Project Report for Design and Implementation of an Interactive Online Gomoku Platform

Practice Module for Certificate in Designing Modern Software Systems

Team 6

Members:

Shashank Bagda

Hao Tian

Cheng Muqin

Li YuanXing

Ding Zuhao

CONTENTS

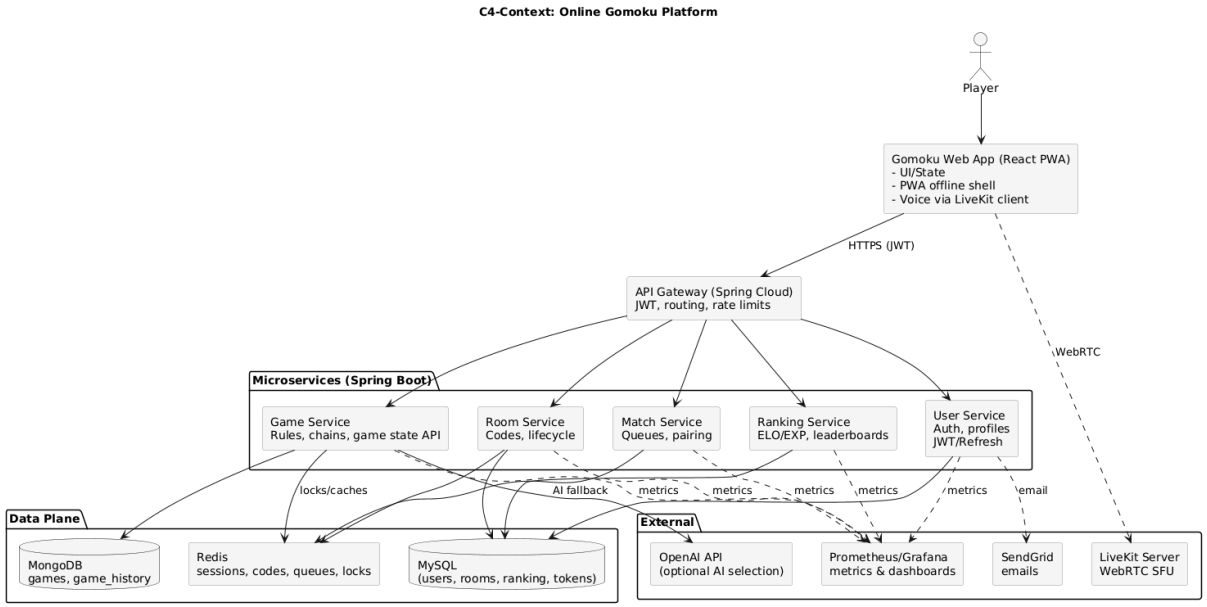
1. Introduction

1.1 Background

Gomoku is a deceptively simple five-in-a-row game that becomes a distributed-systems problem the moment you add real-time multiplayer, matchmaking, rankings, and observability. Our goal was to deliver a **production-grade web platform** that feels instant to a player yet remains maintainable and auditable to an engineer.

We implemented a **polyglot, microservices architecture** (Spring Boot services with Redis/MySQL/MongoDB) fronted by a **React PWA**. Real-time UX is achieved by a blend of **polling/SSE** for deterministic state sync and **LiveKit** for voice. The AI practice mode uses a **hybrid strategy**: local heuristics for fast candidate generation and an **OpenAI adapter** for optional final selection, with strict fallbacks.

The system is deployed on **Docker Swarm** with **GitLab CI/CD** promoting from a test cluster to production, **Prometheus/Grafana** for metrics, and automated **quality gates** (lint, SAST/SCA/DAST, unit & perf tests).



GC-DMSS outcomes:

Architect systems with appropriate decomposition → domain-oriented services with separate data stores and cache responsibilities.

Design for quality attributes → performance (latency budgeted <200 ms/move), availability (redundant replicas), security (JWT, RSA, SAST/SCA/DAST), observability (Prometheus/Grafana).

1.2 Business Needs

To better understand the market landscape of existing Gomoku platforms, our team conducted a comparative analysis of currently available solutions, including traditional offline play, web-based games, and mobile implementations. Part of the research results are summarized in the table below.

