Project Based Evaluation

Project Report

Semester - V (Batch-2023)

Cineverse

Supervised By:

Dr. Priyanka Gupta

Submitted By:

Athrv Sharma, 2310991632

Ananaya Sharma, 2310991605

Gurnoor Singh Gill, 2310991666

Department of Computer Science and Engineering

Chitkara University Institute of Engineering & Technology,

Chitkara University, Punjab

ABSTRACT

Cineverse is a production-ready cinema experience platform that combines a personalized viewer portal, an administrative creator console, and an operational control room in one Express.js application. Built with EJS layouts on top of a dual-database stack—MongoDB Atlas for user identities and profiles, PostgreSQL for the authoritative film catalog—the system delivers responsive catalog browsing, trailer-supported detail views, and synchronized "My List" watchlists for authenticated users. Administrators can add, edit, and retire titles through validated forms that write to PostgreSQL, automatically warm Redis caches, and trigger Socket.io broadcasts so every connected session refreshes without a page reload. A dedicated dashboard synthesizes catalog analytics, user counts, and datastore health using Chart.js and live metrics streamed via the same real-time layer. Security is layered throughout the project: Passport handles local and Google OAuth logins, JSON Web Tokens guard the REST API, Redis-backed rate limiting and session storage throttle abusive traffic, and optional HTTPS/TLS enforcement plus hardened HTTP headers protect every response. By weaving together these fully implemented modules, Cineverse demonstrates how modern full-stack engineering can power a dependable, data-driven movie curation workflow that is ready for classroom demos and real-world deployments alike.

Table of Contents

1.Introduction

1.1 Background

Cineverse responds to the surge in digital film discovery, where viewers hop between multiple apps to track premieres, festival picks, and curated catalogs, while curators juggle spreadsheets, social media, and siloed databases. This fragmentation leaves fans with inconsistent data and creators with no real-time feedback on their catalog. Cineverse consolidates the entire flow: authenticated users explore a responsive EJS-driven catalog, stream trailers, and maintain personal “My List” queues, while administrators manage the authoritative PostgreSQL movie store directly from the browser. MongoDB handles identity, Redis accelerates caching, and Socket.io keeps every audience member, curator, and dashboard in sync the moment a title changes. By grounding the platform in Express.js, Atlas, Postgres, and Redis, Cineverse delivers a cohesive, trustworthy cinema experience instead of another static showcase.

1.2 Objective

The main goals of Cineverse are:

- Build a unified Express.js/EJS backend that serves both the immersive viewer portal and the curator console.
- Provide secure authentication through Passport (email/password plus Google OAuth) so only authorized users reach the catalog and admin tools.
- Maintain the canonical film catalog inside PostgreSQL with validated add/edit/delete workflows, ensuring data consistency and immediate cache invalidation..
- Deliver a live operational dashboard (Chart.js + WebSockets) that reports movie analytics, user counts, and datastore health without manual refreshes.
- Leverage Redis caching, rate limiting, and Socket.io broadcasts so catalog updates, watchlists, and dashboard metrics refresh across sessions in real time.

1.3 Key Features

- **User Authentication:** Email/password plus Google OAuth handled by Passport, with session management via MongoDB and Redis.
- **Movie Catalog:** PostgreSQL-backed listings showing posters, metadata, trailers, and modal details for every title.
- **Watchlist & Discovery:** Logged-in viewers maintain "My List" entries and browse responsive hero rails, grids, and search filters.
- **Admin Console:** Authorized curators add, edit, or delete titles through validated forms that update the catalog immediately.
- **Realtime Dashboard:** The dashboard streams movie analytics, genre splits, user totals, and datastore health using Chart.js and WebSockets.
- **Secure APIs:** JWT-protected REST endpoints plus rate limiting, security headers, and optional HTTPS/TLS keep integrations safe.

1.4 Significance

Cineverse streamlines how our team curates and shares world cinema by unifying viewer experiences, catalog management, and operational insights in one stack. Students, curators, and administrators no longer juggle spreadsheets or stale pages—every trailer, watchlist change, and data point refreshes instantly while the backend enforces consistent, secure storage. This project demonstrates how a thoughtful combination of Express.js, PostgreSQL, MongoDB, Redis, and Socket.io can deliver a polished, trustworthy showcase that is ready for classroom demos and future real-world deployments.

