FACULTY OF TECHNOLOGY
Information & Communication Technology
Subject: Capstone Project

Aryan Mahida – 92200133011 **Jay Mangukiya - 92200133040**

Implementation and Technical Documentation

The code quality, functionality and integration across different components has been already taken care in the implementation part. But here is the brief of them.

1. Code quality:

- We have maintained clean coding practices which includes proper variable naming convention, comments and modular architecture, where components are divided into reusable code blocks.
- Every function is embedded with comments, which mentions what these functions do.
- This below snippet displays that the code follows modular architecture. Variable naming conventions, error handling and comments.

```
* Creates a new quiz with the provided data
 * @param {CreateQuizData} quizData - The quiz data to create
 * @returns {Promise<Quiz>} The created quiz with associations
 * @throws {Error} When category is not found or creation fails
 */
async createQuiz(quizData: CreateQuizData): Promise<Quiz> {
  try {
    // Verify category exists
    const category = await Category.findByPk(quizData.categoryId);

    if (!category) {
      throw new Error(`Category with ID ${quizData.categoryId} not found`);
    }

    const quiz = await Quiz.create({
      title: quizData.title,
      description: quizData.description,
      difficulty: quizData.difficulty || 'MEDIUM',
      timeLimit: quizData.timeLimit,
      maxQuestions: quizData.maxQuestions,
      categoryId: quizData.categoryId,
      createdById: quizData.createdById
    });
        logInfo('Quiz created', { quizId: quiz.id, title: quiz.title });
    return quiz;
  } catch (error) {
    logError('Failed to create quiz', error as Error, { title: quizData.title });
    throw error;
  }
}
```

2. Integration across components:

We have implemented a microservice based architecture where there is clear separation between presentation (frontend), logic, backend and database operations.

A brief about component integration.

Frontend-backend integration:
The frontend uses Axios HTTP client for automatic token refresh and error handling. All API calls are typed using Typescript interfaces shared between frontend and backend.

```
// Shared interface definitions
export interface CreateQuizRequest {
  title: string;
  description?: string;
  difficulty?: 'EASY' | 'MEDIUM' | 'HARD';
  timeLimit?: number;
  categoryId: number;
}

// Frontend service implementation
export class QuizService {
  async createQuiz(data: CreateQuizRequest): Promise<Quiz> {
    const response = await this.apiClient.post<Quiz>('/api/quizzes', data);
    return response.data;
  }
}
```

Websocket integration:
Real time quiz functionality use socket.io. Events are strongly typed and include error handling for connection failures.

```
// WebSocket event definitions
interface ServerToClientEvents {
  match_started: (data: MatchStartedPayload) => void;
  question_received: (data: QuestionPayload) => void;
  answer_result: (data: AnswerResultPayload) => void;
  match_completed: (data: MatchCompletedPayload) => void;
}
// Client-side event handling with error recovery
socket.on('disconnect', () => {
  console.warn('WebSocket disconnected, attempting reconnection...');
  // Implement exponential backoff reconnection
});
```

This is how the integrations between different modules have been made while maintaining the coding standards as discussed above.

Technical documentation:

1. Code structure and organization:

1.1 File system of the project:

The file system contains two major separated folders of frontend and backend which handles the application logic. Further the se folders are divided into number of nested subfolders which makes the structure organized and easier to locate.

```
QuizUP/
├── Backend/                                  # NodeJS Server
│   ├── src/
│   │   ├── controllers/                      # HTTP request handlers
│   │   ├── services/                         #  core logic layer
│   │   ├── models/                           #  sequelize models
│   │   ├── routes/                           # API routes
│   │   ├── middleware/                       #  auth, validation, logging
│   │   ├── utils/                            # helpers
│   │   ├── config/                           # database config
│   │   ├── server.ts                         # API server
│   │   └── matchServer-enhanced.ts           # Websocket server
│   ├── tests/
│   ├── package.json                          # dependencies
│   └── Dockerfile                            # container configuration
├── Frontend-admin/                           # React frontend
│   ├── src/
│   │   ├── components/                       # React components
│   │   │   ├── ui/                           # base UI components (shadcn/ui)
│   │   │   ├── admin/                        # admin components
│   │   │   ├── student/                      # student components
│   │   │   └── shared/                       # shared components
│   │   ├── services/                         # API clients and Websocket
│   │   ├── hooks/                            # react hooks
│   │   ├── contexts/                         # React contexts
│   │   ├── types/                            # Typescript definitions
│   │   ├── utils/                            # helpers
│   │   └── pages/
│   ├── package.json                          # frontend dependencies
│   └── Dockerfile                            # frontend container
├── docker-compose.yml                        # multi service orchestration
└── README.md                                 # documentation
```

