System Evaluation & Scalability Report for EduLearn

1. Executive Summary

The current EduLearn system is a functional Monolithic application built with a modern tech stack (Next.js, Node.js/Express, PostgreSQL). While the codebase is clean and modular, the architecture suffers from significant scalability bottlenecks and performance inefficiencies that will severely impact users on low-bandwidth networks or during high-traffic periods.

Overall Rating: ⚠ **At Risk** (Requires immediate architectural improvements for production scalability)

---

2. System Architecture Evaluation

2.1 Tech Stack Analysis

Component	Technology	Status	Impact
Frontend	Next.js 14 (App Router)	✅ Strong	Good foundation for SSR/SEO, but currently underutilized for data fetching efficiency.
Backend	Node.js + Express	⚠ Moderate	Standard choice, but lacks a caching layer and relies on stateful file storage.
Database	PostgreSQL (Raw SQL)	⚠ Mixed	Raw SQL offers control but current implementation has N+1 queries and inefficient joins.
State Mgmt	React `useState`	❌ Weak	Lack of server-state management (React Query) leads to redundant network requests.

2.2 Database & Schema

Strengths:

- **Normalized Schema:** The database is well-structured with clear relationships.
- **Soft Deletes:** Implemented across major tables (`is_deleted`), preserving data integrity.
- **Basic Indexing:** Common access patterns (e.g., `email`, `role`) are indexed.

Weaknesses & Bottlenecks:

- **Missing Foreign Key Indexes:** Critical foreign keys used in joins (e.g., `course_modules.course_id`, `course_lessons.module_id`) appear unindexed in the migration script. This causes full table scans during joins.
- **Inefficient Join Patterns:** The `getCourseFiles` controller uses a fragile string matching join: `JOIN course_lessons cl ON substring(cl.video_url from '[^/]+$') = f.filename`. This is extremely slow and prevents index usage.
- **N+1 Query Problems:**
  - `getStudentCourses`: Contains a correlated subquery (`SELECT title FROM assignments ...`) that executes for *every single row* in the result set.

- `getCourseModules`: Uses `json_agg` with a subquery. While better than a loop, it fetches the entire course content tree in one massive query, straining DB memory.

2.3 Backend API & Performance

Strengths:

- **RBAC Middleware:** `requireRole` is consistently applied.
- **Slow Query Logging:** `db.ts` logs queries taking >1s, which is excellent for observability.

Weaknesses:

- **Stateful File Storage:** Files are stored locally in `uploads/`. This prevents horizontal scaling (adding more servers) because files won't be shared across instances.
- **No Caching Layer:** There is no Redis or in-memory caching. Every API request hits the database directly.
- **Over-fetching:** Endpoints like `getCourseModules` return *all* lessons, video URLs, and content for a course. For a large course, this payload could be several megabytes.

---

3. Low-Bandwidth & Network Analysis

The application is currently **poorly optimized** for low-bandwidth environments (2G/3G/Slow Wi-Fi).

3.1 Critical Issues

1. **Waterfall Data Fetching:**
   - `app/courses/[id]/page.tsx` fetches 4 separate endpoints (`modules`, `assignments`, `files`, `announcements`) in parallel. If any one fails or stalls, the page content may hang.
   - **Impact:** High latency and "pop-in" effect.
2. **No Client-Side Caching:**
   - The frontend uses `useState` and `useEffect` to fetch data. Every time a user navigates away and back (e.g., from Course -> Dashboard -> Course), **all data is re-fetched**.
   - **Impact:** Wasted bandwidth and frustratingly slow navigation.
3. **Heavy Payloads:**
   - The `getCourseModules` endpoint returns the full text content and video URLs for every lesson.
   - **Impact:** A user on a slow connection waits for the entire course structure to download just to see the first lesson.
4. **Unoptimized Media Delivery:**
   - Files are served directly from the Node.js server (`/uploads/...`). Node.js is not optimized for serving static assets, blocking the event loop.
   - No evidence of image optimization (CDNs, next/image usage is present but source is local).

---

4. Recommendations & Roadmap

4.1 Immediate Improvements (Short-Term)

1. **Fix N+1 Queries:**
   - Rewrite `getStudentCourses` to join assignments efficiently or fetch them in a separate parallel query.
2. **Add Missing Indexes:**
   - `CREATE INDEX idx_course_modules_course_id ON course_modules(course_id);`
   - `CREATE INDEX idx_course_lessons_module_id ON course_lessons(module_id);`
3. **Implement React Query:**
   - Replace `useState`/`useEffect` fetching in `app/courses/[id]/page.tsx` with the already configured `QueryProvider`.
   - **Benefit:** Instant load on navigation (cache-first), background refetching, and automatic retries.
4. **Enable Gzip/Brotli Compression:**
   - Ensure the backend Express server uses `compression` middleware to reduce JSON payload sizes by ~70%.

4.2 Architectural Enhancements (Long-Term)

1. **Migrate Files to Object Storage (S3/R2):**
   - Move uploads to AWS S3, Google Cloud Storage, or Cloudflare R2.
   - **Benefit:** Stateless backend (scalable), faster file delivery via CDN, reduced server load.
2. **Implement Redis Caching:**
   - Cache expensive queries (e.g., Course Structure, Categories) in Redis.
   - **Benefit:** Sub-millisecond response times for read-heavy data.
3. **Optimize API Payloads:**
   - Implement "Lazy Loading" for course modules. Fetch only the module titles first, then fetch lessons/content when a module is expanded.
4. **Database Connection Pooling:**
   - Use an external pooler (e.g., PgBouncer or Supabase Transaction Pooler) instead of just the internal `pg` pool to handle thousands of concurrent connections.

4.3 Monitoring & Reliability

- **Load Testing:** Run `k6` or `Artillery` scripts simulating 500+ concurrent users to identify breaking points.
- **Error Tracking:** Continue using Sentry (already configured).
- **APM:** Integrate New Relic or Datadog to visualize database query performance in real-time.

5. Conclusion

EduLearn has a solid code foundation but needs specific performance tuning to be production-ready. By implementing **client-side caching (React Query)** and **database indexing** immediately, you can drastically improve the user experience for low-bandwidth users. Moving file storage to the cloud is the critical next step for scaling beyond a single server.