2.Problem Definition and Requirements

2.1 Problem Statement

Cineverse addresses the gap between modern cinephiles who want a single, dynamic place to discover films and the curators who currently maintain catalogs through spreadsheets, static sites, or siloed apps. Without a unified workflow, viewers lose track of premieres, descriptions, and trailers, while administrators struggle to keep metadata, watchlists, and analytics synchronized across multiple tools. Cineverse solves this by pairing a secure viewer portal with an admin console and realtime dashboard inside one Express.js deployment. Authenticated users browse the PostgreSQL-backed catalog, stream trailers, and manage "My List" entries; administrators manage titles directly in the browser, and Redis + Socket.io ensure that every change reaches all sessions instantly. The platform therefore eliminates the inconsistency and latency that plague traditional movie showcases.

2.2 Objectives

1. **Identity and Authentication:**

- Provide secure onboarding through Passport—email/password plus Google OAuth—so only verified users reach Cineverse.
- Maintain sessions via MongoDB + Redis and secure all REST APIs with JWT tokens.

2. Catalog Management:

- Persist the canonical movie catalog in PostgreSQL with validated admin CRUD flows.
- Keep metadata consistent (genres, cast, runtime, trailers) and indexed for fast retrieval.

3. Viewer Experience:

- Render cinematic pages with EJS/Tailwind, complete with hero rails, search, and movie detail modals.
- Let authenticated viewers manage "My List" entries and play trailers directly from the catalog.

4. Real-Time Collaboration:

- Use Redis cache invalidation and Socket.io fan-out so every movie add/edit/delete appears instantly across sessions.
- Stream dashboard analytics (movie totals, genres, user counts) live to administrators.

5. Administrative Control:

- Restrict the creator console, dashboard, and analytics APIs to admin roles only.
- Offer a web-based dashboard that tracks catalog status, datastore health, and system security.

6. Security & Reliability:

- Hash passwords with bcrypt, enforce rate limiting, and apply strict HTTP headers/TLS where available.
- Monitor MongoDB, PostgreSQL, and Redis status via health-check services to ensure reliable real-time operation.

2.3 Non Functional Requirements

1. **Performance:**

- Handle concurrent authenticated sessions for viewers and administrators without noticeable slowdowns; Redis caching keeps catalog responses fast.
- Maintain low perceived latency for live dashboards and Socket.io pushes by broadcasting updates immediately after database writes.

2. Security:

- Hash every password with bcrypt and guard REST APIs with JWT plus Passport session checks
- Enforce rate limiting, validate URLs during movie creation, and apply HTTP security headers (HSTS, X-Frame-Options, etc.) to reduce common exploits.

3. Scalability:

- Keep routes, controllers, models, services, and sockets modular so new endpoints or dashboards can be added with minimal coupling.
- Rely on PostgreSQL indexes (year, genres, director) and Redis cache eviction to sustain faster catalog queries as data grows.

4. Reliability & Availability:

- Monitor MongoDB, PostgreSQL, and Redis health via service checks before rendering dashboards; fail gracefully with flash messages when dependencies drop.
- Store Express sessions in Redis (when available) to survive server restarts without logging users out.

5. Usability:

- Provide a responsive UI built with Tailwind CSS, GSAP motion, and accessible modals so both viewers and admins can navigate easily on desktops or tablets.
- Keep key actions (add movie, search, watchlist, dashboard refresh) within one or two clicks/taps.

6. Maintainability:

- Organize the codebase into clear modules (routes/, controllers/, models/, services/, middlewares/, views/) with reusable helpers like redisCache.js and securityService.js.
- Document environment variables via .env.example and use npm scripts to install/run the project consistently across team members.

2.4 Software Requirements

- Frontend: EJS templates, Tailwind CSS, custom CSS (public/css/app.css), Vanilla JavaScript modules plus GSAP animations and Chart.js visualizations
- Backend: Node.js with Express.js, Passport (local + Google), Socket.io, and server middleware (session management, flash messages, body-parser).
- Datastores: MongoDB Atlas for authentication/state, PostgreSQL for the canonical movie catalog, Redis for caching, rate limiting, and session storage.
- Package Manager: npm
- Development Tools:
    - Visual Studio Code
- Operating System: Cross-platform (macOS, Windows, Linux) as long as Node.js, PostgreSQL, MongoDB, and Redis binaries are available.

2.5 Hardware Requirements

- **Processor**: Dual-core Intel i5/AMD equivalent (or Apple Silicon M1/M2) capable of running Node.js plus local databases.
- **RAM**: Minimum 8 GB (16 GB recommended when MongoDB, PostgreSQL, Redis, and the app run concurrently).
- **Storage**: ~2 GB free for source code, dependencies (node_modules), local database files, and logs.

