Realtime Quiz System Design

Introduction

This document is high-level design document of a real-time quiz feature for an English learning application. This feature will allow users to answer questions in real-time, compete with others, and see their scores updated live on a leaderboard.

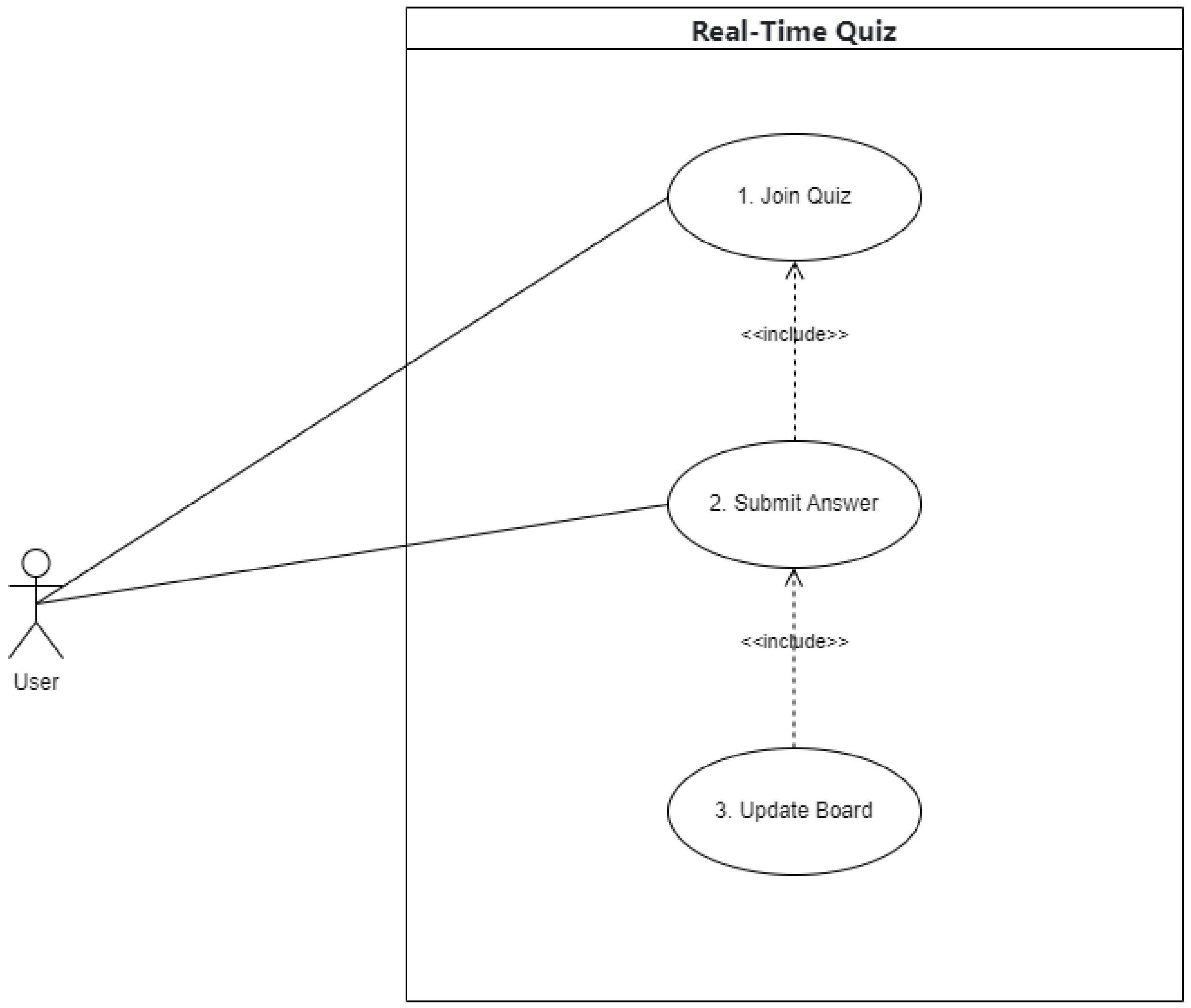
This document includes 2 parts

1. Design for current application
2. Design for future application

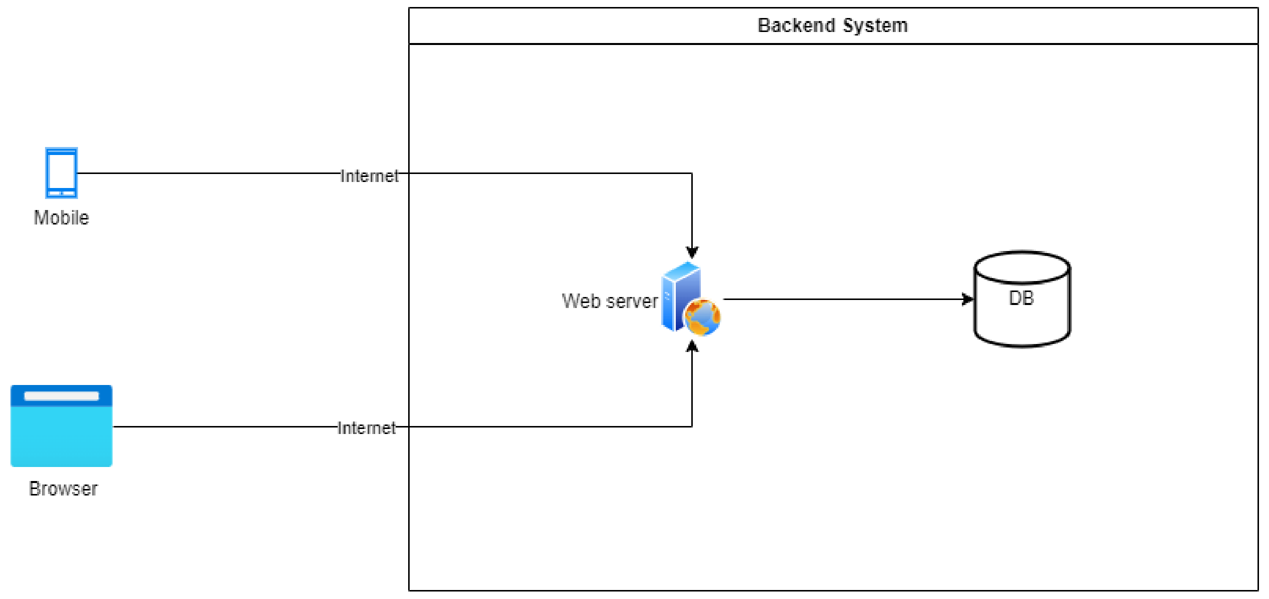
Note: Feature "3. Update board" is chosen to be implemented in coding challenge demo-app.