2. Implementation details:

2.1 Key algorithms and protocols:

The match synchronization code:

```
export class EnhancedMatchService {
  private activeMatches: Map<string, MatchInstance> = new Map();
  private store: StoreInterface; // Redis or In-Memory fallback

  async createFriendMatch(payload: CreateFriendMatchPayload): Promise<FriendMatch>
{
    // Generate collision-resistant join code
    let joinCode: string;
    let attempts = 0;

    do {
      joinCode = this.generateJoinCode(); // 6-character alphanumeric
      attempts++;
    } while (await this.isJoinCodeTaken(joinCode) && attempts < 10);
    // Store match state in Redis with TTL
    const matchData = {
      id: uuidv4(),
      quizId: payload.quizId,
      creatorId: payload.userId,
      joinCode,
      status: 'WAITING',
      players: [payload.userId],
      maxPlayers: payload.maxPlayers || 2,
      createdAt: new Date().toISOString()
    };
    await this.store.set(`match:${matchData.id}`, JSON.stringify(matchData), 3600);
    await this.store.set(`joinCode:${joinCode}`, matchData.id, 3600);
        return matchData;
  }
}
```

When a player creates a match, the system generates a unique 6 character join code and verifies it isn't already taken. The match data (quiz ID, creator, players, status, timestamp) is then stored in redis with a TTL, so inactive matches expire automatically.

Redis also maps the join code to the match ID, which also allows quick lookups when another player joins. Since redis is shared across servers, all players see the same updated state in real time. This makes sure that the matches are unique, just like in memory only systems, which is responsible for 1v1 quiz battle.

2.2 Some of the useful technologies used:

Docker:
We used Docker to package the whole project so it can run the same way everywhere. For development, the container runs with live reload, which makes testing changes faster. For production, only the final build is kept, so the image is smaller and quicker to start. This makes the deployment easier and also the application can also be run on any other machine without any additional dependencies or hardware requirements.

Socket.io:
The real time part of the quiz works through socket.io. It handles things like when a player joins, a new question is sent, or when the match ends, and updates everyone right away. Redis helps share the state between servers, so even if lots of people are playing at the same time, it stays in sync. The system also checks tokens when players connect to make sure only valid users join, and it can recover from errors without kicking players out.

Sequelize
We used PostgreSQL as the main database and Sequelize as the ORM to make handling queries easier in code. To make things run faster, we added indexes on columns that are used a lot, like player ratings, match status, and question categories. This helps reduce query time when the database grows big. We also set up a materialized view to track quiz performance (like average score, total attempts, and completion time). Instead of running heavy queries again and again, this view stores the results and updates every hour, which makes stats and analytics load much quicker.

Redis
Redis is used as a cache layer to make the system faster and reduce load on the database. We store things like user sessions, quiz details, leaderboards, and match states for a limited time (from 1 minute for leaderboards to 24 hours for sessions). This way, if the same data is requested again, Redis can serve it instantly without hitting the database. For example, when a leaderboard is requested, Redis checks if it's already stored. If not, it generates it once, saves it, and then reuses it until it expires. This keeps responses fast and smooth, even with lots of users playing at the same time.

3. Testing and results:

For this QuizUP project, we selected multiple testing frameworks based on the technology stack and testing requirements.

