LogiQ App Project Report

Jianyang Gu, 3759581

1 Introduction

1.1 App Overview

LogiQ is a mobile quiz app that helps users practice formal logic in an interactive way. It is designed for logic learners and provides real-time feedback, detailed performance statistics, and support for propositional logic. The app includes three types of problems: Truth Tables, Equivalence, and Inference, featuring different question formats and difficulty levels.

1.2 Licenses of Components

The LogiQ project integrates several open-source components across the frontend, backend, and deployment layers. Each component follows its own license, as summarized in Table 1.

Table 1: Licenses of main components used in the LogiQ project.

Component	Type	License
Flutter SDK	Cross-platform UI Framework	BSD 3-Clause License
Dart SDK	Programming Language	BSD 3-Clause License
React	Frontend Framework	MIT License
Go (Golang)	Backend Language	BSD-style License
Gin Framework	REST API Framework	MIT License
MongoDB	Database	SSPL (Server Side Public License)
Redis	In-memory Data Store	BSD 3-Clause License
Docker / Docker Compose	Deployment Tools	Apache License 2.0

2 Overview

2.1 Implemented Features

2.1.1 Front-end (Flutter App)

- **Home** Serves as the central hub of the application. Key functionalities include choosing topics and difficulty levels to begin practice sessions and displaying daily accuracy statistics.
- **Quiz** Implements the interactive quiz experience with progress tracking and automatic evaluation. Each quiz presents 10 questions drawn from the question bank, which covers 3 categories (Truth Table, Equivalence, Inference), 3 types (Single Choice, Multiple Choice, True/False) and 3 difficulty levels (Easy, Medium, Hard). Besides, offline quiz is also provided. When network is not available, quiz would be generated by questions from local data. After quiz is completed, users can submit and check results.
- **Review** The Review module allows users to revisit their completed quizzes and analyze their historical results. Users can look through all their quiz history and select one quiz to inspect detailed results, completion time and accuracy.

- **Profile** This page mainly contains two parts: user management and analytical statistics. Users are allowed to logout, to update personal information, including nickname, password and avatar and to see personal performance statistics, such as number of tasks completed, error distribution and daily accuracy.

2.1.2 Back-end (Go+Gin) & Database

The back-end uses Gin web framework. It implements RESTful APIs that handle authentication, CURD functions of quizzes and users, and administrative task generation and management. So, APIs are used by both front-end app and admin panel. It interacts with MongoDB for persistent data storage, while Redis is used for cache handling.

2.1.3 Admin Site

The React-based administrative interface provides tools for managing and analyzing the question bank. Administrators can view, filter, delete, or export questions, as well as generate new ones automatically by specifying quantity and parameters. An integrated dashboard visualizes the accuracy and distribution of the questions, helping maintain a continuously evolving and balanced question repository.

2.2 Additional Features

2.2.1 App Features

- **Timer and Progress in Quiz** The real-time progress of the current quiz is displayed and a timer is shown during each quiz task to record how long the user takes to complete the question. The completion time is stored and used in statistical analysis.
- **Light and Dark Mode** The app supports light and dark themes. All pages can automatically adapt. Manual switch is provided.

2.2.2 Question Diversity by 3 dimensions

The metadata of a question is described by 3 dimensions, categories (Truth Table, Equivalence, Inference), types (Single Choice, Multiple Choice, True/False) and difficulty levels (Easy, Medium, Hard). This design in combination of a random choice of variables and operators in the task generation process can improve the diversity and quality of tasks significantly.

2.2.3 Task Automatic Generation

Except in offline mode, questions are sourced from a pre-configured local question bank. All questions in the database are automatically generated by algorithms. The question generation logic is written in Go and integrated as a separate module within the back-end service. For detailed generation logic, refer to Section 4. Brief Programming-Level Documentation.