Gomoku Way	Private mode	Casual mode	Ranked mode	Training vs AI	Leader board Display	Level and scores setting
Traditional offline	☑	☐	☐	☐	☐	☐
Papergames.io (Web app)	☑	☐	☑	☑	☑	☐ Only scores display
Gomoku (Native app)	☑	☑	☐	☑	☐	☐
Gomoku (Progressive web app)	☑	☑	☐	☑	☐	☐

Table 1. Functional Comparison of Existing Gomoku Platforms

In addition to the comparative analysis of existing Gomoku platforms, the team also conducted a user preference survey to better understand player expectations and gameplay needs. Part of the research results are shown in the table below.

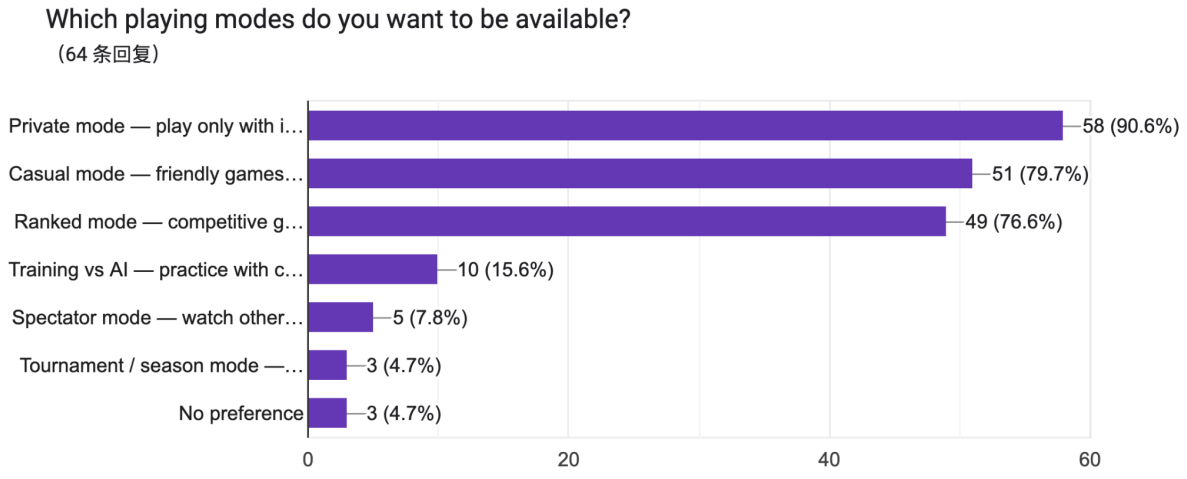


Figure 1. Partial result from the user survey on preferred playing modes

The comparison and survey results indicate that most existing Gomoku platforms only provide limited features, often missing ranked modes, level systems, and real-time interaction.

According to our survey, players show the highest interest in Private Mode (90.6%), Casual Mode (79.7%), and Ranked Mode (76.6%), showing a strong demand for more diverse and competitive gameplay.

These findings suggest a clear need for a comprehensive online Gomoku platform that integrates multiple play modes, ranking and leveling mechanisms, and interactive features to enhance both competitiveness and user engagement.

1.3 Stakeholders

Name	Role	Primary Responsibilities
Shashank Bagda	Frontend Lead & AI Gameroom Developer	React PWA, Tailwind UI, AI Module (Factory + Strategy Pattern), Quality & Security Testing, Game Room Communications (LiveKIT)
Hao Tian	Backend Lead & DevOps Engineer	SpringBoot Microservices, User Module, CI/CD, Security & Performance Testing, Deployment, Game Room, Game Logic
Cheng Muqin	Backend Developer - Leaderboard	Ranking Service Implementation, Database Integration
Li YuanXing	Backend Developer - Matchmaking	Redis Room Management, Real-Time Pairing, Game Room Matching, Prometheus & Grafana
Ding Zuhao	QA Engineer	JMeter Load Testing

1.4 Project Scope

The system delivers a feature-complete web platform for interactive Gomoku play. It spans UI, real-time communication, game logic, and backend persistence, integrated via REST and WebSocket interfaces.

1.4.1 Functionality in Scope

User Management:

- The system shall allow users to register, log in, log out, verify email addresses, and reset passwords.
- Passwords shall be encrypted with RSA during transmission and stored using BCrypt hashing with a random salt.
- Each user shall have a profile including nickname, email, avatar, country, gender, and account status.
- JWT shall be used for authentication and authorization, with a default 24-hour token lifetime.