2.6 Data Sets and Structures

- **Movie Catalog Data:** Stored in PostgreSQL table cineverse_movies (see models/Movie.js). Each row captures title, poster, backdrop, description, year, director, runtime, and a numeric rating, plus JSONB arrays for genres, cast_members, and optional trailer_url. Indexes on year, genres, and director keep lookups fast.

- **User Directory:** Maintained in MongoDB via models/User.js, which defines fields for name, email, hashed password (optional for Google users), googleId, isAdmin, a myList string array for watchlists, and timestamps.
- Cache & Session Data: Redis stores hot movie snapshots under the key cineverse:movies, general API/cache entries via helper methods in services/redisCache.js, and Express.

3. Proposed Design and Methodology

Overview

Cineverse is built as a full-stack cinema management platform that separates concerns using a pragmatic MVC structure: MongoDB handles authentication and user state, PostgreSQL stores the canonical movie catalog, and Express.js + EJS render dynamic views for both viewers and administrators. Redis sits beside the application to cache catalog payloads, throttle brute-force requests, and persist sessions, while Socket.io fans out real-time events so watchlists, dashboards, and hero rails stay synchronized. Security spans the entire stack: Passport manages sessions for the web UI, JSON Web Tokens secure the REST APIs, and optional TLS plus hardened headers protect transport. This modular approach keeps the codebase maintainable and ready to scale as more titles, users, or dashboards are added.

Layers of Architecture:

1.Routes Layer:

➢ Defines HTTP endpoints for landing pages (/), authentication (/users/login, /users/register, Google OAuth), movie CRUD and watchlists (/movies/add, /movies/edit/:id, /movies/list/add/remove), dashboards (/dashboard, /dashboard/metrics), and JWT-protected APIs (/api/movies, /api/system/insights).

➢ Routes apply middleware such as ensureAuthenticated, ensureAdmin, rateLimiterRedis, and authJwt to protect sensitive paths.

2.Controller/Service Layer:

➢ controllers/movieController.js handles business logic: fetching the PostgreSQL catalog, mapping rows, invalidating Redis caches, and emitting socket events for create/update/delete actions.

➢ controllers/dashboardController.js aggregates movie analytics, datastore health checks, scaling guidance, and security hashes before responding (and broadcasting) to the dashboard.

3.Models Layer:

➢ models/Movie.js defines the PostgreSQL schema (JSONB columns, indexes, CRUD helpers) using the `pg` client.

➢ models/User.js uses Mongoose to describe MongoDB documents, including name, email, bcrypt-hashed password, Google ID, admin flag, and the `myList` array.

➢ Ensures data consistency and validation before storing in MongoDB.

4.Middleware Layer:

➢ config/auth.js exposes `ensureAuthenticated` and `ensureAdmin` for session-based access control.

➢ middlewares/authJwt.js validates JWT tokens for API routes, with optional admin enforcement.

5.Utilities Layer:

➢ sockets/index.js initializes Socket.io, wires the internal `eventBus`, and listens for `movie:*` and `dashboard:metrics` events.

System Flow Overview:

- Client sends a request (e.g., login, browse, add movie).
- Route determines the handler and applies middleware (auth checks, rate limits).
- Controller invokes services/models to read/write PostgreSQL or MongoDB, hitting Redis caches where possible.
- Responses are rendered via EJS templates (viewer catalog, dashboard, add-movie form) or returned as JSON for APIs.
- On catalog or analytics changes, controllers invalidate caches, push events to the `eventBus`, and Socket.io propagates updates to all connected browsers.

This layered approach keeps Cineverse clear, modular, and ready for future expansion.

3.2 Frontend Design

The Cineverse frontend focuses on a cinematic UX that stays responsive across desktops and tablets. Server-side rendering uses EJS layouts plus Tailwind CSS and custom styles (public/css/app.css) to keep interactions smooth, while GSAP and Socket.io power live animations and updates.

Key Frontend Components

Login & Registration Pages

➤ Glassmorphic forms (views/login.ejs, views/register.ejs) with client-side validation hints and flash messages for errors or success.

➤ Support for both email/password and Google OAuth buttons, styled consistently with the brand.

Viewer Catalog / Home Feed

➤ views/index-user.ejs renders hero banners, search bar, movie grids, and modal detail views with poster, synopsis, metadata, and "My List" controls.

➤ User watchlists sync instantly across buttons, and a responsive modal plays trailers inline when URLs exist.

Admin Home + Dashboard

➤ views/index.ejs unlocks curator-only sections showing catalog stats, curated carousels, and hero metrics.

