Milestone 4: Architectural Software Design

CSI 680 - Team Information

November 4, 2024

Team Information

Team Number: B

Team Name: SVSMC

Team Members:

- Siddartha Kurmashetti
- Vinay Padala
- Sai Manoj Kartala
- Sushma Kasarla
- Sai Charan Reddy Kanukula

1 Project Summary

Acceptly (From Mistakes to Mastery – One Transition at a Time) is an interactive web-based educational platform designed to help students learn and master Finite Automata (FA) theory through hands-on practice and AI-powered tutoring. The application provides a visual canvas for building finite automata, automated testing capabilities, and intelligent AI assistance that guides students through the learning process without revealing direct solutions.

1.1 Core Requirements and Goals

The system fulfills the following objectives:

1. **Visual FA Builder:** Interactive canvas where students can visually construct finite automata by adding states, transitions, and configuring accepting states.
2. **Automated Testing:** Comprehensive test suite that validates student-constructed automata against predefined test cases, providing immediate feedback on correctness.
3. **AI-Powered Tutoring:** Integration of Google Gemini AI to provide contextual hints, error analysis, and concept explanations using a Socratic teaching method.
4. **Progress Tracking:** Detailed user progress across multiple FA problems and MCQ quizzes, tracking attempts, scores, and completion status.

5. **Quiz System:** Multiple-choice question (MCQ) quizzes to reinforce theoretical understanding alongside hands-on FA construction.

6. **Performance Analytics:** Comprehensive insights and analytics based on user performance, identifying strengths, weaknesses, and areas for improvement.

7. **User Authentication:** Secure user accounts with registration, login, and session management.

1.2 Changes Since Milestone 1

Since the initial project proposal (Milestone 1), the following changes have been implemented:

- **Topic Scope:** The initial proposal included three topics (Graphs, SQL Queries, and Finite Automata). The implementation focused specifically on Finite Automata to ensure deeper feature development and more comprehensive functionality.
- **Technology Stack:**
  - Backend changed from Python to Node.js/Express.js for better integration with the React frontend and unified JavaScript ecosystem
  - Database changed from PostgreSQL to MongoDB for flexible schema design and easier integration with Node.js
  - Frontend maintained React.js as planned
- **Expanded Features:** Added multiple FA problems with varying difficulty levels, comprehensive MCQ quizzes, an Insights analytics dashboard, guided tours, and Google Gemini AI integration (replacing the initially proposed rule-based + lightweight ML approach).

2 Design Pattern Used

The application primarily employs the **Context API Pattern (React Context)**, which is React's built-in solution for global state management. This pattern is implemented through:

- **AuthContext:** Manages authentication state, user information, and authentication-related methods (login, signup, logout) across the entire application.
- **Provider Component:** The `AuthProvider` component wraps the application and provides authentication context to all child components.
- **Consumer Hook:** Components access the context using the `useAuth()` custom hook.

Additional patterns utilized include:

- **Service Layer Pattern:** `apiService` and `geminiService` encapsulate API calls and external service integrations.
- **Protected Route Pattern:** Route guards implemented through `ProtectedRoute` and `PublicRoute` components ensure proper access control.

- **Component Composition Pattern:** React components are composed to build complex UIs from simpler, reusable components.
- **Singleton Pattern:** Service classes (`APIService`, `GeminiService`) are instantiated once and exported as singletons.

3 Software Components

3.1 High-Level Components

1. **Frontend Application (React)**

   Purpose: Provides the user interface and handles all client-side interactions.

   Component Pieces:

   - React Router for navigation
   - Context API for state management
   - Component library for UI elements
   - Canvas API for visual FA construction

2. **Backend API Server (Express.js)**

   Purpose: Handles business logic, authentication, and data persistence operations.

   Component Pieces:

   - Express.js web framework
   - Authentication middleware
   - RESTful API routes
   - Error handling middleware

3. **Database (MongoDB)**

   Purpose: Stores persistent user data, progress tracking, and application state.

