ELO Learning

ARCHITECTURAL REQUIREMENTS V2

ZERO DAY

Last Updated: 26 June 2025

University of Pretoria

Name	Student number
RM (Rene) Brancon	u22556771
NF (Nigel) Mofati	u22528084
TM (Tukelo) Mokwena	u22536800
S (Saskia) Steyn	u17267162
NG (Ntokozo) Tonga	u22506773

Team contact:
ZeroDay0D4y@gmail.com

Contents

1. Quality Requirements

QR1: Usability

- QR1.1 The system shall have a responsive UI that adjusts seamlessly across desktop, tablet, and mobile devices with 100% viewport compatibility testing.
- QR1.2 The interface shall adhere to WCAG 2.1 AA accessibility standards, including keyboard navigation and color contrast compliance with a minimum contrast ratio of 4.5:1.
- QR1.3 The platform shall provide visual cues (color indicators, feedback messages) to enhance user understanding with a measurable task completion rate of >85%.
- QR1.4 Onboarding tutorials shall achieve >90% completion rate and reduce time-to-first-successful-problem-attempt to under 3 minutes.

Client–Server Support for Usability:
Our client–server architecture directly enhances usability by separating user-facing clients from backend servers. The client applications handle all interactions and presentation, ensuring a responsive and optimized user interface, while the server maintains consistent user preferences, progress, and mathematical content. This separation ensures that mathematical problems and user data render uniformly regardless of which client device or platform is used. Standardized server responses allow clients to provide a consistent user experience across different areas of the platform. By isolating user experience responsibilities to the client and data management to the server, usability is improved: clients can focus on delivering smooth interactions while servers evolve their internal logic and storage strategies without disrupting the end-user experience.

QR2: Performance

- QR2.1 The system shall maintain a median page load time of under 2 seconds for all major views measured via synthetic monitoring.
- QR2.2 The system shall support real-time updates using WebSockets with latency <300ms under normal load (up to 1,000 concurrent users).
- QR2.3 Cached content shall reduce server requests by 70% for static resources and frequently accessed data.

Client–Server Support for Performance:
The client–server architecture enables performance optimization by separating user-facing interactions from backend processing. The **frontend on Vercel** delivers static assets through a global CDN, ensuring fast load times and responsive user experiences regardless of user location. The **backend on Render** can scale independently, allowing critical services such as authentication, math problem delivery, or leaderboard updates to handle peak loads without slowing down the client. Backend-level caching strategies (e.g., caching

frequently accessed math problems or precomputed leaderboard data) reduce computational overhead and improve response times. This separation ensures that heavy backend workloads do not impact frontend responsiveness, while database optimization and the use of read replicas allow efficient handling of both write-intensive and read-heavy operations.

QR3: Scalability

- QR3.1 The backend infrastructure shall support horizontal scaling to handle at least 10,000 concurrent users without >10% performance degradation.
- QR3.2 Services shall be independently scalable based on load patterns with automatic scaling triggers at 70% resource utilization.
- QR3.3 Load balancing shall distribute requests with <5% variance across available service instances.

Client–Server Support for Scalability:
The client–server architecture supports scalability by separating presentation from backend processing and enabling independent scaling of each layer. The **frontend on Vercel** leverages global CDN replication to handle sudden spikes in user traffic with minimal latency. The **backend on Render** can scale horizontally by running multiple service instances behind load balancers, ensuring that high-demand features such as authentication, math problem delivery, or leaderboard updates remain responsive during peak usage periods. This separation prevents resource contention between frontend delivery and backend computation, while database read replicas distribute query loads to support growing numbers of concurrent users. Together, these mechanisms allow the system to scale smoothly without performance degradation.

QR4: Reliability

- QR4.1 The system shall maintain 99.5% uptime per month, excluding planned maintenance windows.
- QR4.2 The system shall recover from service failure within 60 seconds using health checks and circuit breaker patterns.
- QR4.3 Daily automated backups shall maintain 99.9% data integrity with point-in-time recovery capability.

Client–Server Support for Reliability:
The client–server architecture improves system reliability by separating frontend delivery from backend operations. If the **backend on Render** experiences partial outages (e.g., leaderboard or analytics endpoints failing), the **frontend on Vercel** continues to serve the application interface, ensuring learners still have access to core functionality such as

authentication and math problem delivery. Backend health monitoring and auto-restart mechanisms on Render provide targeted recovery without requiring full system downtime. Standardized API error handling allows the frontend to implement graceful degradation—for example, if leaderboard data is temporarily unavailable, users can continue solving problems while receiving clear feedback about unavailable features. Database backup and recovery strategies prioritize critical services such as authentication and user profiles, ensuring that essential educational functionality remains reliable even under fault conditions.