Current Application Design

Use Case Diagram



Architecture Diagram



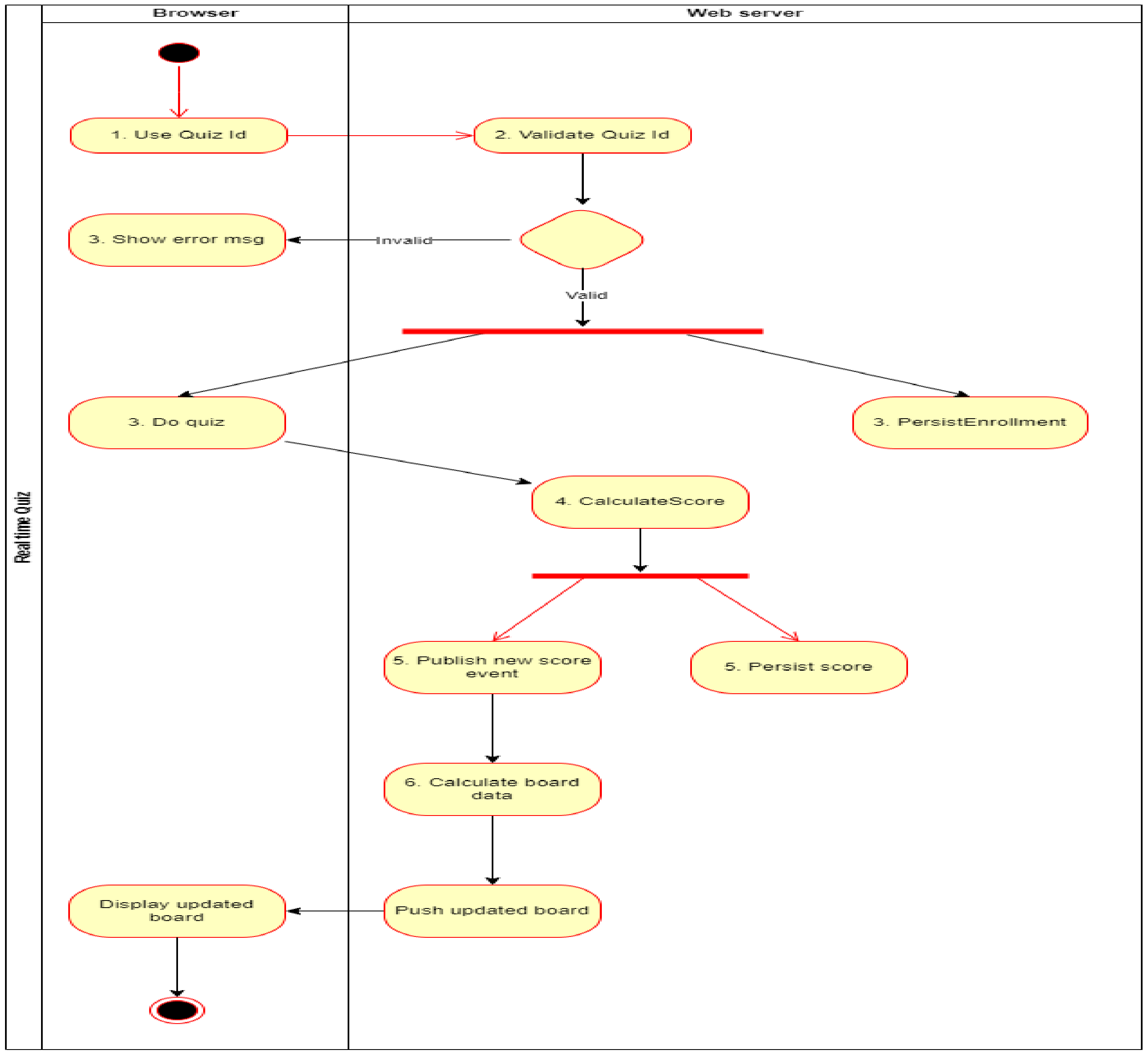
Client-side - more explanation at “Future application design/Architecture styles”

- SPA architecture style
- React app
- **Mobile**: Mobile user connects to BE system via limited throughput internet connection.
- **Browser**: Web user connects to BE system via Internet connection

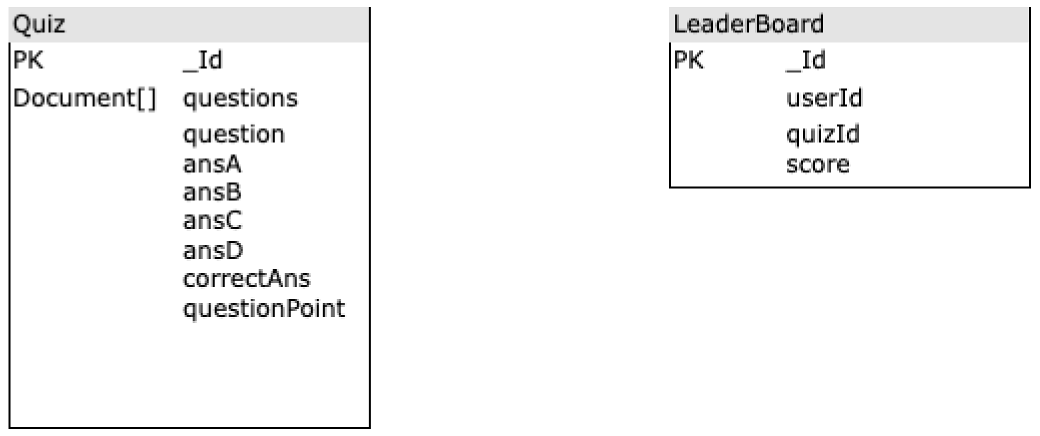
Server-side - more explanation at “Future application design/Architecture styles”

- Modular architecture style
- **Webserver**: Receive client requests, process then return response to client
- **DB**: RDBMS database to persist durable system data