➤ views/dashboard.ejs surfaces a live operations panel with Chart.js visualizations, catalog cards, and security highlights.

Real-Time Enhancements

➤ public/js/realtime-movies.js listens for Socket.io events to update local caches and show toast notifications whenever movies are added, edited, or removed.

➤ public/js/realtime-dashboard.js draws dynamic charts, caches API payloads, and refreshes UI cards without reloading.

Design Principles

➤ Responsive layouts built with Tailwind utilities and custom CSS gradients, guaranteeing consistent behavior across HD displays and projectors.

➤ High-contrast typography and modular "glass-card" components make stats legible even in presentation mode.

➤ Immediate visual feedback—loaders, toasts, button state changes—ensures users know when watchlists or admin actions succeed.

3.3 Backend & Server-Side Logic

Cineverse's backend is the core engine, built on Node.js and Express.js. It manages authentication, catalog CRUD flows, real-time fan-out, and admin analytics with a clean separation between routes, controllers, models, and services.

Authentication Flow

➤ Users register via /users/register; passwords are hashed with bcrypt and stored in MongoDB.

➤ Login (email/password or Google OAuth) establishes a Passport session; rate-limited APIs can issue JWTs for external clients via /api/auth/login.

➤ Protected API routes rely on authJwt middleware to validate tokens and enforce admin-only access where needed.

Movie Management

➤ Admin routes in routes/movies.js handle add/edit/delete forms, validate URLs, and invoke movieController methods to write to PostgreSQL.

➤ After each write, Redis caches are invalidated, events are emitted, and all connected clients receive live updates.

➤ Viewer "My List" actions append or remove titles inside MongoDB using $addToSet and $pull.

Real-Time Pipeline

➤ sockets/index.js initializes Socket.io; the
shared eventBus forwards movie:created, movie:updated, movie:deleted,
and dashboard:metrics events to every browser.

➤ Frontend scripts listen for these events to refresh cached data, show toasts, and re-render Chart.js sections without reloads.

Admin & Analytics Features

➤ /dashboard renders the Mission Control view; /dashboard/metrics aggregates PostgreSQL analytics, MongoDB/Redis health, scaling guidance, and security hashes.

➤ Administrators can monitor totals, genres, user counts, and datastore status in real time, ensuring the catalog remains healthy.

This layered backend keeps business logic isolated from storage details, making Cineverse maintainable and ready for future modules.

3.4 Database Management

Cineverse uses a dual-database strategy to keep authentication and catalog workloads isolated while staying performant and scalable. MongoDB Atlas stores user identities with Mongoose models, whereas PostgreSQL holds the authoritative movie catalog with JSONB fields for genres and cast lists.

Collections / Tables

MongoDB – Users (models/User.js): Stores names, emails, bcrypt-hashed passwords (optional for Google users), linked googleId, isAdmin flag, myList array for watchlists, and timestamps. Mongoose enforces schema validation and makes role checks straightforward.

PostgreSQL – cineverse_movies (models/Movie.js): Contains every
film's title, poster, backdrop, description, year, director, runtime, numeric rating, and optional trailer_url. JSONB columns (genres, cast_members) capture flexible arrays without

sacrificing query speed. Indexes exist on release year, genres (GIN), and director for fast filtering.

Design Considerations

PostgreSQL pooling (services/relationalService.js) manages SSL settings and health checks, while ensureMovieStore auto-provisions tables and indexes when the app boots.

Redis caches (services/redisCache.js) store hot snapshots of movie lists under cineverse:movies, reducing repeated SQL queries and accelerating dashboard requests.

MongoDB handles session-backed data, watchlists, and OAuth metadata, keeping security-sensitive information in a document store that scales horizontally.

Separation of concerns: user actions affect MongoDB and Redis, whereas the catalog remains consistent in PostgreSQL, making backups, analytics, and migrations easier to manage.

3.5 Security Considerations

Security is embedded in Cineverse's architecture:

➤ **Password Encryption:** All email/password registrations are hashed with bcrypt (routes/users.js, config/passport.js), ensuring plaintext credentials never touch MongoDB.

➤ **JWT Authentication:** REST APIs issue and verify JSON Web Tokens (services/jwtService.js, middlewares/authJwt.js) so external integrations and admin dashboards enforce role-based access.

➤ **Rate Limiting & Input Validation:** Redis-backed middleware throttles login and API attempts (middlewares/rateLimiterRedis.js), while movie forms validate URLs and fields before persisting to PostgreSQL (routes/movies.js).