QR5: Security

- QR5.1 All HTTP requests shall be transmitted over HTTPS using TLS 1.3+.
- QR5.2 Authentication shall follow OAuth 2.0 + JWT standard with 1-hour access tokens and 7-day refresh tokens.
- QR5.3 User passwords shall be hashed using bcrypt with minimum 12 salt rounds.
- QR5.4 All API endpoints shall enforce role-based access control with Bearer Token authorization.
- QR5.5 POPIA compliance through user data management capabilities and consent mechanisms.

Client–Server Support for Security:
The client–server architecture enforces robust security by centralizing authentication and access control on the **backend deployed on Render**, while the **frontend on Vercel** handles only presentation and user interaction. The backend issues **JWT tokens** for authenticated users, which the frontend includes in API requests to access protected resources, ensuring consistent security across all interactions. Role-based access control restricts operations based on user permissions—for example, only administrators can edit math problems, while students can access their own progress data. All communication between the frontend and backend uses secure protocols (HTTPS), protecting data in transit. This separation provides security compartmentalization: even if one component faces a vulnerability, the rest of the system remains protected. Backend security auditing and monitoring allow targeted incident detection and response without affecting frontend availability.

QR6: Maintainability

- QR6.1 All backend services shall follow consistent NestJS Controller-Service-Repository pattern.
- QR6.2 Code coverage shall maintain a minimum 80% measured via Jest testing framework.
- QR6.3 JSDoc documentation shall be provided for all public functions and modules.

Client–Server Support for Maintainability:
The client–server architecture enhances maintainability through a clear separation of concerns between frontend and backend. The **frontend on Vercel** focuses solely on user interface and experience, while the **backend on Render** handles all business logic, data access, and authentication. Developers can modify or extend backend functionality—such as the ELO matching algorithm or math problem logic—without impacting the frontend, and vice versa. Well-defined API contracts between frontend and backend prevent breaking changes, enabling confident refactoring and updates within each layer.

Maintainability is further supported through modular backend design, independent deployment, and versioning of backend services. Changes to backend logic, adoption of new libraries, or optimization of algorithms can occur without requiring frontend modifications. Database design with read replicas and structured access patterns simplifies maintenance and ensures data consistency. Additionally, clear documentation, testing, and role-specific maintenance allow specialists to focus on their areas of expertise—frontend developers handle UI improvements while backend developers maintain core logic and data services—resulting in higher quality updates and more efficient bug resolution.

QR7: Testability

- QR7.1 Backend services shall include unit and integration tests with >80% code coverage.
- QR7.2 E2E tests shall cover critical user journeys using Cypress with >95% test pass rate.
- QR7.3 CI pipelines shall fail builds when coverage drops below thresholds or critical tests fail.

Client–Server Support for Testability:
The client–server architecture enhances testability by clearly separating frontend and backend responsibilities. The **backend on Render** can be unit tested independently of the frontend using mock requests and database stubs, enabling faster and isolated test development. API contracts between frontend and backend ensure that changes in backend logic do not break frontend functionality without explicit versioning.

Testing Strategies:

- **Backend Unit Testing:** Test individual backend modules, services, and business logic in isolation using mocks or stubs for external dependencies such as the database.

- **API Contract Testing:** Validate that backend endpoints adhere to agreed request/response formats, ensuring frontend compatibility.

- **Integration Testing:** Verify that backend components work correctly together, including authentication flows, problem delivery, and leaderboard updates.

- **Frontend Testing:** Test UI components and their interactions with mock or staging backend APIs to ensure consistent user experience.

- **End-to-End Testing:** Simulate full user workflows, from frontend interactions through backend processing and database access, validating that the entire system behaves as expected.

This separation allows both frontend and backend to be tested independently or together, supporting efficient development, debugging, and deployment cycles.

2. Architectural Design Strategy

For ELO Learning, we have chosen a design strategy based on quality requirements as our primary architectural approach. This strategy prioritizes the systematic achievement of our most critical quality attributes—usability, performance, and scalability—which directly impact the educational effectiveness of our platform.