2.2.4 User Management

The application includes a complete user management feature that enables secure authentication and personalized usage. Users can register new accounts, verify their email addresses, log in with valid credentials, and log out safely when finished. Once authenticated, the app provides individualized quiz history and performance tracking. All user-related operations are processed through a unified API layer to ensure consistency and session security. More technical details will be further explained in Section 4. Brief Programming-Level Documentation.

2.2.5 Admin Interface

A modern and powerful administrative panel is implemented using React, providing a unified login entry for administrators. Its specific functionalities are described in Section 2.1.3.

2.3 Partially Functional or Pending Improvements

While the current implementation of LogiQ delivers all core learning and administrative functions, several features remain partially implemented or planned for future improvement.

2.3.1 App

- **Login System** The login system can be further enhanced by integrating Single Sign-On (SSO) or other modern authentication mechanisms to improve both security and user convenience. Currently, verification emails are sent via Gmail's SMTP service, which is sufficient for testing and prototype deployment. For a production environment, however, this should be replaced with a scalable and reliable cloud-based email service such as Amazon SES or another enterprise-grade provider to ensure delivery stability, scalability, and compliance with security policies.
- **Draft** A draft feature is also planned, allowing users to resume unfinished quizzes seamlessly.
- **Leaderboard** The task score system, which currently aggregates user scores based on the number of correctly solved questions multiplied by their difficulty coefficient, was designed to support a future leaderboard function but is not yet fully implemented.

2.3.2 Back-end & Database

- **Query Optimization** As this was my first implementation using MongoDB, there is significant potential to refine schema design and indexing strategies to optimize query performance.
- **Asynchronous Mechanism** The current implementation does not effectively leverage Go's goroutines to execute asynchronous tasks. Certain operations, such as updating user and question statistics after a user submits a quiz, could be implemented asynchronously to further optimize response times.

2.3.3 Admin Site

- **Setting** The Settings module in the admin interface remains incomplete. Currently, the parameters for automatic task generation are defined in a static YAML configuration file. A future update will allow administrators to modify these parameters dynamically through the settings panel.
- **Logging Service** A centralized logging and monitoring system could be introduced to record administrative actions and generation activities, improving transparency and maintainability.

3 Setting Up the Project

3.1 App

```
cd frontend
flutter pub get
# According to your device
flutter run -d ios
flutter run -d android
```

3.2 Backend & Environment Variables

```
cd backend
docker build -t backend:latest .
# copy .env file
cp .env docker/.env
```

3.3 Database

```
cd docker
docker compose up -d
docker compose ps
```

4 Brief Programming-Level Documentation

4.1 Project Architecture

The system follows a layered architecture with clear separation of concerns. The Flutter app (left) consists of UI, state, repository, API, and model layers, following an MVVM-like pattern. The Go backend (right), built with Gin, provides RESTful APIs with JWT-based authentication, CORS, and error handling. It includes core services such as Auth, Quiz, Stats, User Management, and Task Generation. At the data layer, MongoDB handles persistent storage, while Redis provides caching. Refer to Figure 1.

