ELO Learning - Architectural Requirements v2

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.

SOA Support for Usability: Our service-oriented architecture directly enhances usability through service specialization and loose coupling. The User Profile Service maintains consistent user preferences and progress across all interactions, while the Math Problem Service ensures mathematical content renders uniformly regardless of which frontend component requests it. Service interfaces provide standardized response formats that enable consistent user experience patterns across different platform areas. The separation of concerns allows the frontend to focus entirely on user experience optimization without being constrained by backend complexity, while services can evolve their internal implementations to better support usability requirements without affecting other system components

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.

SOA Support for Performance: The service-oriented architecture enables performance optimization through independent service scaling and specialized resource management. The Stats/Leaderboard Service can be scaled independently during peak usage periods without affecting the Auth Service or Math Problem Service performance. Service-level caching strategies allow each service to optimize its data access patterns—the Math Problem Service implements aggressive caching for frequently accessed problems, while the Matchmaking Service caches ELO calculations to reduce computational overhead. The

loose coupling between services prevents performance bottlenecks in one service from cascading to others, and service-specific database optimization ensures that each service can tune its data access patterns for optimal performance without constraints from other system components.

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.

SOA Support for Scalability: Service-oriented architecture provides exceptional scalability advantages through independent service scaling and resource allocation. Each service can be scaled based on its specific usage patterns—during peak learning hours, the Matchmaking Service may require more instances to handle problem assignment requests, while the Analytics Service can maintain minimal resources until batch processing periods. The service boundaries prevent resource contention, allowing high-demand services to scale without affecting the resource allocation of stable services like the Auth Service.
Service-level load balancing ensures that scaling decisions for one service don't impact the availability or performance of other services, and the shared database approach allows services to scale their computational resources independently while maintaining data consistency across the platform.

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.

SOA Support for Reliability: The service-oriented architecture enhances system reliability through fault isolation and service independence. When individual services experience issues, the failure is contained within service boundaries—if the Analytics Service encounters problems, students can continue learning through the Math Problem Service and Matchmaking Service without interruption. Each service implements its own health monitoring and recovery mechanisms, allowing targeted recovery procedures that don't require full system restarts. The service interfaces provide standardized error handling that enables graceful degradation—if the Leaderboard Service is temporarily unavailable, the core learning functionality continues while users receive appropriate feedback about temporarily unavailable features. Service-level backup strategies ensure that critical services like Auth and User Profile have priority recovery procedures, while less critical services can be restored without impacting core educational functionality.

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.

SOA Support for Security: Service-oriented architecture provides robust security advantages through service-level access control and security boundary enforcement. The Auth Service acts as a centralized security authority, issuing JWT tokens that other services validate independently, creating a consistent security model across all platform interactions. Each service implements role-based access control tailored to its specific domain—the Math Problem Service restricts problem editing to administrators while allowing read access to authenticated students, and the User Profile Service ensures users can only access their own progress data. Inter-service communication follows secure protocols with service-to-service authentication, preventing unauthorized access between internal components. The service boundaries create security compartmentalization where a potential vulnerability in one service doesn't automatically compromise other services, and service-specific security auditing enables targeted security monitoring and incident response procedures.

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.

SOA Support for Maintainability: Service-oriented architecture provides exceptional maintainability advantages through clear separation of concerns and modular development practices. Each service maintains its own codebase with well-defined boundaries, allowing developers to understand, modify, and extend individual services without needing to comprehend the entire system complexity. When modifications are required for the ELO matching algorithm, developers can focus exclusively on the Matchmaking Service without worrying about unintended effects on the Auth Service or User Profile Service. The service interfaces act as contracts that prevent breaking changes from propagating across service boundaries, enabling confident refactoring within individual services.

Service-level maintainability is further enhanced through independent deployment and versioning capabilities. Each service can evolve its internal implementation, adopt new libraries, or optimize its algorithms without coordinating changes across the entire platform. The shared database approach, while creating some coupling, actually improves maintainability for educational platforms by ensuring data consistency and simplifying backup and recovery procedures. Service-specific documentation and testing strategies

mean that maintenance work can be performed by developers who specialize in particular domains—authentication experts can maintain the Auth Service while education specialists focus on the Math Problem Service, leading to higher quality maintenance and more effective 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.

SOA Support for Testability: Service-oriented architecture significantly enhances testability through service isolation and interface standardization. Each service can be unit tested independently using mock implementations of dependent services, enabling parallel test development and faster test execution. Service interface testing validates API contracts between services, ensuring that changes to one service don't break dependent services without explicit interface versioning. The architecture supports comprehensive integration testing strategies including service-to-service contract testing, where each service interface is tested against defined contracts to ensure compatibility. Mock service implementations allow isolated testing of individual services without requiring the full system to be operational, enabling efficient development and debugging cycles.

Service Testing Strategies: Our SOA implementation employs multiple testing layers tailored to service-oriented concerns. **Service Interface Testing** validates that each service correctly implements its API contracts and handles edge cases appropriately. **Mock Service Strategy** provides lightweight service doubles for unit testing, allowing developers to test service logic without external dependencies. **API Contract Testing** between services ensures backward compatibility and validates that service interfaces meet their specifications. **Service Integration Testing** verifies that services work correctly together, including authentication flows, data consistency, and error handling across service boundaries. **End-to-End Service Chain Testing** validates complete user workflows that span multiple services, ensuring that the service orchestration delivers expected business outcomes.

