Architectural Requirements

Quality Requirements

QR1: Usability:

- QR1.1 The system shall have a responsive UI that adjusts seamlessly across desktop, tablet, and mobile devices.
- QR1.2 The interface shall adhere to accessibility standards, including keyboard navigation and color contrast compliance.
- QR1.3 The platform shall provide visual cues (color indicators, feedback messages) to enhance user understanding and reduce confusion.
- QR1.4 Onboarding tutorials or walkthroughs shall be available to help new users understand key features.

QR2: Performance

- QR2.1 The system shall maintain a median page load time of under 2 seconds for all major views (e.g., dashboard, problem screen).
- QR2.2 The system shall support real-time updates (e.g., leaderboard changes, badge unlocks) using WebSockets with a latency of <300ms under average load.
- QR2.3 Cached content shall be used to reduce server requests for static resources and frequently accessed data (e.g., profile info, question metadata).

QR3: Scalability

- QR3.1 The backend infrastructure shall support horizontal scaling to handle at least 10,000 concurrent users without performance degradation.
- QR3.2 Microservices shall be independently deployable and scalable based on load (e.g., analytics service scales separately from the problem service).
- QR3.3 Load balancing shall be implemented to evenly distribute requests across services.

QR4: Reliability

- QR4.1 The system shall maintain 99.5% uptime per month, excluding scheduled maintenance.
- QR4.2 The system shall recover from microservice failure within 60 seconds using service isolation and fallback mechanisms.
- QR4.3 Daily backups of all relational and time-series data shall be automatically performed and stored securely.

QR5: Maintainability

- QR5.1 All backend services shall follow a consistent folder structure using the NestJS Controller-Service-Repository pattern.
- QR5.2 New features or modules shall be added using existing microservice conventions without requiring changes to unrelated services.
- QR5.3 Code documentation (using JSDoc) shall be provided for all public functions and modules.

QR6: Security

- QR6.1 All HTTP requests shall be transmitted over HTTPS using TLS 1.2+.
- QR6.2 All authentication shall follow the OAuth 2.0 + JWT standard, with access tokens valid for 1 hour and refresh tokens valid for 7 days.
- QR6.3 User passwords shall be hashed using bcrypt with at least 12 salt rounds before being stored.
- QR6.4 All API endpoints handling sensitive data shall require Bearer Token authorization and enforce role-based access control.
- QR6.5 The platform shall comply with POPIA regulations by:
    - Allowing users to view and delete their personal data.
    - Logging all access to sensitive user information.
    - Providing privacy policy consent before account creation.

QR7: Testability

- QR7.1 All backend services shall include unit and integration tests with at least 80% code coverage, measured via Jest.
- QR7.2 E2E (end-to-end) tests shall be implemented for user registration, login, and problem solving using Cypress.
- QR7.3 All CI pipelines shall fail builds when test coverage drops below the minimum threshold or when any critical test fails.

Architectural Patterns

ELO Learning will follow a Service-Oriented architecture, supported by RESTful APIs and WebSocket communication channels.

Service-Oriented Architecture:

The system is structured using a microservices architecture, where each core feature is implemented as an independent, loosely coupled service. This supports scalability, fault isolation, and independent development.

Each service exposes RESTful API endpoints, follows the service layer pattern internally, and is deployed as a standalone container (via Docker).

Services are orchestrated through internal API calls and WebSocket gateways, ensuring seamless interaction between modules.

The following core services have been identified:

Service	Description
Auth Service	Handles user registration, login, password hashing, JWT generation, and OAuth2 flow. Responsible for issuing and validating access tokens.
Matchmaking Service	Implements the ELO-based algorithm to assign math problems based on a user's current skill level. Continuously updates ratings after problem attempts.
Math Problem Service	Manages the storage, retrieval, and tagging of math problems. Supports filtering by difficulty, topic, and ELO rating.
Stats / Leaderboard Service	Computes and delivers leaderboard data, user ranks, and ELO histories. Pulls time-based metrics from InfluxDB for trend tracking.
User Profile Service	Stores and retrieves personal user data, such as name, avatar, progress, and achievements. Supports user dashboard and gamification views.
Analytics Service	(Optional/Planned) Logs performance data and interaction metrics for user feedback and system insights. May integrate with ML-based recommendation systems.