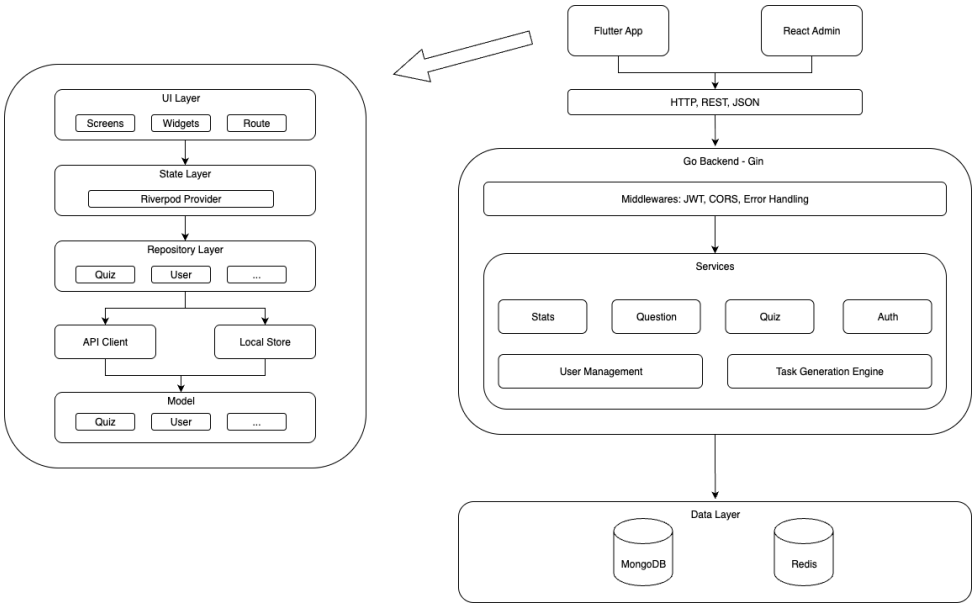

Figure 1: System Architecture of LogiQ

4.2 ER Diagram of the Database

Figure 2 shows the entity–relationship model of the LogiQ system. It illustrates the main entities, their key attributes, and relationships among them.

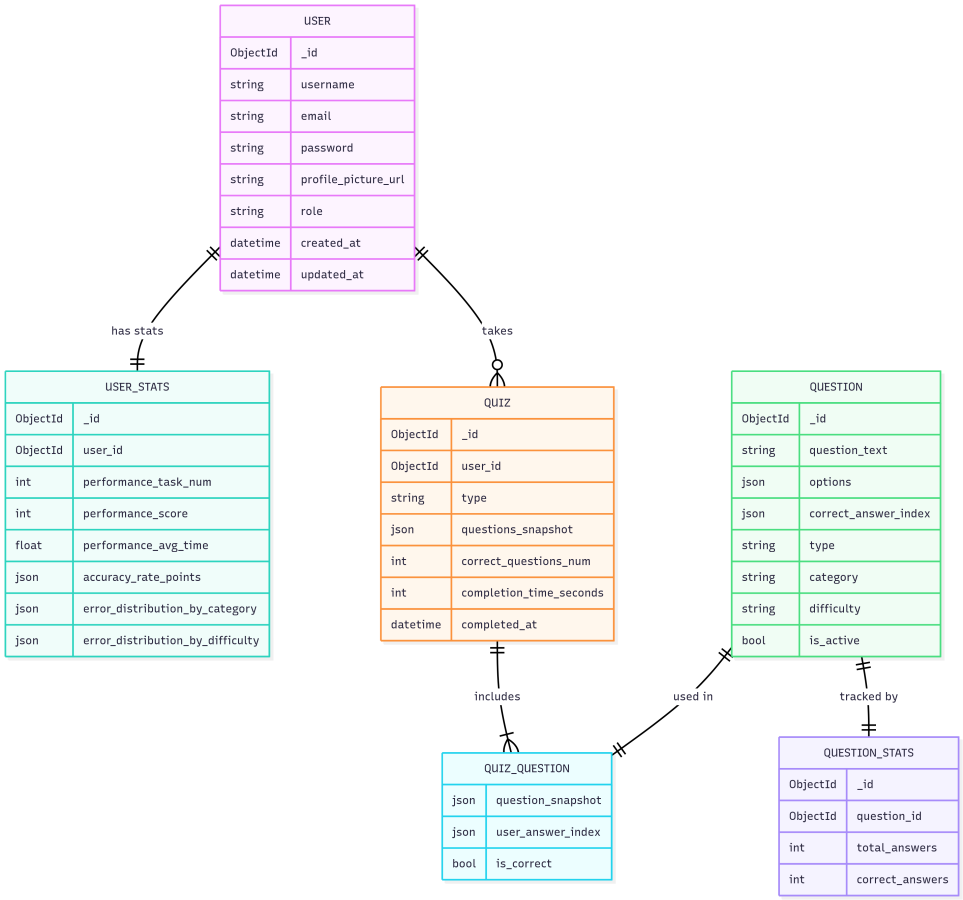

Figure 2: ER Diagram of LogiQ

4.3 Key Feature Implementations

4.3.1 Automatic Task Generation

The task generation module is designed as a configuration-driven and multi-layered pipeline that ensures both flexibility and logical correctness in automatically generating quiz questions. Its workflow consists of several core stages as follows:

