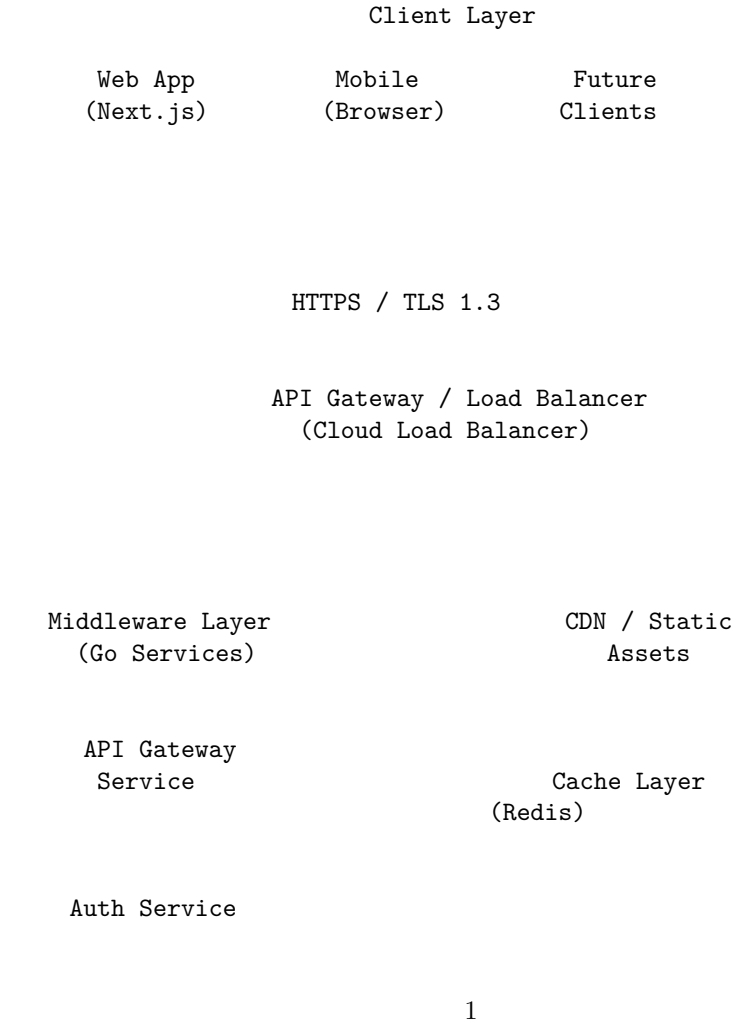
AI-Based Automated STEM Evaluation System - System Architecture Document

1. Architecture Overview

1.1 Architecture Style

The system follows a **three-tier microservices architecture** with clear separation between presentation, business logic, and data layers. The architecture prioritizes: - **Scalability**: Horizontal scaling of independent services - **Maintainability**: Clear service boundaries and responsibilities - **Security**: Defense in depth with multiple security layers - **Performance**: Efficient data flow and caching strategies

1.2 High-Level Architecture Diagram



User Service

Assessment Service

Evaluation Service

Report Service

Forum Service

Notification Svc

Data Layer

PostgreSQL Database

Redis Cache

Cloud Storage (GCS)

External Services

Vertex AI API

Google OAuth

SendGrid Email

2. Component Architecture

2.1 Frontend Layer (Next.js)

2.1.1 Technology Stack

- **Framework**: Next.js 14+ (App Router)
- **Language**: TypeScript 5+
- **UI Library**: React 18+
- **Styling**: Tailwind CSS + shadcn/ui components
- **State Management**: React Query (TanStack Query) + Zustand
- **Form Handling**: React Hook Form + Zod validation
- **Charts**: Recharts or Chart.js

2.1.2 Application Structure

```
/frontend
  /app                    # Next.js App Router
    /(auth)               # Auth-related pages (login, register)
    /(dashboard)          # Protected dashboard routes
      /student            # Student views
      /coach              # Coach views
      /parent             # Parent views
      /admin              # Admin views
    /forum                # Forum pages
    /api                  # API routes (minimal, mostly proxy)
  /components             # Reusable React components
    /ui                   # Base UI components
    /features             # Feature-specific components
    /layouts              # Layout components
  /lib                    # Utilities and helpers
    /api                  # API client functions
    /auth                 # Auth utilities
    /utils                # General utilities
  /hooks                  # Custom React hooks
  /types                  # TypeScript type definitions
  /stores                 # Zustand stores
  /public                 # Static assets
```