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

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 Load Distribution

- **Tactics**: Multiple copies of data, multiple copies of computation
- **Implementation**: Container-based service deployment, horizontal pod autoscaling, database read replicas
- **Quality Target**: Support 10,000+ concurrent users with linear scaling

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

Architectural Patterns

Primary Pattern: Service-Oriented Architecture (SOA)

ELO Learning employs a **Service-Oriented Architecture (SOA)** pattern as its primary architectural approach. This represents an evolution from our initial microservices consideration, adapted to better suit our project's specific constraints and requirements.

Migration from Microservices to SOA: Justification

Initially, we considered a pure microservices architecture for its benefits of independent deployment and technology diversity. However, after careful analysis of our project constraints and team capabilities, we transitioned to SOA for the following reasons:

Complexity Management: Microservices proved too complex for our team size and timeline. The overhead of managing independent deployments, service discovery, distributed monitoring, and inter-service communication protocols would have consumed significant development time better spent on core educational features.

Resource Constraints: True microservices require substantial DevOps infrastructure and monitoring capabilities. Our three-component Demo 2 requirement and limited infrastructure budget made SOA's shared deployment model more practical.

Integration Simplicity: SOA's shared database approach and simplified inter-service communication patterns reduce the "distributed system tax" that microservices impose, allowing us to focus on educational functionality rather than distributed systems engineering.

Team Expertise: Our team's existing experience with monolithic and service-based patterns made SOA a more natural progression than the leap to full microservices architecture.

SOA Implementation Details

Our SOA pattern provides the modularity benefits we need while maintaining manageable complexity:

Service Characteristics:

- Each service is modular and focused on a specific domain (Auth, Matchmaking, Problem Management, etc.)
- Services communicate through well-defined REST APIs and WebSocket connections
- Services share database access but maintain clear domain boundaries
- Services are deployed together but can be scaled independently through container orchestration

Service Inventory:

1. **Auth Service**: Handles user registration, login, JWT generation, and OAuth 2.0 flow
2. **Matchmaking Service**: Implements ELO-based algorithm for problem difficulty matching
3. **Math Problem Service**: Manages problem storage, retrieval, and metadata
4. **Stats/Leaderboard Service**: Computes rankings and delivers leaderboard data
5. **User Profile Service**: Manages personal data, progress tracking, and achievements
6. **Analytics Service**: Logs interaction metrics and performance data (planned)

Benefits Realized

- **Simplified Integration**: Shared database access reduces inter-service communication complexity
- **Manageable Deployment**: Services deploy together while maintaining logical separation
- **Development Velocity**: Team can work on different services without complex coordination
- **Quality Assurance**: Easier to implement comprehensive testing across service boundaries

Tradeoffs Accepted

- **Less Independence**: Services are not fully independent, requiring some coordination for changes
- **Shared Database**: Potential coupling through shared data schemas
- **Deployment Coupling**: Services deploy together, reducing deployment flexibility compared to microservices

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: Consistently applied across all backend services using NestJS's Controller-Service-Repository structure, supporting our maintainability quality requirements.

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

Architectural Constraints

Technical Constraints

- **Container Deployment**: All services must be containerized using Docker for consistent deployment across environments
- **Cloud Platform**: System must deploy to either AWS or Azure 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**: Three components must be fully implemented by June 27, 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

Technology Choices

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

Alternatives Considered:

1. **Vue.js & Nuxt.js**: Strong SSR capabilities and gentle learning curve
2. **SvelteKit**: Excellent performance with minimal bundle size
3. **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:

1. **Express.js**: (Selected) Lightweight, flexible Node.js framework with extensive middleware ecosystem
2. **NestJS**: Structured TypeScript framework with built-in dependency injection and testing
3. **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:

1. **MongoDB + PostgreSQL**: NoSQL flexibility with relational consistency
2. **MySQL + Prometheus**: Standard relational with monitoring-focused time-series
3. **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:

1. **Socket.IO (standalone)**: Feature-rich but requires additional integration overhead
2. **Firebase Realtime Database**: Easy setup but vendor lock-in concerns
3. **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:

1. **Firebase Auth**: Simplified implementation but vendor dependency
2. **Session-based Auth**: Traditional approach but limited scalability
3. **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.

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.

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.

Appendix: Reference Documentation

Demo 1 Architectural Requirements

For historical context and to trace the evolution of our architectural thinking, the original architectural requirements document from Demo 1 can be referenced at: Demo 1 Architectural Requirements Specification

This original document provides insight into our initial architectural considerations and shows how our understanding has matured through the development process. The transition from our Demo 1 specifications to this current version demonstrates the iterative nature of architectural design and how real-world constraints and deeper understanding of quality requirements have shaped our final architectural decisions.

The comparison between these versions illustrates several key architectural learning points: how initial complexity assumptions were refined through practical experience, how quality requirements evolved from general statements to measurable specifications, and how our service-oriented approach emerged as the most suitable pattern for our specific project context and team capabilities.