Backend (NodeJS + Typescript):

- **Jest**: Primary testing framework for unit and integration tests
- **Supertest**: HTTP assertion library for API endpoint testing
- **Sequelize**: Database testing with mock implementations

Frontend (React + Typescript):

- **Vitest**: Modern testing framework optimized for Vite projects
- **React testing library**: Component testing with user centric approach
- **User event**: Simulating real user interactions

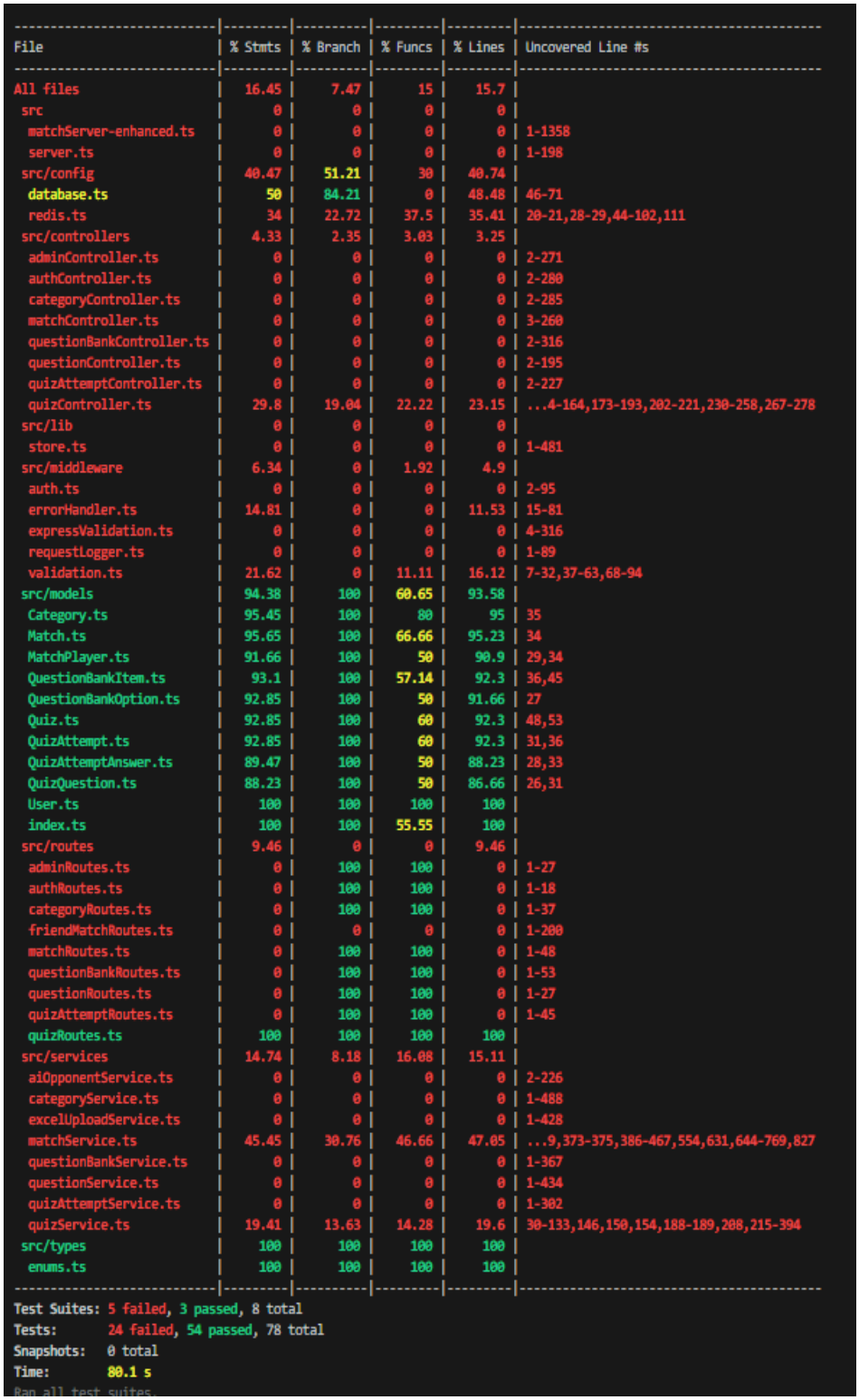
Performance Testing:

- **Built in nodeJS performance API**: Response time measurements
- **Docker health checks**: Container performance monitoring
- **Prometheus metrics**: Real time performance tracking

The testing approach follows a three tier strategy

1. **Unit tests**: Testing individual components and functions in isolation
2. **Integration tests**: Testing component interactions and API endpoints
3. **Performance tests**: Measuring system performance under various loads

```
Test Suites: 5 failed, 3 passed, 8 total
Tests:       23 failed, 55 passed, 78 total
Snapshots:   0 total
Time:        56.641 s
Ran all test suites.
```