2.1.3 Key Frontend Features

- **Server-Side Rendering (SSR)**: For initial page loads and SEO
- **Client-Side Rendering (CSR)**: For interactive components
- **Incremental Static Regeneration (ISR)**: For frequently accessed public content
- **Optimistic Updates**: For better UX in form submissions
- **Real-time Updates**: WebSocket connection for notifications

2.1.4 Frontend Security

- Content Security Policy (CSP) headers
- XSS protection through React's built-in escaping
- CSRF protection using double-submit cookies
- Secure cookie storage for tokens
- Input sanitization before display

2.2 Middleware Layer (Go Services)

2.2.1 Service Architecture Each service follows clean architecture principles:

```
/service
  /cmd                    # Entry points
    /server               # Main server
  /internal               # Private application code
    /domain               # Business entities
    /usecase              # Business logic
    /repository           # Data access
    /delivery             # HTTP handlers
      /http               # REST handlers
      /middleware         # HTTP middleware
    /infrastructure       # External integrations
  /pkg                    # Public libraries
  /config                 # Configuration
  /migrations             # Database migrations
```

2.2.2 Core Services

A. API Gateway Service

- **Responsibility**: Request routing, authentication, rate limiting
- **Port**: 8080
- **Key Functions**:
  - JWT token validation
  - Request/response logging
  - Rate limiting per user/IP
  - CORS handling
  - Request tracing
  - API versioning

B. Authentication Service

- **Responsibility**: User authentication and session management
- **Port**: 8081
- **Key Functions**:

  - User registration and login
  - Password hashing (bcrypt)
  - JWT token generation and validation
  - OAuth 2.0 integration (Google)
  - Session management
  - Password reset flow
  - MFA support

C. User Service

- **Responsibility**: User profile and relationship management
- **Port**: 8082
- **Key Functions**:
  - User CRUD operations
  - Role management (RBAC)
  - Class and team management
  - Parent-student linking
  - User search and filtering
  - Profile photo management

D. Assessment Service

- **Responsibility**: Question management and test administration
- **Port**: 8083
- **Key Functions**:
  - Dynamic question generation
  - Assessment session management
  - Adaptive difficulty adjustment
  - Response recording
  - Question pool management
  - Assessment scoring

E. Evaluation Service

- **Responsibility**: Submission processing and AI evaluation
- **Port**: 8084
- **Key Functions**:
  - File upload handling
  - Image analysis coordination
  - Code evaluation orchestration
  - Engineering notebook processing
  - AI feedback generation
  - Evaluation result storage
  - Queue management for async processing

F. Report Service

- **Responsibility**: Analytics and report generation
- **Port**: 8085
- **Key Functions**:
  - Individual student reports
  - Class analytics
  - Benchmarking calculations
  - Progress tracking
  - PDF report generation
  - Export functionality

G. Forum Service

- **Responsibility**: Discussion board and Q&A management
- **Port**: 8086
- **Key Functions**:
  - Post CRUD operations
  - Comment threading
  - Vote management
  - Content moderation
  - Search functionality
  - Tag management

H. Notification Service

- **Responsibility**: Multi-channel notifications
- **Port**: 8087
- **Key Functions**:
  - Email notifications
  - In-app notifications
  - Notification preferences
  - Digest scheduling
  - WebSocket push notifications

2.2.3 Middleware Technology Stack

- **Framework**: Gin (high performance) or Echo
- **Language**: Go 1.21+
- **Database Driver**: pgx (PostgreSQL)
- **Cache Client**: go-redis
- **HTTP Client**: resty
- **Validation**: validator.v10
- **Testing**: testify
- **API Documentation**: Swag (Swagger)

2.2.4 Inter-Service Communication

- **Synchronous**: REST over HTTP/HTTPS

- **Asynchronous**: Message queue (optional: Cloud Pub/Sub)
- **Service Discovery**: Kubernetes DNS or Cloud Run service URLs
- **Circuit Breaker**: gobreaker library
- **Retry Logic**: Exponential backoff