Each service is:

- Modular (partitioned) but not independently deployable (reliant on the ESB)
- Shares databases and allows other services to have access to it’s schemas
- Documented via Swagger or OpenAPI (to be included in developer documentation)*

A Client-Server Architecture:

The system follows a **client-server model**, separating the frontend application (React + Next.js PWA) from the backend (NestJS REST API and WebSocket Gateway).

The frontend communicates with the backend using **RESTful API calls** and **WebSockets** for real-time updates.

Services will be used to modularize core services such as:

- Authentication
- Matchmaking (ELO algorithm)
- Problem management
- Leaderboard and stats
- Analytics and feedback

Each service can be deployed, maintained, and scaled independently.

Additional services such as **background processing**, **ranking updates**, and **adaptive learning** are designed to run asynchronously.

ELO Learning adopts a service-oriented architecture rather than a traditional monolithic design. This architectural choice aligns with the project's goals for scalability, modularity, resilience, and long-term maintainability.

Aspect	Services	Monolith
Scalability	Individual services (e.g., Matchmaking, Analytics, Leaderboard) can scale independently based on load (e.g., elastic scaling in AWS/Azure).	Entire system must scale as one unit, leading to inefficient resource usage under uneven load.
Modularity	Each feature (Auth, ELO Engine, Content Management, etc.) lives in its own service, enabling cleaner separation of concerns and domain-driven design.	Tight coupling between modules increases the risk of regressions during updates.
Deployment	Enables partial deployments and independent versioning; only affected services need to be rebuilt or redeployed.	Requires full system redeployment even for small updates—slows down release cycle.
Team Autonomy	Backend services can be developed and tested by different sub-teams concurrently without waiting on unrelated features.	Single shared codebase makes parallel development harder and riskier.