   Component Pieces:

   - MongoDB database server
   - Mongoose ODM for schema definition
   - User model schema
   - Connection management

4. **AI Service (Google Gemini)**

   Purpose: Provides intelligent tutoring, hints, and educational feedback.

   Component Pieces:

   - Google Generative AI SDK
   - Gemini 2.0 Flash model
   - Prompt engineering system
   - Response processing

3.2 Class-Level Components

3.2.1 Frontend Classes/Services

1. **APIService (apiService.js)**

   Purpose: Centralized service for all backend API communication.

   Methods:

   - `setToken(token)` - Stores authentication token
   - `getHeaders()` - Constructs HTTP headers with authentication
   - `request(endpoint, options)` - Generic HTTP request method
   - `signup(username, email, password)` - User registration
   - `login(email, password)` - User authentication
   - `getCurrentUser()` - Retrieve current user data
   - `getProgress()` - Fetch user progress
   - `updateFAProgress(problemId, data)` - Update FA problem progress
   - `updateQuizProgress(quizId, data)` - Update quiz progress

2. **GeminiService (geminiService.js)**

   Purpose: Manages interactions with Google Gemini AI for educational assistance.

   Methods:

   - `getHint(problemStatement, currentFA)` - Generate contextual hints
   - `analyzeErrors(problemStatement, currentFA, testResults)` - Analyze failed tests
   - `explainConcept(concept)` - Provide concept explanations
   - `getInsights(progress)` - Generate overall performance insights
   - `getRecommendations(progress, weakConcepts)` - Generate study recommendations
   - `getConceptImprovement(concept, progress)` - Get concept-specific tips

3. **AuthContext (AuthContext.js)**

   Purpose: Manages global authentication state and provides authentication methods.

   Methods:

   - `login(email, password)` - Authenticate user
   - `signup(username, email, password)` - Register new user
   - `logout()` - Clear authentication state
   - `resetPassword(email)` - Initiate password reset
   - `updateProgress(type, data)` - Update local progress state

   State:

   - `user` - Current user object
   - `loading` - Authentication loading state
   - `isAuthenticated` - Boolean authentication status

3.2.2 Backend Classes/Models

1. **User Model (User.js)**

   Purpose: Defines the MongoDB schema for user data and progress tracking.

   Schema Fields:

   - `username` - Unique username
   - `email` - Unique email address
   - `password` - Hashed password
   - `progress` - Nested object containing FA and MCQ progress
   - `createdAt` - Account creation timestamp
   - `lastLogin` - Last login timestamp

   Methods:

   - `comparePassword(candidatePassword)` - Password verification
   - `updateProgress(type, data)` - Update progress statistics

   Pre-save Hook: Password hashing using bcrypt before saving

3.3 Page-Level Components

1. **LandingPage** - Public landing page with project overview and features
2. **UnifiedAuthPage** - Combined login/signup interface with form validation
3. **Dashboard** - Main user dashboard displaying progress overview and quick access
4. **ProblemSelection** - Browse and filter available FA problems and quizzes
5. **FASimulation** - Interactive FA construction environment with canvas, testing, and AI assistance
6. **QuizPage** - MCQ quiz interface with timer and submission handling
7. **Insights** - Performance analytics, AI insights, and recommendations dashboard

3.4 Reusable UI Components

1. **Header** - Application navigation and user menu
2. **AutomataCanvas** - Canvas-based visualization for FA states and transitions
3. **StringTester** - Test case execution and result display
4. **AIHelper** - Floating AI assistant interface for hints and analysis
5. **QuizResults** - Post-quiz results display with analytics
6. **AlertDialog** - Reusable modal dialog component
7. **Pagination** - Pagination controls for problem lists
8. **GuidedTour** - Interactive tutorial system for onboarding

3.5 Backend Route Handlers

1. **Auth Routes (/api/auth)**

   Purpose: Handle authentication and user management.