2.3 Data Layer

2.3.1 PostgreSQL Database **Database Version**: PostgreSQL 15+

Key Design Decisions: - Multi-tenant design with organization isolation - JSONB columns for flexible schema elements - Partitioning for large tables (submissions, activity logs) - Read replicas for reporting queries - Connection pooling (PgBouncer)

Performance Optimizations: - Proper indexing strategy - Materialized views for complex reports - Query optimization with EXPLAIN ANALYZE - Vacuum and analyze scheduling - Table partitioning by date for logs

2.3.2 Redis Cache **Use Cases**: - Session storage - API rate limiting counters - Cached API responses - Real-time leaderboards - Pub/Sub for WebSocket notifications - AI prompt caching

Configuration: - Redis 7+ - Persistence: RDB + AOF - Eviction policy: allkeys-lru - TTL strategy per data type

2.3.3 Cloud Storage (GCS) **Bucket Structure**: - `{env}-user-uploads`: Student submissions (images, documents) - `{env}-processed-files`: Processed files and thumbnails - `{env}-reports`: Generated PDF reports - `{env}-backups`: Database backups

Access Control: - Signed URLs for temporary access - Service account authentication - Lifecycle policies for old files - Versioning enabled

2.4 External Services Integration

2.4.1 Google Vertex AI **Integration Architecture**:

```
Evaluation Svc




AI Abstraction
    Layer
```

```
Gemini   Gemini
 Pro     Vision
```

Key Features: - **Prompt Templates**: Versioned, testable prompt templates - **Prompt Caching**: 24-hour cache for common prompts - **Model Selection**: Route requests to appropriate model - **Response Parsing**: Structured output parsing - **Error Handling**: Fallback strategies for API failures - **Cost Tracking**: Per-request cost monitoring

Models Used: - **Gemini Pro**: Text analysis, code evaluation, feedback generation - **Gemini Pro Vision**: Image analysis, robotics design evaluation - **Future**: Fine-tuned models for specific VEX evaluation tasks

Prompt Engineering Strategy: - System prompts with VEX-specific context - Few-shot examples for consistency - Output format specification (JSON) - Temperature control per use case - Max token limits

2.4.2 Google OAuth **Flow**: Authorization Code Flow with PKCE **Scopes**: openid, email, profile **Implementation**: - oauth2 Go library - Token refresh handling - Account linking with existing users

2.4.3 Email Service (SendGrid) **Use Cases**: - Welcome emails - Password reset - Notification emails - Weekly digest - Report delivery

Templates: Transactional templates with branding

3. Data Flow Diagrams

3.1 Submission Evaluation Flow

```
Student
 Client

     1. Upload files (images, code, notebook)


API Gateway

      2. Validate auth & route

             3. Store files
Evaluation                    GCS
  Service
```

4. Create submission record

PostgreSQL

5. Queue for AI processing

Background
Worker

6. Process each file type

Image Analysis (Vertex AI Vision)
Code Analysis (Vertex AI + Static Analysis)
Notebook Analysis (Vertex AI)

7. Aggregate results

PostgreSQL 8. Store feedback

9. Trigger notification

Notification
Service

10. Send notification

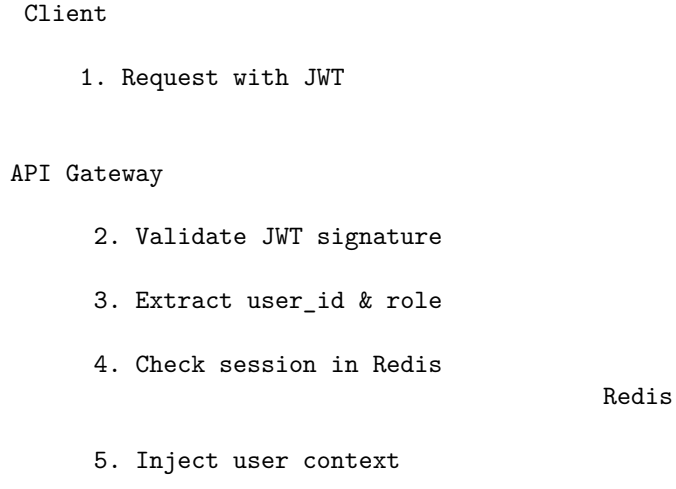
Student
Client

3.2 Authentication Flow

User

1. Enter credentials

3.3 RBAC Authorization Flow



```
  Service

       6. Check RBAC policy

        Student accessing own data?     Allow
        Parent accessing child data?    Check linkage
                                                          PostgreSQL
        Coach accessing class student?   Check class
        Admin?                Allow all

       7. Execute business logic
```