- **Configuration-driven Bootstrap** The generation service loads YAML configurations (difficulty distributions, intent weights, and inference/equivalence parameters) and passes them to the sampler, generators, and builders, ensuring that the question style remains configurable and easily tunable.
- **Sampler – Sampling** Upon receiving a `GenerateQuestion` request, the sampler selects difficulty, category, question type, and intent (falling back to weighted randoms when unspecified). It also determines variable count, formula depth, and allowed operators; for multiple-choice questions, it decides the required number of correct answers. Together these parameters form the `Plan` and `Profile`.
- **Generator – Drafting Content** The service selects the appropriate generator for each category: truth-table generators synthesize propositions and enumerate truth/falsehood assignments; equivalence generators derive equivalent and non-equivalent expressions using rewrite rules; inference generators select premise/conclusion templates and perform equivalence transformations to increase variety. Each generator validates its candidates and verifies that they meet the Plan's constraints for correct and distractor counts, retrying within a capped attempt limit.
- **Builder – Presentation** Once candidate pools are ready, the preparation stage converts formulas or inference data into template strings and determines the correct answer for True/False types. The prompt builder selects an intent template and replaces placeholders, while the choice builder maps intent configuration to candidate pools, samples correct and incorrect options, shuffles them, and records correct indices.
- **Assembler – Final Question** The assembler combines the Plan (difficulty, category, and type), prompt text, and options into a standardized `Question` object, aggregating all generated questions into a list for return.
- **Service – Persistence and APIs** The `QuestionService` calls the generation service, then inserts the resulting questions into MongoDB's `questions` collection. Administrators can query or soft-delete questions via the `/question` route, while end users draw quiz items from the same collection.

Overall, the task generation pipeline ensures that every generated question adheres to configuration constraints and logical correctness, achieving both automation and consistency across all categories.