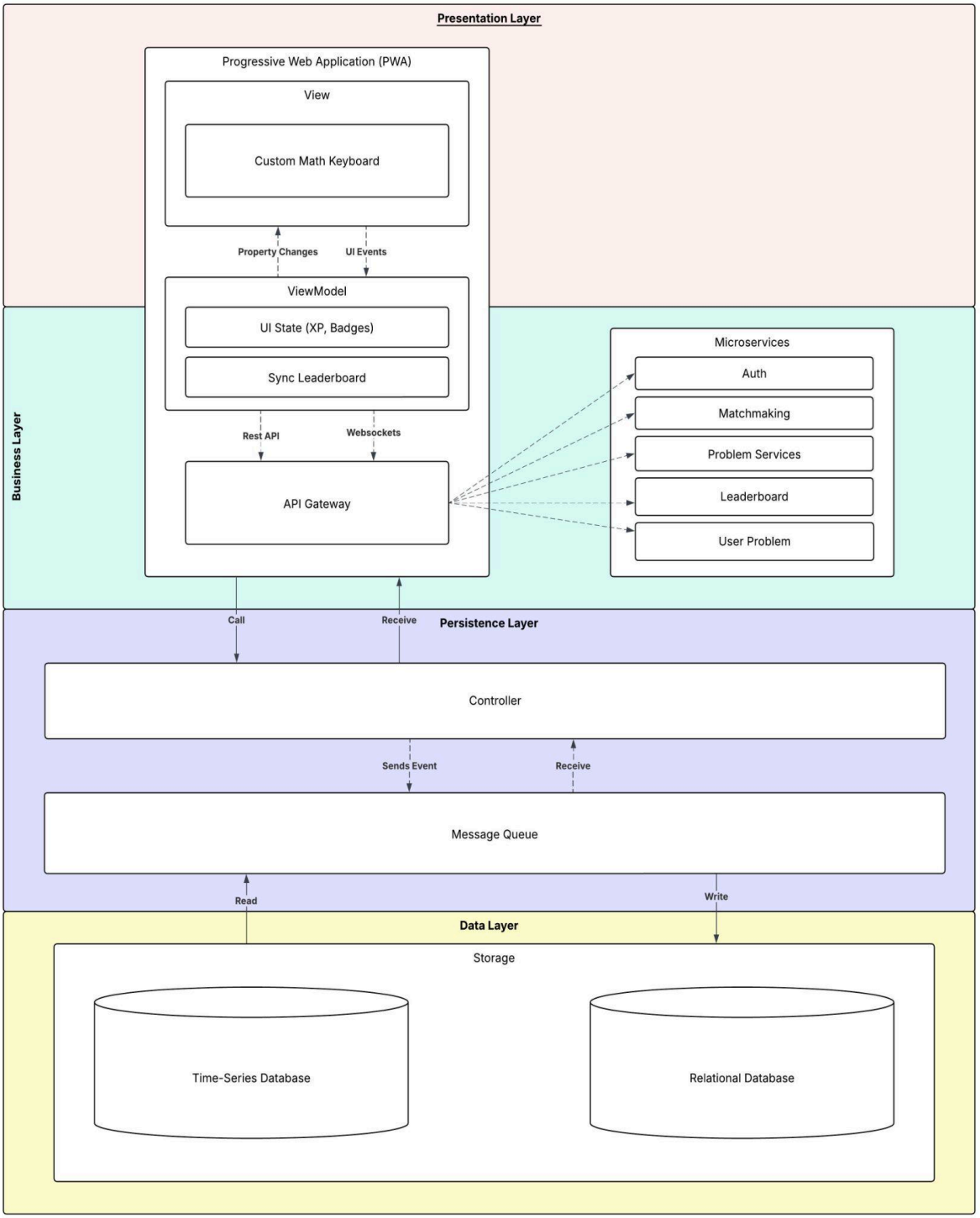
Fault Isolation	Failures in non-critical services (e.g., leaderboard) do not crash core learning functionality (e.g., question delivery, auth).	A crash in any module can bring down the entire system.
Technology Choice	Allows flexibility to introduce different tools/languages for specific services in the future (e.g., using Python ML models in Analytics while keeping NestJS core).	Single stack requirement across the whole system may limit flexibility or force compromises.
Long-Term Growth	Supports future expansion to new subjects (e.g., science, coding) via new services that plug into the ecosystem cleanly.	Monoliths become increasingly difficult to manage as scope and user base grow.

Given that ELO Learning is designed to be:

- **User-scalable** (supports thousands of concurrent learners),
- **Feature-rich** (matchmaking, analytics, gamification),
- **Continuously evolving** (future mobile apps, subject expansion, AI features),

the service-oriented architecture provides a **robust, future-proof foundation** that facilitates iterative development and rapid delivery without sacrificing maintainability or performance.

Architectural Diagram: Layered Overview of ELO Learning



ELO-Learning-Architectural-Diagram.pdf

Event-Driven Architecture (Optional):

The system supports **event-driven communication** for real-time updates (e.g., leaderboard updates, new problems).

WebSocket communication (NestJS Gateway) enables **live interaction** between users and the platform.

Future versions will incorporate:

- A **publish-subscribe (pub/sub)** model for broadcasting updates (e.g., using Redis Pub/Sub or RabbitMQ).
- Optional **event replay and recovery** features to support resilience and progress recovery.

Design Patterns

Strategy Pattern:
Used in the domain model to allow interchangeable problem selection and ranking logic(e.g., ProblemSelectionStrategy, RankingStrategy)

Singleton Pattern:
Ensures only one instance of each database connection per service (e.g., PostgreSQL, InfluxDB), using connection pooling.