Gomoku - Game Logic:

- The system shall implement standard Gomoku rules (five-in-a-row to win, full-board draw detection).
- Game states shall include waiting, in progress, and finished.
- Players shall be able to perform actions ready, move, resign, undo, request draw, and restart.
- Draw and restart shall only take effect upon mutual agreement.
- Game board states shall be persisted in MongoDB.

Gomoku - Room Logic:

- Players shall be able to create private rooms with a unique room code.
- Other players can join the room using the generated code.
- If a player leaves during an active game, the system shall automatically treat it as a loss.

Gomoku - Match Logic:

- The system shall provide a matchmaking feature for both casual and ranked modes.
- Players can join or cancel the matchmaking queue and query their queue status.
- Matchmaking shall be managed via Redis to ensure concurrency safety and real-time response.

Ranking Logic:

- The system shall award experience and score based on match result and mode.
- Players' experience shall determine level progression.

- Ranked matches shall affect player rating points, while casual and private matches shall not.
- The leaderboard shall display top 50 players daily, weekly, monthly.
- A personal ranking query shall be available on each player's profile.

Real-Time Communication:

- The system shall support real-time updates of game status via REST and SSE (Server-Sent Events).
- The frontend shall use 1–2 s polling or SSE streaming to synchronize game state.
- Communication shall be secured using token-based URL authentication.

Security:

- All endpoints shall be protected by JWT-based authentication and role authorization.
- Passwords shall be transmitted using RSA encryption and stored using salted BCrypt hashes.
- Email verification codes shall be randomly generated and cached in Redis for five minutes.
- Gateway-level CORS and security filters shall prevent unauthorized access.

Data Storage:

- MySQL shall store structured data such as user profiles, scores, and leaderboard information.
- MongoDB shall store game boards and historical match records.
- Redis shall cache sessions, verification codes, matchmaking data, and distributed locks.

API Specification:

- The system shall expose RESTful APIs using JSON format.
- APIs shall be categorized as public (registration, login, leaderboard) and protected (room, match, ranking).

1.4.2 Functionality out of Scope

- Mobile app (native) versions.
- AI training using machine learning models.
- Payment gateways or subscription features.
- Social network integration (e.g., OAuth via Google/Facebook).
- Cloud auto-scaling Kubernetes deployment (beyond Docker compose setup).

1.4.3 Quality Attributes

Attribute	Target Goal	Implementation Mechanism
Scalability	Support 50+ concurrent matches	Microservice backend, Redis cache
Availability	≥ 99% uptime in demo deployment	Container health checks & CI/CD rollback
Security	Protect user data & prevent injection	Spring Security JWT, input validation
Maintainability	Modular codebase per service	Clear separation of concerns & DTOs
Performance	≤ 200 ms latency per move	WebSocket SSE and Redis optimisation
Observability	Real-time system metrics available	Prometheus + Grafana integration
Testability	≥ 80% code coverage in CI builds	JUnit + JMeter automated testing

2. Project Conduct

2.1 Project Plan

The team adopted an Agile Scrum Lifecycle, executed across five sprints over a ten-week academic window. The planning centred around incremental feature delivery and DevSecOps integration.

Sprint Overview:

Sprint	Duration	Key Objectives	Core Deliverables
Sprint 0	05 Sep - 19 Sep	Requirement gathering, architecture ideation	Vision document, initial C4 diagram, tech-stack decision
Sprint 1	20 Sep - 03 Oct	Backend service scaffolding, REST API contracts, FE layout skeleton	Spring Boot services, React routing & PWA shell
Sprint 2	04 Oct - 17 Oct	WebSocket integration, Redis room cache, LiveKit voice/chat	Functional multiplayer room
Sprint 3	18 Oct - 31 Oct	AI strategy engine (Factory + Strategy), Leaderboard persistence	AI vs Player mode, ranking service
Sprint 4	01 Nov - 05 Nov	JMeter load tests, Grafana dashboards, final refactors	TPS benchmarks, Prometheus metrics