Justification for Quality-Driven Design Strategy

The quality-driven approach is most suitable for ELO Learning because educational platforms must excel in user experience and performance to maintain student engagement. Unlike purely functional-driven design, this strategy ensures that architectural decisions directly support measurable learning outcomes. The strategy involves:

1. **Quality Attribute Scenarios:**
   Each quality requirement is expressed as testable scenarios that drive architectural decisions.
2. **Tactic Selection:**
   Specific architectural tactics are chosen to address quality attribute requirements systematically.
3. **Pattern Application:**
   Architectural patterns are selected based on their ability to satisfy our prioritized quality attributes.
4. **Iterative Refinement:**
   The architecture evolves through continuous measurement against quality targets.

3. Architectural Strategies

Our architectural strategies directly address our quality requirements through specific tactics and techniques:

Performance Strategy: Asynchronous Processing and Caching

- **Tactics:**
  Manage resources, increase available resources, reduce overhead
- **Implementation:**
  WebSocket-based real-time updates, Redis caching layer, CDN for static assets
- **Quality Target:**
  <2 second page load times, <300ms real-time update latency

Scalability Strategy:
Horizontal scale-out with distributed load across frontend and backend services.

Tactics:

- **Multiple copies of frontend**: Vercel automatically replicates and serves the frontend from its global CDN, ensuring fast responses and high availability.
- **Multiple copies of backend**: Render can run multiple instances of the backend service, distributing computation and API requests.
- **Multiple copies of data**: Using database read replicas to spread read-heavy workloads while keeping a primary database for writes.

Implementation:

- **Frontend:** Deployed on Vercel with built-in CDN replication to handle user requests worldwide.
- **Backend:** Deployed on Render with containerized services and horizontal autoscaling of backend instances based on demand.
- **Database:** Scaled with read replicas to balance query loads and improve responsiveness.
  Quality Target:
  Support **10,000+ concurrent users** with near-linear scaling across frontend and backend layers.

Usability Strategy: Real-time UI Responsiveness

- **Tactics:**
  Maintain task model, maintain system model, maintain user model

- **Implementation:**
  Progressive Web App architecture, responsive design patterns, accessibility-first development
- **Quality Target:**
  >85% task completion rate, <3 minute onboarding time

Security Strategy: Defense in Depth with Token-based Authentication

- **Tactics:**
  Authenticate users, authorize users, maintain data confidentiality
- **Implementation:**
  OAuth 2.0 + JWT, HTTPS/TLS 1.3, RBAC, input validation
- **Quality Target:**
  Zero security incidents, full POPIA compliance

Availability Strategy: Fault Detection and Recovery

- **Tactics:**
  Fault detection, fault recovery, fault prevention
- **Implementation:**
  Health checks, circuit breakers, automated failover, backup systems
- **Quality Target:**
  99.5% uptime with <60 second recovery time

4. Architectural Patterns

Primary Pattern: Client- Server Architecture

ELO Learning employs a Client- Server pattern as its primary architectural approach. This represents an evolution from our initial microservices and later SOA consideration, adapted to better suit our project's specific constraints and requirements.

Migration from SOA to Client–Server: Justification

After our migration from **Microservices**, we implemented a **Service-Oriented Architecture (SOA)** to gain modularity and separation of concerns across backend services. However, after reviewing our project constraints, team size, and deployment needs, we transitioned to a **Client–Server architecture** for the following reasons:

Complexity Management:
SOA introduced overhead with multiple service boundaries, service-specific deployment, and inter-service API management. A client–server model reduces this complexity by consolidating core backend logic into a single server, allowing the team to focus on feature development rather than managing multiple service lifecycles.

Resource Constraints:
Maintaining multiple SOA services required significant DevOps infrastructure and monitoring. With limited infrastructure and budget, a single backend deployed on **Render** is simpler, more cost-effective, and easier to maintain.

Integration Simplicity:
The client–server model centralizes business logic and database access, removing the "distributed system tax" of SOA. Frontend clients (deployed on **Vercel**) interact with the backend via well-defined APIs, streamlining integration and reducing points of failure.

Team Expertise:
Our team's experience with monolithic and service-based patterns made transitioning to a unified backend more natural than continuing with a distributed SOA.