➤ **Session & OAuth Security:** Passport manages sessions, Google OAuth logins, and flash-based feedback; Redis session storage hardens cookie handling, and Google callbacks are restricted to trusted domains.

➤ **Transport & Headers:** securityService.applySecurityHeaders adds HSTS, X-Frame-Options, and Referrer-Policy headers; optional TLS certificates can be loaded via environment variables to run Cineverse over HTTPS.

➤ **Secret Management**: .env (documented in .env.example) stores MongoDB/PostgreSQL URIs, JWT/SESSION secrets, Redis URLs, Google OAuth keys, and TLS paths—keeping sensitive data out of source control.
These measures ensure authentication, data integrity, and live dashboards remain protected even under concurrent usage.

4. Results

4.1 Functional Deliverables

Cineverse delivers every core feature that was scoped for this build: secure onboarding flows, curator-grade catalog controls, synchronized viewer experiences, and an operational dashboard. Each module was implemented directly in Express.js/EJS, wired to MongoDB, PostgreSQL, Redis, and Socket.io, and exercised through the web UI and REST APIs.

User Authentication & Verification:

➤ Email/password registration stores bcrypt-hashed credentials in MongoDB, while Google OAuth lets users sign in with their campus accounts.

➤ Passport sessions keep the viewer portal protected, and JWT tokens from /api/auth/ login guard REST calls.

➤ Role checks (viewer vs. admin) determine who can access creator tools, dashboards, or protected APIs.

Catalog Management & Discoverability:

➤ Administrators add, edit, or delete titles via validated forms; data flows into PostgreSQL with Redis cache invalidation and immediate socket fan-out.

➤ Viewers browse responsive hero rails, search, open cinematic modals, and toggle “My List” entries that persist in MongoDB.

➤ Trailer playback, metadata badges, and curated carousels confirm the catalog renders correctly for both authenticated viewers and admins.

Real-Time Updates:

➤ Socket.io broadcasts movie:created/updated/deleted events, triggering toasts and watchlist refreshes across every open session without page reloads.

➤ The same channel streams dashboard:metrics, keeping analytics cards and Chart.js visualizations current with each API poll.

Operational Dashboard:

➤ /dashboard displays total titles, average rating, user counts, and genre breakdowns pulled from PostgreSQL analytics.

➤ Additional panels surface MongoDB/Redis health, scaling guidance, and hash digests from the security service, giving administrators live insight into infrastructure status.

4.2 Technical Achievements

Modular Backend Design:

➤ Express.js is organized into distinct routes, controllers, models, services, and middlewares (routes/, controllers/, models/, services/, middlewares/), ensuring each concern is testable and easy to extend.

➤ Shared helpers such as redisCache, databaseScaling, securityService, and jwtService encapsulate infrastructure logic so new features can plug in without touching core flows.

Realtime Event Fabric:

➤ Socket.io (initialized in sockets/index.js) listens to an internal eventBus so movie CRUD operations and dashboard metrics propagate instantly to every browser.

➤ Frontend listeners (public/js/realtime-movies.js, public/js/realtime-dashboard.js) update caches, show toasts, and redraw Chart.js components without a reload, proving the live-sync pipeline end-to-end.

Secure Authentication & Access Control:

➤ Passport handles local (bcrypt-hashed) credentials plus Google OAuth, while JWT tokens protect REST APIs and enforce admin-only analytics endpoints.

➤ Redis-powered rate limiting and optional TLS/security headers harden login flows, session cookies, and API transport.

Database Optimization & Caching:

➤ PostgreSQL movie storage uses JSONB columns with indexes on year, genres (GIN), and director, provisioned automatically via ensureMovieStore().

➤ Redis caches catalog snapshots and sessions, reducing repeated SQL reads and enabling fast dashboard loads; health checks (MongoDB, PostgreSQL, Redis) surface status in the admin UI.

Database Optimization:

➤ PostgreSQL catalog tables are provisioned automatically with indexes on release year, genres (GIN), and director via ensureMovieStore(), ensuring fast queries as data grows.

➤ MongoDB houses user identities and watchlists, while Redis caches (cineverse:movies, session store) keep hot payloads in memory and invalidate instantly after CRUD operations.

Operational Telemetry & Security Instrumentation:

➤ The admin dashboard consumes /dashboard/metrics, blending PostgreSQL analytics, MongoDB + Redis health checks, scaling guidance, and security hashes (MD5/SHA-256) so curators see system status in real time.

➤ Security headers, optional TLS certificates, and rate-limit telemetry are centralized inside securityService and rateLimiterRedis, making it easier to audit transport protections.