4. Security Architecture

4.1 Security Layers

```
Layer 1: Network Security
    Cloud Load Balancer with DDoS protection
    TLS 1.3 encryption
    Private VPC networking

Layer 2: Application Security
    API Gateway authentication
    Rate limiting (per user/IP)
    Input validation
    CORS policies

Layer 3: Service Security
    JWT validation
    RBAC enforcement
    Service-to-service authentication
    Audit logging

Layer 4: Data Security
    Encryption at rest (AES-256)
    Database access controls
    Parameterized queries (SQL injection prevention)
    Secure file storage with signed URLs
```

4.2 Authentication Mechanism

Token Strategy: Dual-token approach - **Access Token**: Short-lived (15 minutes), contains user info and role - **Refresh Token**: Long-lived (7 days), stored in httpOnly cookie

JWT Payload:

```
{
  "user_id": "uuid",
  "email": "user@example.com",
  "role": "student",
  "organization_id": "uuid",
  "iat": 1234567890,
  "exp": 1234568790
}
```

Token Rotation: Refresh tokens rotated on each use

4.3 RBAC Policy Matrix

Resource	Student	Parent	Coach	Admin
Own submissions	CRUD	R	RU	CRUD
Own assessments	CRUD	R	RU	CRUD
Class submissions	-	-	RU	CRUD
Class reports	-	-	R	R
Parent reports	-	R	-	R
User management	R(self)	R	R	CRUD
System settings	-	-	-	CRUD
Forum posts	CRUD	R	CRUD	CRUD
AI model config	-	-	-	CRUD

Legend: C=Create, R=Read, U=Update, D=Delete

4.4 Data Privacy

PII Handling: - Full names, emails encrypted in database - Student performance data accessible only to authorized users - Parent access requires verified linkage - Data retention policy: 3 years after account closure

Anonymization: - Benchmarking data fully anonymized - Forum posts use display names - Reports to parents exclude other students' identities

5. Scalability and Performance

5.1 Horizontal Scaling Strategy

Frontend (Next.js): - Stateless design enables easy scaling - Deploy multiple instances behind load balancer - CDN for static assets - Auto-scaling based on CPU/memory

Middleware (Go Services): - Each service can scale independently - Kubernetes HPA (Horizontal Pod Autoscaler) - Scale based on request rate or queue depth - Minimum 2 replicas per service for HA

Database: - PostgreSQL read replicas for read-heavy operations - Connection pooling to manage connections - Vertical scaling for write master - Consider sharding for extreme scale (future)

5.2 Caching Strategy

Cache Hierarchy:

1. **Browser Cache**: Static assets (1 year)
2. **CDN Cache**: Public pages, images (1 hour - 1 day)
3. **Application Cache (Redis)**:
   - User sessions: 2 hours
   - API responses: 5-60 minutes (depends on data volatility)
   - Leaderboards: 5 minutes
   - Cached AI prompts: 24 hours
4. **Database Query Cache**: PostgreSQL shared buffers

Cache Invalidation: - Time-based expiration (TTL) - Event-based invalidation (on data updates) - Cache-aside pattern for most read operations

5.3 Database Optimization

Indexing Strategy: - B-tree indexes on foreign keys - Composite indexes for common queries - GIN indexes on JSONB columns - Full-text search indexes for forum content

Query Optimization: - Use EXPLAIN ANALYZE for slow queries - Avoid N+1 queries (use JOINs or batch fetching) - Implement pagination for large result sets - Use materialized views for complex reports

Partitioning: - Partition `submissions` table by month - Partition `activity_logs` table by month - Archive old partitions to cold storage

5.4 Asynchronous Processing

Use Cases: - AI evaluation (can take 2-5 minutes) - Report generation - Email sending - Image processing and thumbnail generation

Implementation: - Background workers poll database queue table - OR use Cloud Pub/Sub for message queue - Retry logic with exponential backoff - Dead letter queue for failed jobs

6. Monitoring and Observability

6.1 Metrics

Application Metrics: - Request rate, latency, error rate (RED metrics) - Service dependencies and health - Database query performance - Cache hit rates - AI API usage and costs