Data Flow


Realtime Quiz
Browser
Web server
1. Use Quiz Id
2. Validate Quiz Id
3. Show error msg
Invalid
Valid
3. Do quiz
3. PersistEnrollment
4. CalculateScore
5. Publish new score event
5. Persist score
6. Calculate board data
Display updated board
Push updated board

ERD Diagram



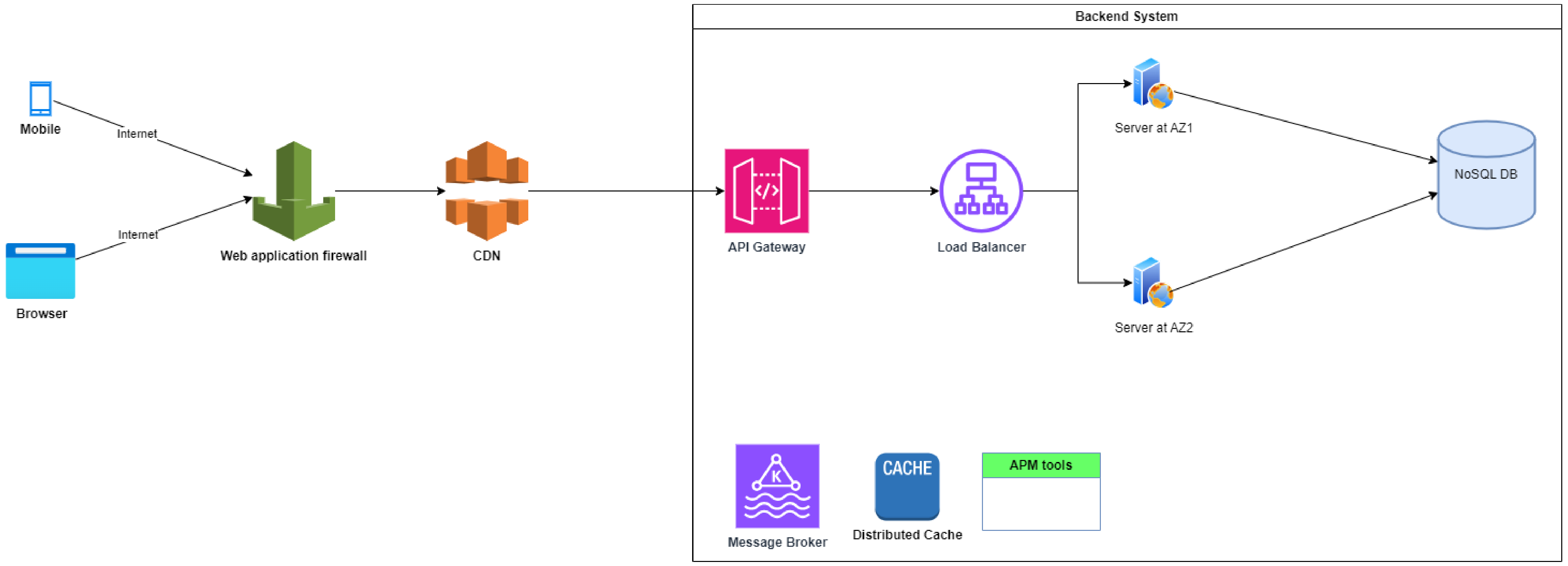
Technology

Category	Content	Explanation
Client-side language	React	React’s component-based architecture, virtual DOM, declarative syntax, and strong community support make it an excellent choice for building scalable and maintainable client-side applications
Server-side language	Java	Java offers several benefits due to its robust features, extensive ecosystem, and broad applicability
Web service	Restful	Simple, suitable for demo app
Open-Source Framework	Spring	Powerful development framework
Event publish	Spring event	Simple, suitable for demo app
Database	MongoDB	MongoDB's flexibility, scalability, performance, and ease of use make it an attractive choice for modern application development

Future Application Design

“Use case diagram”, “data flow diagram”, ERD diagram are the same as “current application design”

Architecture Diagram



Client-side

- React app
- **Mobile**: Mobile user connects to BE system via limited internet connection.
- **Browser**: Web user connects to BE system via Internet connection