Testing & Validation Workflow:

➤ Browser-based walkthroughs confirmed registration/login (local + Google), movie CRUD forms, watchlist toggles, and dashboard rendering.

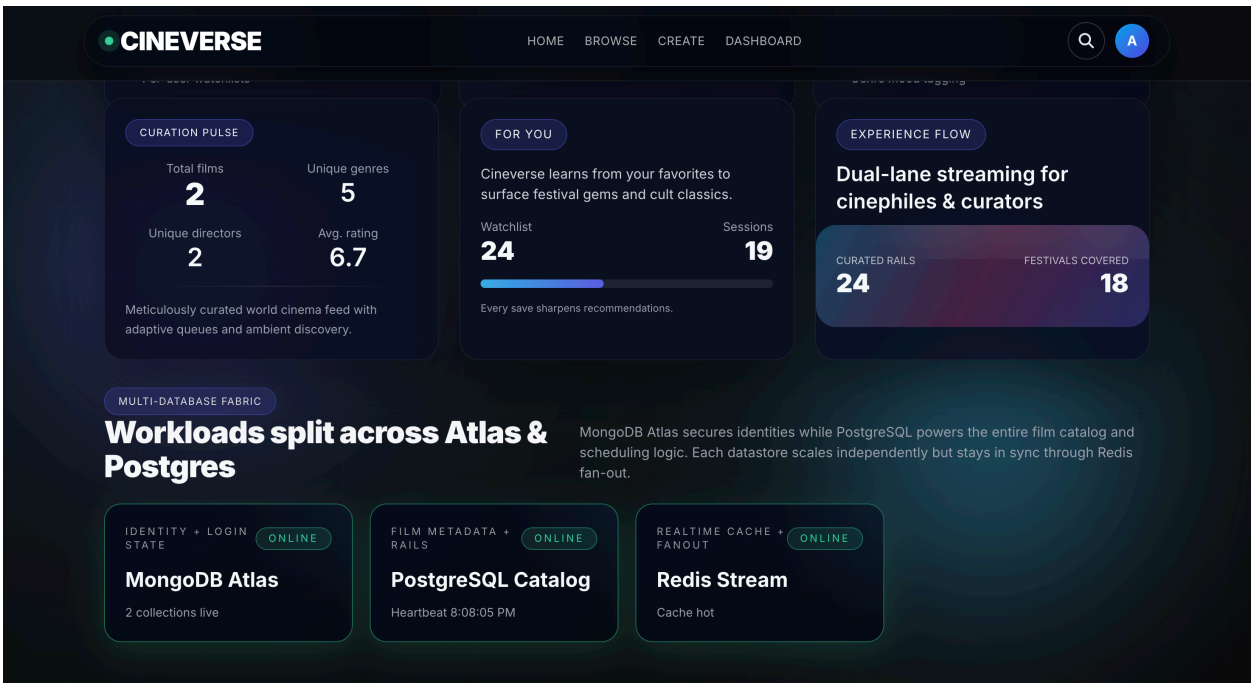
➤ REST endpoints (/api/auth/login, /api/movies, /api/system/insights) were exercised with Postman/cURL to verify JWT issuance, role checks, and JSON payload integrity.