Client–Server Implementation Details

Architecture Characteristics:

- **Frontend (Vercel):** Handles all user interface and experience logic, delivering content through a global CDN for fast access.
- **Backend (Render):** Consolidates core business logic, including authentication, problem management, leaderboard calculations, and analytics.
- **Communication:** Frontend and backend interact via REST APIs and secure HTTP protocols.
- **Database:** Centralized database with read replicas for performance and reliability.

Backend Responsibilities (formerly separate services):

- **Auth & User Profile:** User registration, login, JWT issuance, progress tracking, and achievements.
- **Math Problem & Matchmaking:** Problem storage, retrieval, and difficulty matching using the ELO algorithm.

- **Stats/Leaderboard & Analytics:** Ranking calculations, leaderboard delivery, and interaction metrics logging.

Benefits Realized

- **Simplified Integration:** A single backend simplifies API contracts and reduces inter-service communication overhead.

- **Manageable Deployment:** Backend is deployed independently, while the frontend can scale via Vercel's CDN.

- **Development Velocity:** Team members can focus on features across frontend and backend without coordinating multiple service deployments.

- **Quality Assurance:** Testing is easier with a single backend, while frontend testing can be done independently using mocked API responses.

Tradeoffs Accepted

- **Reduced Modularity:** Consolidating services reduces the strict separation between previously independent SOA services.

- **Shared Backend Logic:** Updates to one backend module may require coordination to avoid impacting other functionality.

- **Single Deployment Point:** While easier to manage, the backend represents a single point of failure compared to fully independent SOA services.

Secondary Pattern: Model-View-Controller (MVC) for Frontend

The frontend implementation follows the Model-View-Controller (MVC) pattern, implemented through React's component architecture with Next.js:

Model:
Application state management through React hooks and context, representing user data, problem state, and UI state View: React components that render the user interface, including the custom math keyboard and problem displays

Controller:
Event handlers and business logic that coordinate between user interactions and state updates

This pattern supports our usability quality requirements by providing clear separation between presentation and logic, enabling consistent UI behavior and easier maintenance of the complex mathematical input interfaces.

Supporting Patterns

Observer Pattern:
Implemented through WebSocket connections for real-time leaderboard updates and progress notifications. This pattern directly supports our performance quality requirements for <300ms real-time updates.

Strategy Pattern:
Used in the matchmaking service to allow different ELO calculation strategies and problem selection algorithms. This supports future extensibility without architectural changes.

Service Layer Pattern:
Implemented in the backend using an **Express.js router–service–database structure**, separating request handling, business logic, and data access. This supports maintainability by keeping concerns isolated, allowing the backend to evolve without impacting the frontend deployed on Vercel.

Mediator Pattern:
Applied through our API Gateway pattern, which coordinates communication between frontend and backend services while providing security and monitoring capabilities.

5. Architectural Constraints

Technical Constraints

- **Deployment:**
  The **frontend** is deployed on **Vercel**, which automatically builds and distributes the application through its global CDN. The **backend** is deployed on **Render**, which manages service instances and scaling, providing consistent deployment across environments without requiring manual containerization.
- **Cloud Platform:**
  System must deploy to Render using infrastructure-as-code principles
- **Database Technology:**
  Must use PostgreSQL for relational data and InfluxDB for time-series analytics data
- **Security Standards:**
  Must implement HTTPS/TLS 1.3, OAuth 2.0 + JWT authentication, and POPIA compliance

Project Constraints

- **Demo Timeline:**
  Seven components must be fully implemented by 20 August,2025, limiting architectural complexity
- **Team Size:**
  Architecture must be manageable by a small development team without dedicated DevOps engineers
- **Budget Limitations:**
  Infrastructure costs must remain within educational project constraints

Regulatory Constraints

- **POPIA Compliance:**
  User data handling must comply with South African privacy regulations
- **Educational Standards:**
  Math content and progress tracking must support pedagogical best practices
- **Accessibility Requirements:**
  Interface must meet WCAG 2.1 AA standards for inclusive education

Integration Constraints

- **Math Input Complexity:**
  Architecture must support complex mathematical notation input and rendering
- **Real-time Requirements:**
  Must support WebSocket connections for immediate feedback and collaborative features
- **Progressive Web App:**
  Must function as PWA for mobile accessibility without native app development

6. Technology Choices

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

Alternatives Considered:

- **Vue.js & Nuxt.js:**
  Strong SSR capabilities and gentle learning curve
- **SvelteKit:**
  Excellent performance with minimal bundle size

- **React.js & Next.js:**
  (Selected) Mature ecosystem with comprehensive PWA support

Selection Justification:
React + Next.js provides the best balance of development velocity, PWA capabilities, and ecosystem support for complex mathematical interfaces. The mature component ecosystem includes specialized math rendering libraries that directly support our usability quality requirements.