Observer Pattern:
Enables real-time communication via WebSockets. For example, when a user completes a problem, the frontend is updated live with new ELO stats or leaderboard changes.

Mediator Pattern:
Used via a messaging queue to promote loose coupling by having different subsystems communicate with each other via a central mediator rather than directly with each other. This will make the components more independent by reducing dependencies between components, making the system more maintainable and scalable.

Service Layer Pattern:
In NestJs, our logic is organized into Controllers(for routing) → Services(for business logic) → Repositories(for data access). This ensures clean separation of routing, business logic and data access.

Data Transfer Objects Pattern:
DTOs are used to validate and structure incoming/outgoing data for all microservice endpoints.

Constraints

Time:
Three components must be demonstrated by Demo 2 (27 June 2025), limiting time for full system integration.

Security:
The platform must comply with POPIA and enforce secure authentication using OAuth 2.0 and JWT. All communication must occur over HTTPS with TLS encryption. User data must be stored securely and may not be exposed publicly. Backend services must be isolated to prevent unauthorized cross-service access.

Infrastructure:
The system must be containerized (Docker) and deployable to cloud platforms (AWS/Azure) using CI/CD pipelines.

Techstack

Use-case	Proposed Technologies and Frameworks
Frontend Development	React.js, Next.js (PWA)
Backend Development	ExpressJS
Containerization	Docker
Hosting & Infrastructure	AWS or Azure
Real-Time Communication	Native WebSocket integration (via NestJS Gateway)
Database (Core Data: Users, Problems)	Relational Database (PostgreSQL)
Database (Time-Series Data: Progress, Rankings)	Time-Series Database (InfluxDB)
Testing	Cypress and Jest
Version Control	Github and Docker
Documentation	JSDocs and Markdown (Github)
DevOps & Deployment	Github Actions (CI/CD Pipelines)
Security	Secure data transmission (HTTPS - TLS encryption)
User Authentication	Token-Based Security (OAuth 2.0 / JWT)

Custom Math Keyboard/Calculator

To improve mathematical input and support the platform's gamified learning flow, the team has implemented a custom-built math keyboard and input field, inspired by platforms like Mathway and Symbolab.

Key Features:

- Interactive, on-screen keyboard with symbols for:
  - Exponents, square roots, fractions, integrals
  - Basic arithmetic operators
  - Trigonometric functions
- Real-time input via:
  - Virtual keyboard clicks
  - Physical keyboard typing
- Supports LaTeX-style syntax and math rendering

Technology Stack:

Use case	**Technology**
Math input field	MathLive (custom wrapped in React component)
Real-time preview	KaTeX (via react-katex) for lightweight math rendering
Optional renderer fallback	MathJax (for advanced layout and accessibility support)
Backend parsing (optional)	math.js for symbolic expression evaluation and backend grading
Styling & layout	TailwindCSS or component-level styling via React

System Flow:

- Student enters input using the math keyboard.
- The LaTeX expression is previewed live using KaTeX.
- After submission, the backend may receive and evaluate the expression. And then ELO score updates and problem feedback are returned.

Technology choices

1. Frontend Development: React.js and Next.js (PWA)

- **React.js** offers a component-based, reusable structure ideal for complex UI development.
- **Next.js** provides server-side rendering (SSR) and PWA capabilities, improving performance and SEO.
- The PWA (Progressive-Web-App) approach ensures mobile responsiveness and offline usability, key for educational accessibility.

Alternatives:

1. **Vue.js & Nuxt.js** offer SSR and a great development experience, but React has a larger ecosystem and team familiarity is often higher.
2. **SvelteKit** is Lightweight and fast with simple syntax, but lacks the orthogonality and third-party support that React/Next.js provide.

React & Next.js strike a balance between performance, ecosystem, and long-term maintainability.

2. Backend Development: NestJS (Built on ExpressJS)

- **NestJS** adds structure (controllers, services, modules) on top of Express, aligning perfectly with your Controller-Service-Repository pattern.
- Strong TypeScript support, dependency injection, and built-in testability.
- Ideal for microservices with modular architecture.

