 09). (Estimated completion time: 1 hour)
- Implement CRUD operations for GameJdbcTemplateRepository. Create, Update, and Delete* are to be used by an Admin user only. All users can view (Read) games, but nothing else. (Estimated completion time: 1 hour- 90 minutes)

  ** if a game is deleted, all of its reviews are also deleted.*
- Implement CRUD operations for UserJdbcTemplateRepository. Admin users can perform all CRUD operations on all users* ** (like for a password reset!). All users can use all CRUD operations on/for their own account (via a settings page), but they can only create one! (Estimated completion time: 1 hour- 90 minutes)

  ** Deletion entails deactivating their account, and deleting their profile alongside it.*
  *** Since profiles and users are created and deleted together always, there is no real dependency issue there. However, deleting a user will leave an invalid user_id on all of their reviews. To remedy this, we plan on having an “un-deletable” user named “deleted account” that will “become” the review’s new writer!*

- Implement CRUD operations for ProfileJdbcTemplateRepository. Neither admin nor user can create or delete a profile, as that’s handled on account creation and deletion respectively. Admins can read and update all profiles (as a moderation tool), and users can read all profiles, but only update their own. (Estimated completion time: 1 hour)
- Implement CRUD operations for ReviewJdbcTemplateRepository. Admins can perform CRUD operations on all reviews, and users can read all reviews, but can only create, update, and delete their own reviews. (Estimated completion time: 1 hour)

Data Layer (Testing) (Total Estimated Time: ~2-3 hours)

- UserJdbcTemplateRepositoryTest: findAll and findById(happy and sad paths), add, update(happy and sad paths), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- ProfileJdbcTemplateRepositoryTest: findById(happy and sad paths), add(for backend use only), update(happy and sad paths), and delete(for backend use only, happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- GameJdbcTemplateRepositoryTest: findAll, findById(happy and sad paths), findByGenre(happy and sad paths), add, update(happy and sad paths), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- GameReviewJdbcTemplateRepositoryTest: findAll, findById(happy and sad paths), add, update(happy and sad paths), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- KnownGoodState object for calling the previously written SQL function for setting the good state in the test database (Estimated completion time: 10-15 minutes)

Domain Layer (Total Estimated Time: ~2-3 hours)

- Create a Result object (+ status enum) for keeping track of errors and result status (Estimated completion time: 10 minutes)
- UserService: findAll and findById are pass-throughs*, add requires a unique username, a unique email, and a 9-character minimum, 50-character maximum password with at least 1 special character, update requires a unique username, a unique email, and a 9-character minimum, 50-character maximum password with at least 1 special character, on top of not allowing the previously used password, and delete is pass-through*. (Estimated completion time: 30-45 minutes)
- ProfileService: findById is pass-through, add doesn't require any fields, and is therefore pass-through, update doesn't require any fields, and is therefore pass-through, and delete is pass-through. (Estimated completion time: 5-15 minutes)
- GameService: findAll, findById, and findByGenre are pass-through, add requires a title and developer, update requires a title and developer, and delete is pass-through. (Estimated completion time: 30-45 minutes)
- GameReview: findAll and findById are pass-through, add requires a rating from 1-5, decimals allowed (rounded up to 1 digit ex. 2.25/5 => 2.3/5), update requires a rating from 1-5, decimals allowed, and delete is pass-through. (Estimated completion time: 30-45 minutes)

** If convenient, all pass-throughs can simply store their results in a result object and return that instead of being purely pass-through. Remember YAGNI!*

Domain Layer (Testing) (Total Estimated Time: 2-3 hours)

- UserServiceTest: findAll and findById(happy and sad paths), add (valid, empty email, duplicate email, empty username, short username, duplicate username, no password, short password, long password, weak password), update (valid, empty email, duplicate email, empty username, short username, duplicate username, no password, short password, long password, weak password), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- ProfileJdbcTemplateRepositoryTest: findById(happy and sad paths), add (for backend use only), update(happy and sad paths), and delete(for backend use only, happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- GameJdbcTemplateRepositoryTest: findAll, findById(happy and sad paths), findByGenre(happy and sad paths), add (empty/null title, empty/null developer), update (empty/null title, empty/null developer), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)
- GameReviewJdbcTemplateRepositoryTest: findAll, findById(happy and sad paths), add (null rating, invalid rating, reviewWithMultipleDigitDecimals), update(null rating, invalid rating, reviewWithMultipleDigitDecimals), and delete (happy and sad paths) need testing. (Estimated completion time: 30-45 minutes)

Controller Layer (Total Estimated Time: ~4-5 hours)

- Create an ErrorResponse object for handling HTTP statuses and attaching our error messages from lower layers. (Estimated completion time: 30-45 minutes)
- UserController implements endpoints per service method, ensures the correct HTTP code is being returned per method call and establishes endpoints for account creation and user info modification. Include testing endpoints via VSCode.(Estimated completion time: 30-45 minutes)
- Profile Controller implements endpoints for viewing + editing a profile *only* and ensures the correct HTTP code is being returned per method call. Include testing endpoints via VSCode. (Estimated completion time: 30-45 minutes)
- GameReviewController implements endpoints per service method, ensures the correct HTTP code is being returned per method call and establishes endpoints for account creation and user info modification. Include testing endpoints via VSCode. (Estimated completion time: 30-45 minutes)
- GameController implements endpoints per service method, ensures the correct HTTP code is being returned per method call and establishes endpoints for

account creation and user info modification. Include testing endpoints via VSCode. (Estimated completion time: 30-45 minutes)

- Implement a GlobalExceptionHandler that obfuscates application structure through JDBC errors by replacing errors with custom, more vague errors + generic error status codes. Include testing error handling via VSCode. (Estimated completion time: 30-45 minutes)

HTTP/JS “Layer” (Total Estimated Time: ~12-13 hours)

- Design a home page with a navbar for account creation + account login, and a front page list of the top 10 rated games listed on LIGR.* Don’t allow profile access while signed out. (Estimated completion time: 90 minutes)

  ** Lorem Ipsum Game Reviews, colloquial site name, pronounced “liger”*
- Design a login/sign up page, handled via Supabase for authentication. (Estimated completion time: 2 - 2.5 hours)
  - Create forms for account creation, likely reusable in a similar way to add/update forms being reusable
- Design a game page: displays game title, developer, genre, region, ESRB rating, platform, and all of its reviews. Include a conditional delete button for admins to delete the game listing. (Estimated completion time: 90 minutes)
- Design a game review page: Links back to the game’s page on the review, and displays the review writer, rating, and the review itself, if the reviewer wrote one. Include a conditional delete button for admin users and the user that wrote the review to delete the review. (Estimated completion time: 90 minutes)
- Design a user profile page: displays the user’s username, any fields in their profile they may have filled out, and all the reviews they may have written. (Estimated completion time: 90 minutes)
  - If you recall, deleted accounts + their reviews are being handled by a special user named “deleted account” - this user’s profile is admin access-only!
- Design a settings page for users (or admins) to edit their account settings (username + email), or delete their account. (Estimated completion time: 90 minutes)
- Design a page for users (or admins) to edit their profile. (Estimated completion time: 90 minutes)
- There should be a generic 404 error page for accessing any user profile, game, or review that doesn’t exist, that acts doubly as a page redirect for non-admin users that attempt to access admin-only pages (Estimated completion time: 90 minutes)
- Design an admin access only user repository page, to list all users + include a button that takes you to their profile and/or account settings page for moderation purposes. (Estimated completion time: 90 minutes)