Business Metrics: - User registrations per day - Submissions per day - Active users (DAU/MAU) - Forum engagement - AI evaluation completion rate

Tools: Prometheus + Grafana, or Google Cloud Monitoring

6.2 Logging

Log Levels: DEBUG, INFO, WARN, ERROR, FATAL

Structured Logging:

```
{
  "timestamp": "2025-10-18T10:30:00Z",
  "level": "INFO",
  "service": "evaluation-service",
  "trace_id": "abc123",
  "user_id": "user-uuid",
  "message": "Submission evaluated",
  "metadata": {
    "submission_id": "sub-uuid",
    "duration_ms": 3500
  }
}
```

Log Aggregation: Cloud Logging or ELK stack

6.3 Tracing

Distributed Tracing: OpenTelemetry - Trace requests across services - Identify bottlenecks - Debug production issues

6.4 Alerting

Alert Conditions: - Service down (healthcheck fails) - Error rate $> 1\%$ - P95 latency > 1 second - Database connections exhausted - AI API budget exceeds threshold

Notification Channels: PagerDuty, Slack, Email

7. Deployment Architecture

7.1 Environments

Development: Local Docker Compose **Staging**: Cloud Run (GCP) - mirrors production **Production**: Cloud Run (GCP) with auto-scaling

7.2 Infrastructure as Code

Tool: Terraform **Managed Resources**: - Cloud Run services - Cloud SQL (PostgreSQL) - Redis instance (Memorystore) - Cloud Storage buckets - Load balancers - IAM policies - VPC networks

7.3 CI/CD Pipeline

```
Developer Commit


GitHub Push


GitHub Actions
      Lint code
      Run unit tests
      Build Docker images
      Security scan (Trivy)
      Push to Artifact Registry


Deploy to Staging (auto)


Integration Tests


Manual Approval


Deploy to Production


Smoke Tests
```

7.4 Disaster Recovery

Backup Strategy: - Database: Daily automated backups, 30-day retention - Cloud Storage: Versioning enabled - Configuration: Version controlled in Git

Recovery Objectives: - **RTO** (Recovery Time Objective): 4 hours - **RPO** (Recovery Point Objective): 1 hour

8. Technology Decisions and Trade-offs

8.1 Why Next.js?

Pros: - Excellent developer experience - Built-in SSR/SSG for performance and SEO - Great React ecosystem - Vercel optimization

Cons: - Vendor lock-in concerns (mitigated by hosting on GCP) - Larger bundle sizes than vanilla React

Alternatives Considered: Remix, SvelteKit, Astro

8.2 Why Go for Middleware?

Pros: - Excellent performance and low memory footprint - Simple concurrency model (goroutines) - Strong standard library - Fast compilation - Growing adoption in cloud-native applications

Cons: - Smaller ecosystem than Node.js - Steeper learning curve for some developers

Alternatives Considered: Node.js (TypeScript), Rust

8.3 Why PostgreSQL?

Pros: - Robust ACID compliance - Rich feature set (JSONB, full-text search, arrays) - Excellent performance for relational data - Strong community and tooling

Cons: - Vertical scaling challenges at extreme scale

Alternatives Considered: MySQL, MongoDB (rejected - need strong relationships)

8.4 Why Vertex AI?

Pros: - Managed infrastructure - Latest Google models (Gemini) - Prompt caching support - Integration with GCP ecosystem - Multimodal capabilities (text + vision)

Cons: - Vendor lock-in - Pricing can be expensive at scale

Alternatives Considered: OpenAI API, Azure OpenAI, Open-source models (less capable)

9. Future Considerations

9.1 Potential Enhancements

- **GraphQL API**: For more efficient data fetching
- **Event Sourcing**: For audit trails and time-travel queries
- **Machine Learning**: Custom models fine-tuned on VEX data
- **Real-time Collaboration**: WebRTC for live code editing
- **Mobile Apps**: Native iOS/Android applications

9.2 Scaling Beyond 10K Users

- Implement database sharding by organization
- Separate read/write databases
- Consider serverless functions for spiky workloads
- Multi-region deployment for global users

Document Version: 1.0 **Last Updated**: 2025-10-18 **Next Review**: 2025-11-18