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	16.45	7.47	15	15.7	
src	0	0	0	0	
matchServer-enhanced.ts	0	0	0	0	1-1358
server.ts	0	0	0	0	1-198
src/config	40.47	51.21	30	40.74	
database.ts	50	84.21	0	48.48	46-71
redis.ts	34	22.72	37.5	35.41	20-21,28-29,44-102,111
src/controllers	4.33	2.35	3.03	3.25	
adminController.ts	0	0	0	0	2-271
authController.ts	0	0	0	0	2-280
categoryController.ts	0	0	0	0	2-285
matchController.ts	0	0	0	0	3-260
questionBankController.ts	0	0	0	0	2-316
questionController.ts	0	0	0	0	2-195
quizAttemptController.ts	0	0	0	0	2-227
quizController.ts	29.8	19.04	22.22	23.15	...4-164,173-193,202-221,230-258,267-278
src/lib	0	0	0	0	
store.ts	0	0	0	0	1-481
src/middleware	6.34	0	1.92	4.9	
auth.ts	0	0	0	0	2-95
errorHandler.ts	14.81	0	0	11.53	15-81
expressValidation.ts	0	0	0	0	4-316
requestLogger.ts	0	0	0	0	1-89
validation.ts	21.62	0	11.11	16.12	7-32,37-63,68-94
src/models	94.38	100	60.65	93.58	
Category.ts	95.45	100	80	95	35
Match.ts	95.65	100	66.66	95.23	34
MatchPlayer.ts	91.66	100	50	90.9	29,34
QuestionBankItem.ts	93.1	100	57.14	92.3	36,45
QuestionBankOption.ts	92.85	100	50	91.66	27
Quiz.ts	92.85	100	60	92.3	48,53
QuizAttempt.ts	92.85	100	60	92.3	31,36
QuizAttemptAnswer.ts	89.47	100	50	88.23	28,33
QuizQuestion.ts	88.23	100	50	86.66	26,31
User.ts	100	100	100	100	
index.ts	100	100	55.55	100	
src/routes	9.46	0	0	9.46	
adminRoutes.ts	0	100	100	0	1-27
authRoutes.ts	0	100	100	0	1-18
categoryRoutes.ts	0	100	100	0	1-37
friendMatchRoutes.ts	0	0	0	0	1-200
matchRoutes.ts	0	100	100	0	1-48
questionBankRoutes.ts	0	100	100	0	1-53
questionRoutes.ts	0	100	100	0	1-27
quizAttemptRoutes.ts	0	100	100	0	1-45
quizRoutes.ts	100	100	100	100	
src/services	14.74	8.18	16.08	15.11	
aiOpponentService.ts	0	0	0	0	2-226
categoryService.ts	0	0	0	0	1-488
excelUploadService.ts	0	0	0	0	1-428
matchService.ts	45.45	30.76	46.66	47.05	...9,373-375,386-467,554,631,644-769,827
questionBankService.ts	0	0	0	0	1-367
questionService.ts	0	0	0	0	1-434
quizAttemptService.ts	0	0	0	0	1-302
quizService.ts	19.41	13.63	14.28	19.6	30-133,146,150,154,188-189,208,215-394
src/types	100	100	100	100	
enums.ts	100	100	100	100	

Test Suites: 5 failed, 3 passed, 8 total
Tests: 24 failed, 54 passed, 78 total
Snapshots: 0 total
Time: 80.1 s
Ran all test suites.

Above images shows how many tests have passed and details related to each file / code performance against the tests which have been performed.

4. Setting up the project:

Method 1: Docker setup

1. Clone repository

git clone https://github.com/Jaymangukiya22/Capstone-Project.git

cd Capstone-Project

2. Copy environment files

cp .env.example .env

3. Start all services

docker-compose up --build

4. Initialize database (in another terminal)

docker-compose exec backend npm run db:setup

docker-compose exec backend npm run seed:quick

Method 2: Local development

1. Clone and setup backend

git clone httpshttps://github.com/Jaymangukiya22/Capstone-Project.git

cd Capstone-Project /Backend

2. Install dependencies

npm install

3. Setup database

createdb quizup_db

npm run build

npm run db:setup

npm run seed:quick

4. Start backend services

npm run dev

npm run dev:match

5. Setup frontend (new terminal)

cd ../Frontend-admin

npm install

npm run dev # Frontend (port 5173)