2.2 Project Status

Milestone	Status	Sprint Period	Owner
Architecture baseline approved	☑	Sprint 0	All members
REST APIs operational	☑	Sprint 1	Hao Tian
WebSocket multiplayer stable	☑	Sprint 2	Shashank Bagda + Hao Tian
Room management complete	☑	Sprint 2	Li YuanXing
AI practice module functional	☑	Sprint 3	Shashank Bagda
Gameplay logic complete	☑	Sprint 2	Hao Tian
Leaderboard integration complete	☑	Sprint 2	Cheng Muqin
Game Matching integration complete	☑	Sprint 2	Li YuanXing
CI/CD pipeline and Grafana dashboards	☑	Sprint 3	Hao Tian
Add grafana, prometheous successfully	☑	Sprint 4	Li YuanXing
Final presentation delivered	☑	Sprint 4	All Members

2.3 Project Metrics

Effort Distribution:

Member	Primary Roles	Total Effort (hrs)	Key Contributions
Shashank Bagda	Frontend Lead, AI Engine, QA	100 h	Game Room UI, AI Factory/Strategy, testing
Hao Tian	Backend Lead, DevOps & CI/CD	110 h	Spring services, pipelines, deployment, performance test
Cheng Muqin	Backend Leaderboard	70 h	ranking logic, database design
Li YuanXing	Backend Matchmaking	75 h	Redis room, pairing logic, grafana, prometheous
Ding Zuhao	Performance Engineer	10h	JMeter plans

Total team effort ≈ **375 hours**.

Velocity and Burndown Snapshot:

Sprint	Story Points Committed	Story Points Completed	Velocity (%)
Sprint 0	10	10	100
Sprint 1	22	20	91
Sprint 2	25	23	92
Sprint 3	27	27	100
Sprint 4	16	15	94

Average velocity ≈ **95 %**. The steady completion trend demonstrates balanced scope management and effective sprint retrospectives.

KPI Summary:

KPI	Target	Achieved	Evidence
Functional Coverage	100 % MVP stories	100 %	Jira Sprint Review
Build Success Rate	≥ 95 %	97 %	GitLab CI Logs
Unit Test Coverage	≥ 80 %	82 %	Jacoco Report
Performance TPS	≥ 30	35 TPS (avg)	JMeter Summary
Mean Latency	≤ 200 ms	~180 ms	Grafana Panel
Defect Leakage	≤ 5 %	< 3 %	QA tracking sheet

AI Code Coverage:

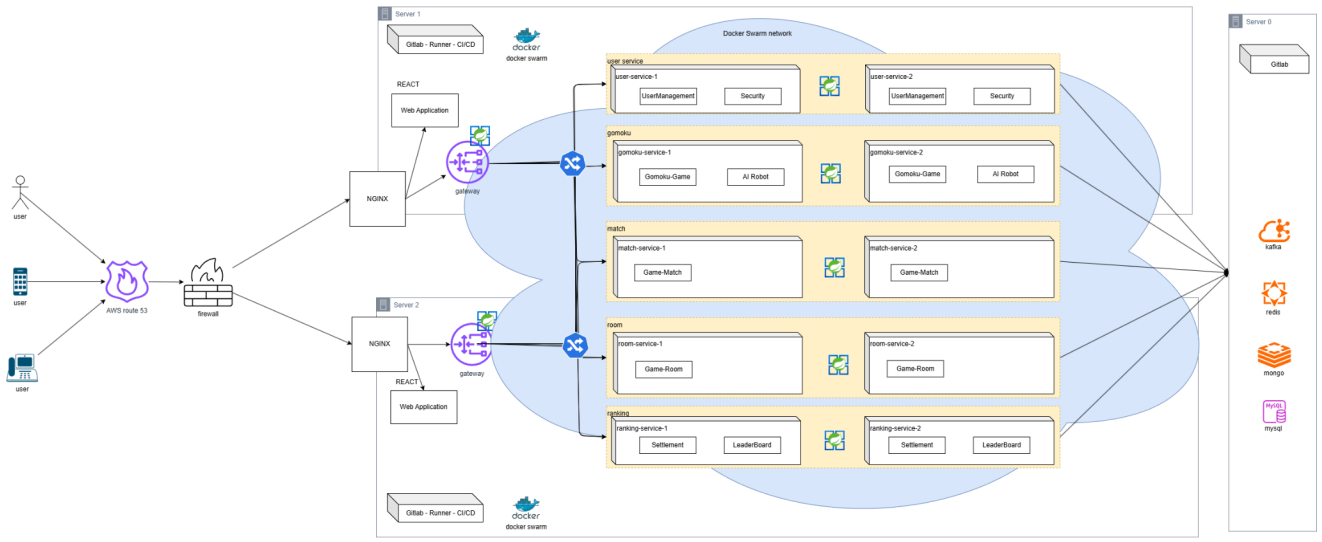
frontend: 1-2%

backend: less than 15% ~ 20%

3. System Design

3.1 Software Architecture

3.1.1 Physical Architecture Diagram



3.1.1.1 User Entry & Web Layer (Servers 1 & 2)

- **User Access**:
  - Users (on devices like desktops, laptops, or mobile phones) connect to the system via the internet.
- **Security & DNS**:
  - The initial request is managed by **AWS Route 53** (for DNS routing) and passes through a **Firewall** for security screening.
- **Load Balancing & High Availability**:
  - The traffic is then load-balanced across two identical physical servers: **Server 1** and **Server 2**. This dual-server setup ensures High Availability (HA); if one server fails, the other can continue to handle the full load.
- **Reverse Proxy & Frontend**:
  - On each server, **NGINX** acts as a reverse proxy. It serves the static **React Web Application** (the user interface) to the user and routes all API requests to the appropriate backend gateway.
- **Gateway**:
  - Each server runs a **Gateway** service. This gateway is the single entry point into the backend application logic, responsible for routing, authentication, and rate limiting.

3.1.1.2 Application Service Layer (Docker Swarm Network)

- **Microservice Orchestration**:
  - The gateways forward requests to the core application layer, which is managed as a Docker Swarm network.
- **Service Redundancy**:
  - This layer runs the application's business logic, which is broken down into several microservices. Critically, each service runs two instances (e.g., user-service-1 and user-service-2). This redundancy ensures that the failure of a single container does not take down the service.
- **Core Services**: The diagram shows five key microservices, all related to a "Gomoku" (a strategy game):
  - **user-service**:
    - Manages user accounts, authentication, and security.
  - **gomoku-service**:
    - Contains the main game logic, including an "AI Robot" for single-player mode.
  - **match-service**:
    - Handles game matching and watching.
  - **room-service**:
    - Manages game rooms and player lobbies.
  - **ranking-service**:
    - Manages leaderboards and post-game settlement.