Alternatives:

1. **Express.js (raw)** is more flexible but lacks NestJS's opinionated architecture and built-in structure.
2. **Spring Boot (Java)** is excellent for enterprise apps but less writable and more complicated, with steeper learning curve and slower iteration speed.

NestJS enables rapid, scalable development while maintaining clean code separation and testability.

3. Real-Time Communication: NestJS WebSocket Gateway

- Tight integration with your existing NestJS services.
- Scales well for features like live leaderboards, ELO updates, and collaboration. Supports event-driven architecture and observer patterns.

Alternatives:

1. **Socket.IO** (standalone) has more powerful real-time tools, but additional integration overhead, which could prove to be challenging for a small team without designated integration engineers.

2. **Firebase Realtime Database** is an easier setup for small apps but less flexible, vendor-locked, and not ideal for backend-heavy logic. ELO learning is in its genesis phase, this would make scalability painfully difficult.

NestJS Gateway keeps everything under a unified framework, making real-time communication more manageable.

4. Database (Relational): PostgreSQL

- Robust, open-source SQL database with strong ACID compliance.
- Excellent for complex queries, indexing, and analytics.
- Supports JSON for semi-structured data (e.g., user metadata).

Alternatives:

1. **MySQL** is also relational, but PostgreSQL is generally more feature-rich and performant for analytics-heavy workloads. PostgreSQL is also easier and cheaper to link with services provided by platforms like cloudflare.

2. **MongoDB** is great for flexible schemas, but less suitable for consistent, transactional data like ELO ratings or problem metadata.

PostgreSQL balances performance, structure, and flexibility which are perfect for education-based applications. Experts online and alike from platforms like ITSI recommended this approach.

5. Time-Series Data: InfluxDB

- Optimized for time-series metrics is ideal for tracking user progress, score changes, and trends over time.
- High write throughput and efficient retention policies.

Alternatives:

1. **Prometheus** is strong for monitoring metrics, but less suited for user-generated educational data.
2. **TimescaleDB** is built on PostgreSQL and more SQL-friendly, but slightly heavier and potentially overlapping with your main DB.

InfluxDB is specialized for time-series needs without bloating your relational layer.

6. Authentication: OAuth 2.0 and JWT

- Industry-standard for secure, stateless authentication.
- Works well for token-based sessions across microservices.
- Refresh tokens provide a smooth UX for long sessions.

Alternatives:

1. **Firebase Auth** is easier to integrate but vendor-locked (you need to rely heavily on Firebase and just run with what they have available) and less customizable.
   Session-based Auth is simpler for monoliths, but less scalable and stateless for distributed systems.

OAuth and JWT supports secure, scalable auth for SOA architecture and aligns with modern best practices.

7. DevOps: Docker and GitHub Actions (CI/CD)

Why you chose it:

- Docker enables consistent environments and microservice containerization.
- GitHub Actions provides simple, powerful CI/CD directly integrated with your repo.

Alternatives:

1. **Jenkins** – Mature, but more complex and harder to maintain for smaller teams.
2. **CircleCI** – Excellent CI/CD service but introduces another platform to manage.

Docker and GitHub Actions offer quick setup, seamless integration, and simplicity for small-to-mid scale dev teams. Docker is also more widely used in the industry and its familiarity is a more valuable skill than any of the other platforms.

8. Math Input: MathLive + KaTeX + math.js

- **MathLive** offers a rich virtual math keyboard with LaTeX support.
  KaTeX provides fast, high-quality math rendering.
- **math.js** enables backend symbolic evaluation and grading logic.

Alternatives:

1. **Desmos API** is great for graphing but not built for full math input workflows.

2. **Quill.js & MathQuill plugin** have decent math input, but harder to customize for deep gamified workflows.

Our stack offers high control, performance, and extensibility, aligned with our educational goals.