Backend Development: Express.js

Alternatives Considered:

- **Express.js:**
  (Selected) Lightweight, flexible Node.js framework with extensive middleware ecosystem
- **NestJS:**
  Structured TypeScript framework with built-in dependency injection and testing
- **Spring Boot (Java):**
  Enterprise-grade framework with comprehensive features but steeper learning curve

Selection Justification:
Express.js was chosen for its simplicity and rapid development capabilities, which align perfectly with our project timeline and team expertise. While NestJS offers more structure through its opinionated architecture, Express.js provides the flexibility needed to implement our SOA pattern without the overhead of learning a complex framework.

The extensive middleware ecosystem allows us to add exactly the features we need for authentication, WebSocket support, and API routing without unnecessary complexity. This choice directly supports our performance quality requirements through minimal overhead and our maintainability requirements through the team's existing familiarity with Express.js patterns. The framework's lightweight nature also supports our scalability goals by reducing resource consumption per service instance.

Database Strategy: PostgreSQL + InfluxDB

Alternatives Considered:

- **MongoDB + PostgreSQL:**
  NoSQL flexibility with relational consistency

- **MySQL + Prometheus:**
  Standard relational with monitoring-focused time-series
- **PostgreSQL + InfluxDB:**
  (Selected) Robust relational with specialized time-series capabilities

Selection Justification:
PostgreSQL provides the ACID compliance needed for user data and ELO calculations, while InfluxDB offers optimized time-series storage for learning analytics. This combination directly supports our performance and scalability quality requirements.

Real-Time Communication: NestJS WebSocket Gateway

Alternatives Considered:

- **Socket.IO (standalone):**
  Feature-rich but requires additional integration overhead
- **Firebase Realtime Database:**
  Easy setup but vendor lock-in concerns
- **NestJS WebSocket Gateway:**
  (Selected) Integrated with existing backend architecture

Selection Justification:
Native integration with our SOA services eliminates additional complexity while providing the <300ms latency required by our performance quality requirements.

Authentication: OAuth 2.0 + JWT

Alternatives Considered:

- **Firebase Auth:**
  Simplified implementation but vendor dependency
- **Session-based Auth:**
  Traditional approach but limited scalability
- **OAuth 2.0 + JWT:**
  (Selected) Industry standard with scalable token-based architecture

Selection Justification:
Provides the security requirements while supporting our SOA pattern's stateless service communication. The standard approach ensures long-term maintainability and compliance with security best practices.

7. Custom Math Keyboard Implementation

Architecture Decision

The custom math keyboard represents a critical architectural component that directly impacts our highest-priority quality requirement: usability. Rather than relying on external services or complex integrations, we've architected an integrated solution that provides seamless mathematical input within our educational platform.

Technology Stack

- **MathLive:**
  Provides the interactive math keyboard with LaTeX support, enabling complex mathematical notation input
- **KaTeX:**
  Handles real-time math rendering with superior performance compared to MathJax
- **math.js:**
  Enables backend expression evaluation and automated grading capabilities
- **React Integration:**
  Custom wrapper components that integrate mathematical input with our MVC frontend pattern

Architectural Benefits

This integrated approach supports multiple quality requirements simultaneously: it enhances usability through intuitive math input, improves performance through optimized rendering, and maintains security through controlled input validation. The architecture ensures that mathematical notation handling remains consistent across all problem types while supporting future extensibility for advanced mathematical concepts.

8. Summary

The revised architecture for ELO Learning represents a carefully balanced approach that prioritizes our critical quality requirements while remaining practical for our team and timeline constraints. The migration from microservices to SOA reflects architectural maturity—choosing the right tool for the job rather than following trends. Our quality-driven design strategy ensures that every architectural decision directly supports measurable educational outcomes, while our comprehensive technology stack provides the foundation for a scalable, maintainable learning platform.

The architecture successfully addresses the tension between educational effectiveness and technical complexity, providing a robust foundation that can evolve with our platform's

growth while maintaining the performance and usability standards that educational success requires.