   Endpoints:

   - `POST /signup` - User registration
   - `POST /login` - User authentication
   - `GET /me` - Get current user (protected)
   - `POST /check-email` - Email availability check
   - `POST /reset-password` - Password reset initiation

   Middleware: `authMiddleware` for JWT token verification, `generateToken` for JWT token generation

2. **Progress Routes (/api/progress)**

   Purpose: Manage user progress tracking.

   Endpoints:

   - `GET /` - Get user progress (protected)
   - `POST /fa/:problemId` - Update FA problem progress (protected)
   - `POST /quiz/:quizId` - Update quiz progress (protected)

3. **Problems Routes (/api/problems)**

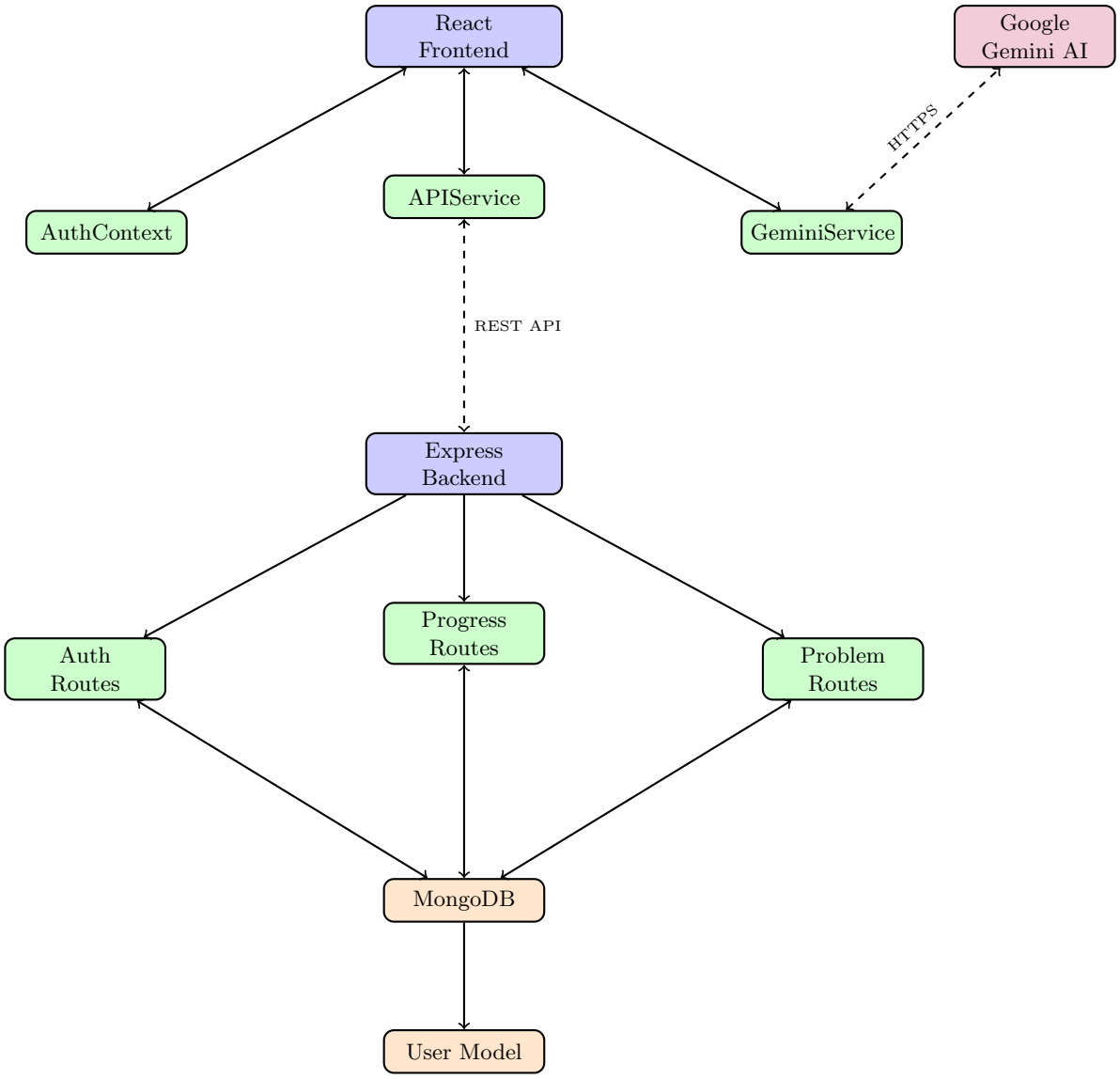
   Purpose: Serve problem and quiz data.