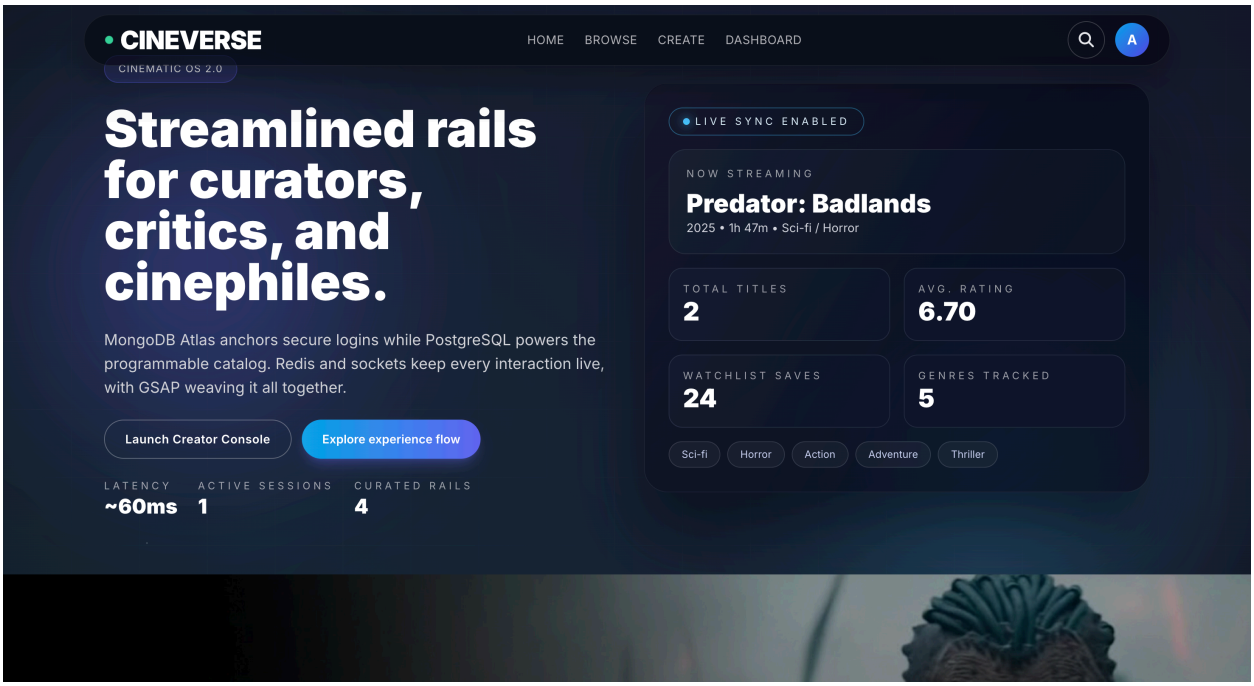
Scalability & Extensibility:

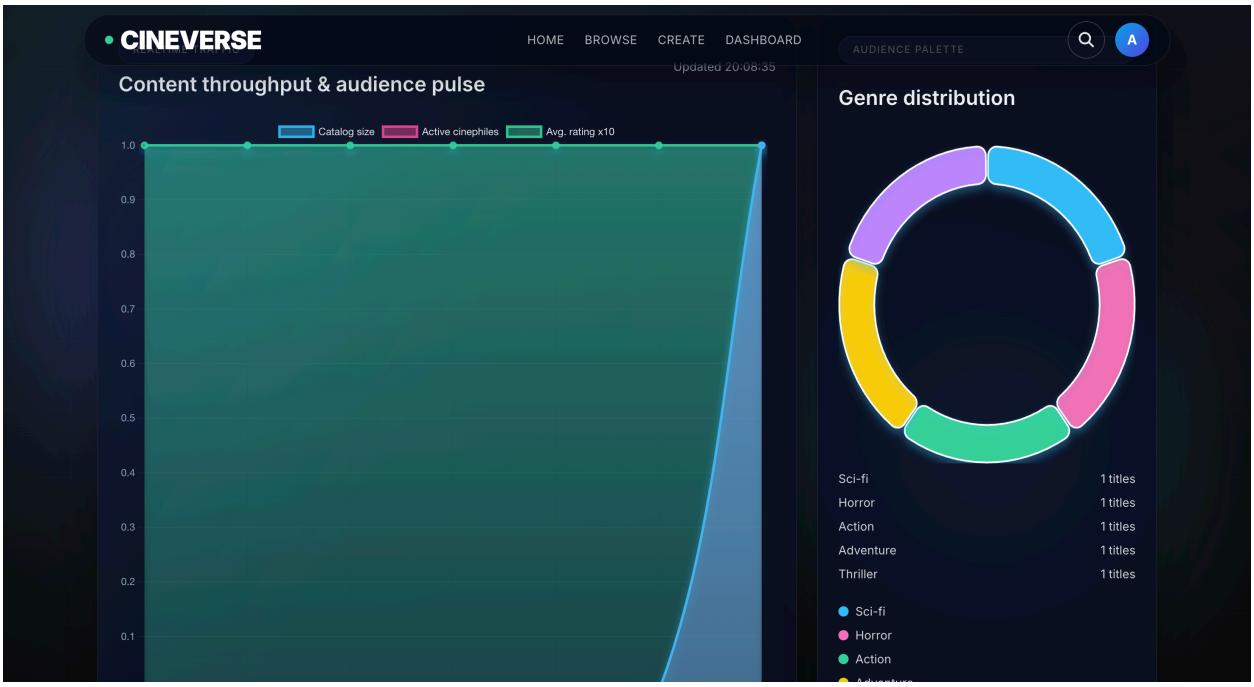
➤ The layered architecture (routes → controllers → services/models → sockets) allows future modules—such as additional analytics cards or external clients consuming the JWT API—to be added without touching core flows.