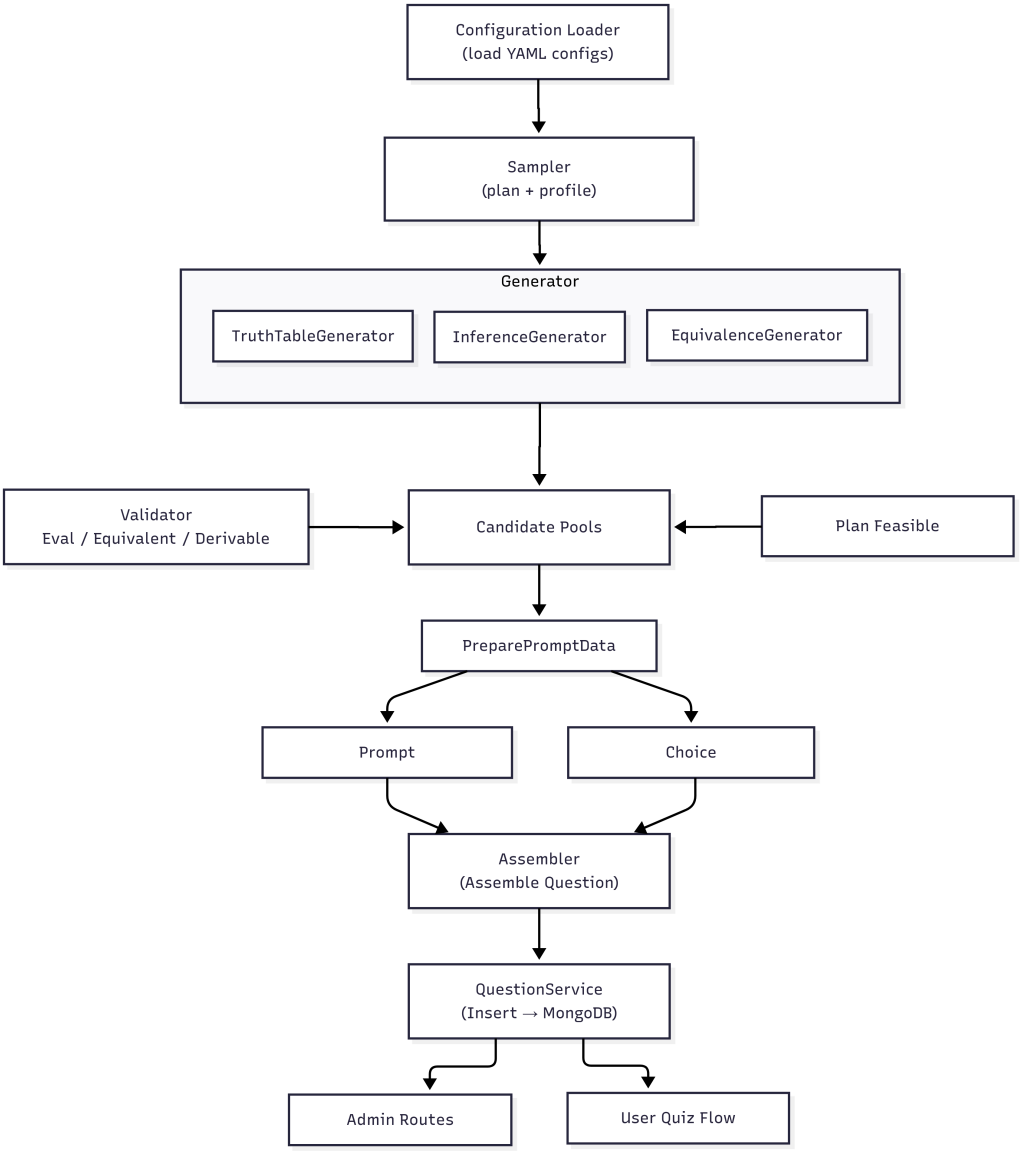

Figure 3: Task Generation Flow

4.3.2 User Management

The user management module covers functions such as registration, login and user profile maintenance. The system adopts a layered design that clearly separates frontend and backend logic.

- **Frontend Presentation Layer (UI Layer)** Users complete actions such as login, registration, verification code input, profile update, and logout on the interface. The UI layer is responsible for form validation and result feedback, keeping the interaction process concise and consistent while decoupling it from core business logic.
- **State and Communication Layer (State / API Layer)** After successful login, the state management layer caches the JWT and user information, while the API layer uniformly appends the `Authorization: Bearer ...` header to all requests. If a request returns a 401 status, the system immediately clears the cache and redirects to the login page to prevent unauthorized access.

This layer also synchronizes frontend state after profile updates to keep local data consistent with the server.

- **API Proxy Layer** The frontend encapsulates actions such as login, registration as REST API calls. The verification process maintains a countdown and temporary token on the frontend, and upon successful validation, triggers navigation or password reset. This layer abstracts the underlying communication logic, making business calls clearer.
- **Backend Routing and Middleware Layer** The backend centrally manages endpoints such as `/auth` and `/user`. All protected resources are validated through JWT middleware. The middleware parses the token and passes user identity information downstream, allowing services to remain stateless and achieving backend statelessness.
- **Backend Business Service Layer** The user service handles password verification (bcrypt comparison), verification code validation, and creation of formal user accounts, while also managing user profile and statistical data. Users can update information on the frontend; after verifying permissions, the backend updates the corresponding documents and synchronizes statistical fields. The verification service generates short-term OTPs, stores them in Redis, and sends them to the user's email via the mail service.
- **Data and Infrastructure Layer** This layer consists of MongoDB, Redis, and the mail service. MongoDB stores user account data and statistics; Redis temporarily stores verification codes; the mail service is responsible for sending verification emails; JWT secrets are configured via environment variables, and tokens are set with expiration times to ensure security.