   Endpoints:

   - `GET /` - Get all problems/quizzes with filters (protected)
   - `GET /:id` - Get specific problem/quiz (protected)

4 Wiring Diagram

The following diagram illustrates how components connect and communicate:



4.1 Shared Software Resources

- **JWT Tokens:** Shared authentication tokens passed between Frontend and Backend via HTTP Authorization headers. Tokens are generated by the backend and stored in browser localStorage.
- **User Session Data:** User authentication state and user object shared between Frontend components via AuthContext.
- **MongoDB Connection:** Shared database connection managed by Mongoose ODM, used by all backend route handlers.

4.2 Data Passed Between Components

- **Frontend ↔ AuthContext:** User object, authentication token, loading state, and authentication status. Accessed via `useAuth()` hook.

- **Frontend ↔ APIService:** HTTP requests (login credentials, progress updates) and JSON responses (user data, progress statistics).
- **Frontend ↔ GeminiService:** Problem context, current FA state, test results. Returns AI-generated hints, error analysis, and insights.
- **APIService ↔ Express Backend:** JSON request/response payloads over REST API. Requests include JWT tokens in headers. Responses include user data, progress updates, and problem definitions.
- **GeminiService ↔ Google Gemini AI:** Prompt strings containing problem context and FA state. Returns AI-generated educational responses.
- **Backend Routes ↔ MongoDB:** User documents (username, email, hashed password, progress objects). Queries and updates performed via Mongoose ODM.
- **MongoDB ↔ User Model:** BSON documents with schema validation. Model provides abstraction layer for database operations.

5 Persistence

5.1 Data Persisted

The application persists the following data:

1. **User Account Data**
   - Username (String, unique, required)
   - Email (String, unique, required, validated)
   - Password (String, hashed with bcrypt, required)
   - Created timestamp (Date, auto-generated)
   - Last login timestamp (Date, updated on login)
2. **User Progress Data**
   - FA problem progress: problem ID, status (solved/attempted/unsolved), attempts count, best score, last attempt timestamp
   - MCQ quiz progress: quiz ID, status (completed/attempted/not_started), attempts count, best score, last attempt timestamp, selected answers array
   - Overall progress statistics: solved counts, totals, percentages for both FA and MCQ categories
3. **Session Data**
   - JWT authentication token (stored in browser localStorage)
   - Serialized user object (stored in browser localStorage)

5.2 Storage Format

- **MongoDB:** Data stored as BSON (Binary JSON) documents in MongoDB collections. Database name: `acceptly`, Collection: `users`. Schema defined using Mongoose ODM.
- **localStorage:** Key-value pairs stored as strings in browser's local storage. Objects serialized using JSON.stringify() before storage.

5.3 Storage Location

- **MongoDB:** Cloud-hosted MongoDB Atlas or local MongoDB instance. Connection string stored in `MONGODB_URI` environment variable.
- **localStorage:** Browser's local storage (client-side), persistent across browser sessions and page refreshes.

6 Languages Used

6.1 Primary Languages

1. **JavaScript (ECMAScript 6+)**

   Usage: Primary language for both frontend and backend development.