Backend-side

a. **WAF**: Protect against common web exploits such as XSS, SQL Injection.
b. **Content delivery network (CDN):** Cache static content to respone time
c. **API Gateway**: Implement cross-cutting tasks
    - Request mapping: Ensure internal API changes do not impact external API being used by client
    - Authn, Authz: Do early authn, Authz to improve performance
    - Rate limit, Throttling: Request management to manage API usages and avoid attacks such as DDOS
d. **Load Balancer**: Distribute network traffic to improve scalability
e. **Servers**: Located in different availability zones to ensure fault-tolerance, availability and reliability
f. **Database**: Persist durable data, NoSQL database can be chosen for easy vertical/horizontal scale to serve more users
g. **Message broker**: Serve event-driven features such as ScoreUpdate, BoardUpdate interaction. Event-driven programming helps distribute requests cross servers to enhance processing capability needed in peak hours.
h. **Distributed cache**: Cache heavy responses to reduce processing time
i. **APM tools:** Monitoring and Observability

Architecture styles

Client-side

Client-side Single Page Application (SPA) architecture is chosen

- Performance Benefits
- Better Control Over User Interface
- Scalability and Modularity

Service-side

At the beginning time, monolith first approach is chosen

- Reduce time to market
- Easy deployment, monitoring, debugging
- Save infrastructure cost
- Easily support transactional business and strong consistency

Specifically, modular architecture is used

- Separation of concerns for modules e.g. Quiz Management, Score Management, Board Management
- Module development independent, easy maintenance and fast development

In each module, hexagonal architecture is used

- Separation of concerns among layers
- Flexibility and adaptability for new requirements or library, middleware replacement

Client and server interaction follows headless architecture meaning that client talks to server via web service such REST

- Client development is independent from server
- Client and server can change technology without impact the other thanks to REST nature

In the future, microservice architecture should be considered when

- System needs independent developments
- Requirement becomes complicated
- Organization is bigger with multiple teams
- System needs independent deployments among modules

Technology

Category	**Content**	**Explanation**
Client-side language	React	React's component-based architecture, virtual DOM, declarative syntax, and strong community support make it an excellent choice for building scalable and maintainable client-side applications
Server-side language	Java	Java offers several benefits due to its robust features, extensive ecosystem, and broad applicability
Web service	GraphQL	Suitable for complicated query requirement and performance optimization especially for mobile client with limited network throughput
Open-Source Framework	Spring	Powerful development framework for java
Event publisher	Kafka	Suitable for app with a big number of users

Database	MongoDB	MongoDB's flexibility, scalability, performance, and ease of use make it an attractive choice for modern application development

Non-functional requirement

1. **Scalability**
    a. Server can serve many concurrent users by vertical scaling and horizontal scaling if necessary by using load balancer and auto-scaling
    b. Message broker helps distribute requests to different servers and do not exhaust server resources.
    c. Distributed cache helps data visible in different servers
    d. No-SQL database can be easily vertical/horizonal scaling to avoid bottle neck in DB
2. **Performance**
    a. CDN helps store static contents
    b. Distributed cache stores heavy responses
    c. Multiple servers help logic processing more smoothly
3. **Reliability**
    a. Load-balancer and multiple servers in different AZ ensure system fault tolerance and resilience
    b. API tools to monitor log, failure, deliver alert, dig deep into system insights which help system to deal with errors promptly
    c. WAF to avoid common attacks, rate limit, throttle, time-out to avoid API overuse. All of this make system immune from malicious usages
    d. Coding follows best practices and patterns such as: Circuit breaker, bulkhead pattern, proper test suites, exception handling
4. **Maintainability**
    a. Code is organized into separate modules for easier management, less learning curve., For example, each team can focus on one module and if there are bugs then they can focus on one module instead of big code base
    b. Each module utilizes hexagonal architecture. By this system can easily adapt new requirements without code changes, reduce error rate thanks to separate of concerns across layers
    c. APM tools provides log tracing, monitoring, system insights so that developers can easily trace, debug, fix bugs
5. **Monitoring and Observability:**
    a. Logging: ELK stack - a powerful stack for log and event data collection, processing, and visualization.
        i. Elasticsearch: A search and analytics engine.
        ii. Logstash: A server-side data processing pipeline that ingests data from multiple sources simultaneously.
        iii. Kibana: A data visualization tool for Elasticsearch.
    b. Observability: Grafana to visualize metrics, alert and monitoring allowing users to create dashboards and visualizations for metrics collected from various data sources, including Prometheus, InfluxDB, and others.