Overall, the user management module implements a unified framework for authentication, profile management, and statistics synchronization through layered design, achieving both security and maintainability. On this basis, future extensions such as SSO, two-factor authentication, and personalized data presentation can be further developed.

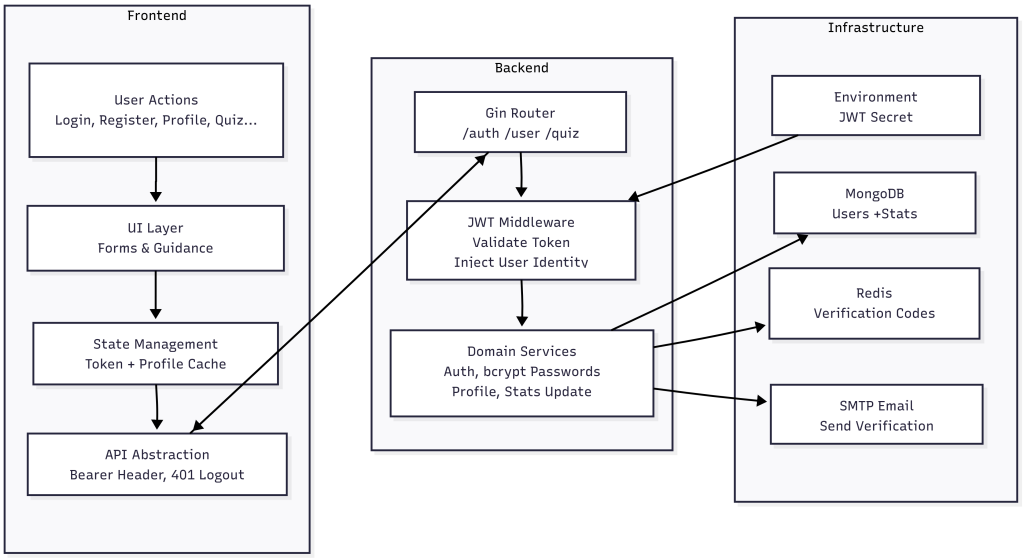

Figure 4: User Management Flow

5 Tests

5.1 Kinds of Automatic Tests

- **Unit Tests (Go)**: Unit tests were written for the `task generation` module to verify the behavior of the sampling and generation logic, since the development difficulty of this part is larger.
- **API Tests (Postman)**: For the main application logic, API testing was conducted manually using Postman. Most endpoints were verified for expected status codes and responses.
- **Frontend**: No automated frontend testing was implemented. Functional correctness was verified through manual debugging on the iOS simulator; no verification was performed on the Android simulator.

5.2 Test Coverage

Backend: The coverage report from `go test -cover ./...` showed as Table 2. This is expected.

Table 2: Backend Test Coverage by Package

Package / Module	**Coverage (%)**
`backend/generation/assembler`	100.0
`backend/generation/builder/choice`	57.5
`backend/generation/builder/prompt`	89.5
`backend/generation/generator/eq`	66.4
`backend/generation/generator/inf`	81.7
`backend/generation/generator/tt`	70.5
`backend/generation/sampler`	75.8
`backend/generation/service`	70.2
Other backend modules	
(handlers, middleware, models, routes, services, utils)	0.0

Frontend: No automated coverage available; manual verification only.

6 Packages and Libraries

The project relies on third-party libraries across the frontend, backend, and admin site. Each package provides key functionality for state management, data communication, authentication, and visualization, collectively improving scalability and development efficiency.