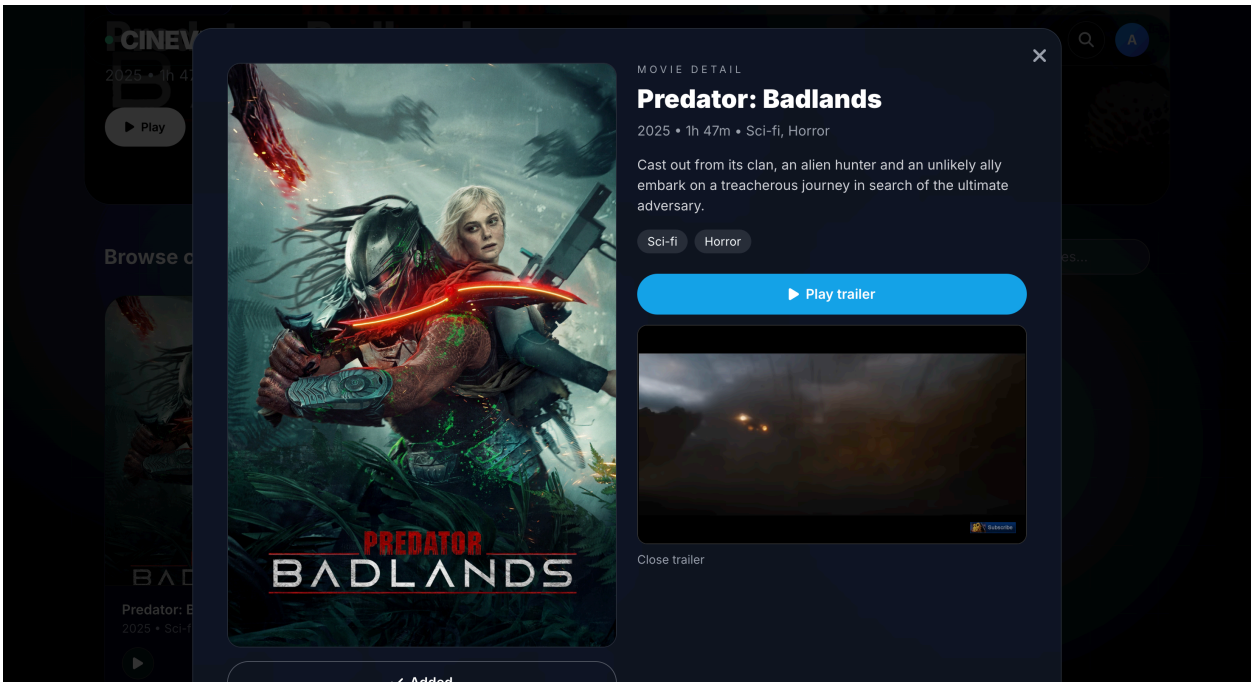
➤ Environment-driven configuration (.env) means the same code runs locally or in cloud deployments, and the event-based Socket.io bus can broadcast new message types as features expand.

4.3 Screenshots of Website






CINEVERSE
HOME
BROWSE
CREATE
DASHBOARD
AUDIENCE PALETTE
A
Content throughput & audience pulse
Updated 20:08:35
Catalog size
Active cinephiles
Avg. rating x10
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Genre distribution
Sci-fi 1 titles
Horror 1 titles
Action 1 titles
Adventure 1 titles
Thriller 1 titles
Sci-fi
Horror
Action


MOVIE DETAIL
Predator: Badlands
2025 • 1h 47m • Sci-fi, Horror
Cast out from its clan, an alien hunter and an unlikely ally embark on a treacherous journey in search of the ultimate adversary.
Sci-fi
Horror
Play trailer
Close trailer
PREDATOR
BADLANDS

REFERENCES

- Node.js Documentation — https://nodejs.org/en/docs/
- Express.js Documentation — https://expressjs.com/
- PostgreSQL Documentation — https://www.postgresql.org/docs/
- MongoDB & Mongoose Docs — https://www.mongodb.com/docs/ and https://mongoosejs.com/docs/
- Redis Documentation — https://redis.io/docs/latest/
- Socket.io Documentation — https://socket.io/docs/v4/
- Passport.js Guides — http://www.passportjs.org/docs/
- JSON Web Token (jwt.io) — https://jwt.io/introduction
- Tailwind CSS Documentation — https://tailwindcss.com/docs/installation
- Chart.js Documentation — https://www.chartjs.org/docs/latest/