   Components Built:

   - Entire React frontend application
   - All React components (functional components with hooks)
   - Express.js backend server
   - All API route handlers
   - Service layer classes (APIService, GeminiService)
   - Data models (Mongoose schemas)

   Reasoning: React ecosystem is JavaScript-based, providing seamless integration. Node.js enables JavaScript on the server, allowing code reuse and shared tooling. Modern ES6+ features (async/await, destructuring, arrow functions) improve code readability. Single language reduces context switching and simplifies development workflow.

2. **JSX (JavaScript XML)**

   Usage: React's templating syntax for component definitions.

   Components Built:

   - All React component render methods
   - Conditional rendering logic
   - Event handler bindings

   Reasoning: Declarative syntax makes UI structure clear and maintainable. Seamless integration with JavaScript expressions. React's virtual DOM optimization for performance.

3. **CSS**

   Usage: Styling for all UI components.

   Components Built:

   - Component-specific CSS files (e.g., FASimulation.css, QuizPage.css)
   - Global styles (index.css)
   - Responsive design rules
   - Animation and transition effects

   Reasoning: Standard web styling language. Component-scoped CSS files maintain separation of concerns. CSS Grid and Flexbox for responsive layouts.

4. **JSON**

   Usage: Data serialization and API communication.

   Components Built:

   - API request/response payloads
   - Problem and quiz data definitions (problemsData.js)
   - Configuration files (package.json)

   Reasoning: Standard format for REST API communication. Human-readable and easy to parse. Native JavaScript support (JSON.parse, JSON.stringify).

6.2 Configuration and Markup Languages

- **HTML:** Base markup structure (minimal, handled by React) - `public/index.html`
- **Markdown:** Documentation and README files

7 Deployment

7.1 Deployment Platform

The application is deployed as a **web application** accessible through web browsers (desktop and mobile).

7.2 Deployment Architecture

The application follows a full-stack web deployment architecture with separate frontend and backend services:

- **Frontend:** React application deployed as static files on a web hosting platform
- **Backend:** Express.js server deployed on a cloud Platform-as-a-Service (PaaS) or cloud server
- **Database:** MongoDB Atlas (cloud-hosted) or self-hosted MongoDB instance
- **External Services:** Google Gemini AI accessed via HTTPS API (cloud-hosted)

7.3 Deployment Options

7.3.1 Frontend Deployment

Target Platform: Web (hosted on cloud platform)

Deployment Options:

- **Netlify** or **Vercel** - Optimized for React applications with automatic builds and CDN
- **AWS S3 + CloudFront** - Scalable static hosting with global CDN
- **Heroku** - Full-stack hosting capability

Deployment Process:

1. Build production bundle: `npm run build`
2. Static files generated in `build/` directory
3. Deploy build directory to hosting platform
4. Configure environment variables (API keys, API URLs)

7.3.2 Backend Deployment

Target Platform: Cloud server or Platform-as-a-Service (PaaS)

Deployment Options:

- **Heroku** - Easy deployment with automatic scaling
- **AWS EC2** or **Elastic Beanstalk** - Full control over server environment
- **DigitalOcean App Platform** - Simplified container deployment
- **Railway** or **Render** - Modern PaaS with MongoDB integration

Requirements:

- Node.js runtime environment (v14+)
- Environment variables: `MONGODB_URI`, `JWT_SECRET`, `PORT`, `NODE_ENV`

7.3.3 Database Deployment

Target Platform: MongoDB Atlas (cloud) or self-hosted MongoDB

Recommended: MongoDB Atlas - Fully managed cloud database with automatic backups, scaling, and global cluster distribution. Free tier available for development.

7.4 Accessibility

The application is accessible through:

- Desktop web browsers (Chrome, Firefox, Safari, Edge)
- Mobile web browsers (responsive design)
- Tablet devices (responsive layout)

No native mobile app or desktop application - purely web-based for maximum accessibility and easy updates.