Table 3: Packages and Their Purposes

Platform	Packages	Purpose
App	flutter_riverpod, riverpod_annotation	Reactive state management and code generation.
	go_router	Declarative routing and guarded navigation.
	http	REST API requests to backend services.
	shared_preferences	Local storage of JWT and user profile data.
	pin_code_fields	OTP input component for email verification.
	fluttertoast	Toast notifications for feedback messages.
	gap, google_fonts, font_awesome_flutter	UI enhancement with spacing, and icons.
	fl_chart, percent_indicator	Visualization of quiz performance.
	image_picker, video_player	Avatar selection and tutorial video playback.
Backend	gin-gonic/gin	Go HTTP web framework.
	go.mongodb.org/mongo-driver	MongoDB driver.
	go-redis/redis/v8	Redis client caching.
	golang-jwt/jwt/v5	JWT-based stateless authentication.
	golang.org/x/crypto/bcrypt	Secure password hashing and verification.
	Azure/azure-sdk-for-go/sdk/storage/azblob	Presigned URL generation for avatar uploads.
	gopkg.in/yaml.v3	Parse YAML configs for task generation settings.
	joho/godotenv	Load local .env for development configuration.
Admin	react, react-router-dom	Frontend framework and route management.
	antd	UI component library .
	react-query	Query caching and async state management.
	axios	HTTP requests.
	echarts	Visualization of quiz statistics.
	vite	build tool for development and bundling.

7 Sources

The following external sources were consulted during the development of the LogiQ project. They include troubleshooting resources, technical blogs, and official documentation that supported specific implementation steps. Often, it's after reviewing others' solutions that I suddenly gain the inspiration to solve my own problems.

During the development process, ChatGPT was used as an assistant tool. Typical usage scenarios included exploring possible technical options, clarifying official documentation, and comparing architec-

tural design trade-offs.

1. https://stackoverflow.com/questions/77332218/unknown-riverpod-generator-erro
2. https://blog.51cto.com/yingnanxuezi/13767632
3. https://app.flchart.dev/
4. https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-upload-go
5. https://blog.csdn.net/weixin_43578304/article/details/127935919
6. https://kinsta.com/blog/gmail-smtp-server/
7. https://juejin.cn/post/7053715866126385165
8. https://www.runoob.com/mongodb/mongodb-aggregate.html
9. https://redis.io/docs/latest/develop/clients/go/
10. https://www.reddit.com/r/AZURE/comments/1g7g3vs/signature_did_not_match_error_while_generating_a/?tl=zh-hans
11. https://blog.csdn.net/wohu1104/article/details/126533806

8 Conclusion

During the development of LogiQ, I learned and applied cross-platform development with Flutter, backend service design using Go, and containerized deployment based on Docker. This process not only deepened my understanding of modern web architecture but also provided insights into collaboration methods and integration approaches across different technology stacks.

I also encountered numerous challenges during the project, such as coordinating frontend-backend API integration, designing a rational database structure, and implementing complex question generation logic. Through continuous debugging and optimization, I gradually developed a clear modular design mindset. Although the project's business logic was relatively straightforward, independently completing a fully functional full-stack system significantly enhanced my capabilities in engineering implementation, troubleshooting, and system architecture.

Furthermore, the project still holds considerable optimization potential. Beyond the functional enhancements mentioned in Section 2.3, a more comprehensive and automated testing framework remains to be developed, which would further enhance system stability and usability.

During development, I also used AI tools like ChatGPT as an auxiliary resource. For less complex features, it could indeed generate usable code after minor modifications. However, its greatest value to me lies in guiding me through challenging decisions like architectural design, layer separation, and component abstraction. AI acts as a mentor, helping me analyze the advantages and disadvantages of different approaches and offering sound recommendations.

Finally, I extend my sincere gratitude to PD Dr. Manfred Kufleitner for his guidance and dedication throughout this semester.