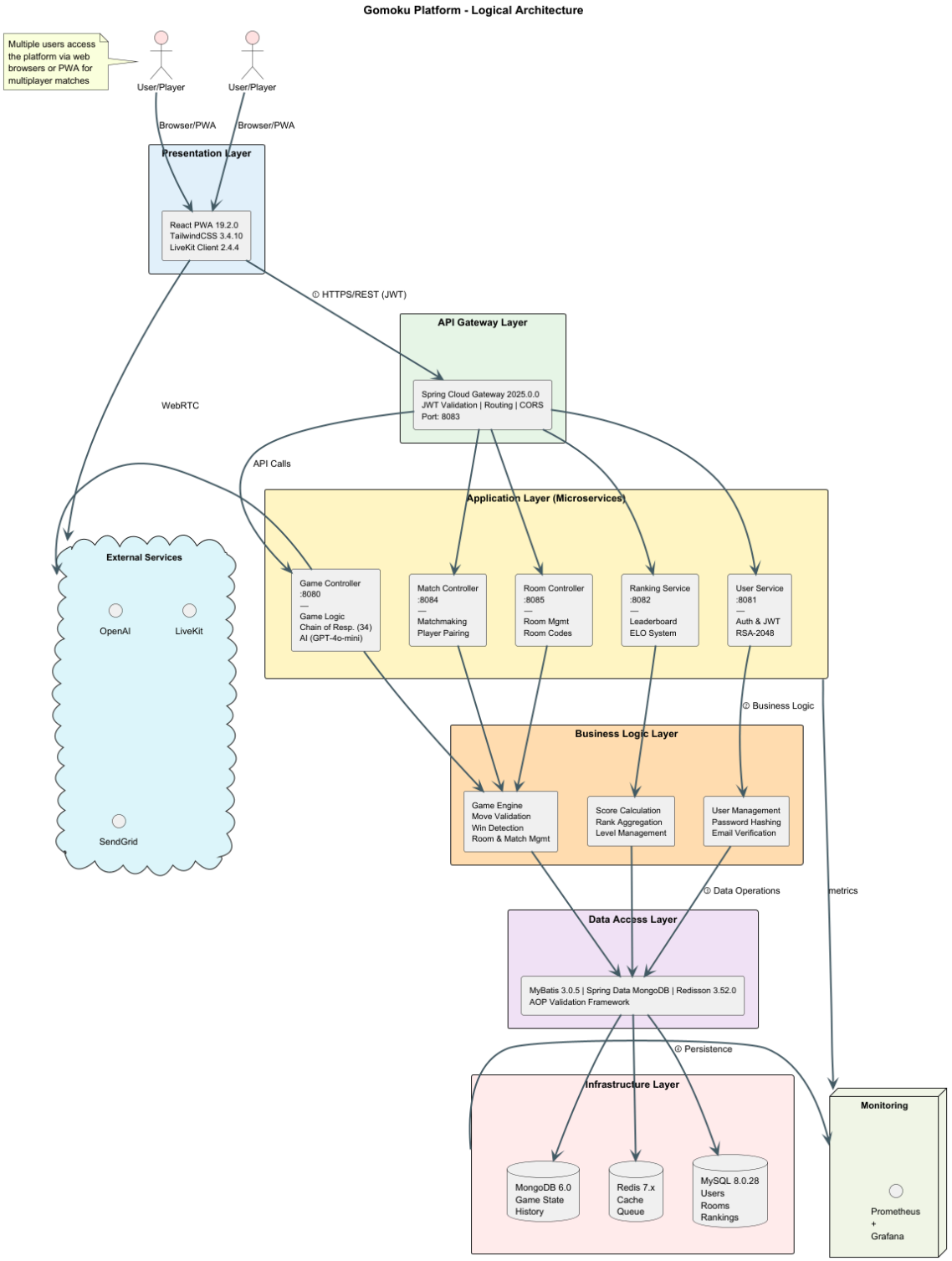
3.1.1.3 Backend Data & Infrastructure Layer (Server 0)

- **Separation of Concerns**: The application services (which are stateless) communicate with a dedicated Server 0, which holds all the stateful components. This separation is a best practice, allowing the application layer to be scaled independently from the data layer.
- **Data & Infrastructure Components**: Server 0 hosts the system's core data and messaging platforms:
  - **Redis**: An in-memory cache, likely used for session management, fast-access data (like leaderboards), or real-time game state.
  - **Mongo (MongoDB)**: A NoSQL database, ideal for storing flexible data like game logs or user profiles.
  - **MySQL**: A relational database, likely used for structured data such as user credentials, rankings, and financial transactions (settlement).

3.1.1.4 CI/CD Pipeline

- The diagram also notes **Gitlab Runner - CI/CD** on Servers 1 and 2. This indicates that the project utilizes an automated **Continuous Integration/Continuous Deployment** pipeline to build, test, and deploy code changes to the Docker Swarm environment.

3.1.2 Logic Architecture Diagram



Layer	Technology
Frontend	React 19.2.0
Gateway	Spring Cloud 2025.0.0
Backend	Spring Boot 3.5.5, Java 17
Data	MyBatis 3.0.5
Cache	Redisson 3.52.0

Patterns: Chain of Resp. (34), Service Layer, Repository
Quality: 965 Tests, 81% Coverage, 38 TPS, 147ms P50

3.1.2.1 Overview

The Gomoku Platform implements a layered microservices architecture that separates concerns across six distinct logical layers, promoting modularity, scalability, and maintainability. This architectural approach enables independent development, deployment, and scaling of individual services while maintaining clear separation between presentation, business logic, and data persistence layers.

3.1.2.2 Layer 1: Presentation Layer

The **Presentation Layer** serves as the user-facing interface, implemented as a Progressive Web Application (PWA) using React 19.2.0. This layer is responsible for:

- **User Interface Rendering**: Built with React's latest concurrent rendering features and styled using TailwindCSS 3.4.10 for responsive design across desktop and mobile devices
- **Real-time Communication**: Integrates LiveKit Client 2.4.4 for WebRTC-based voice chat, enabling players to communicate during matches
- **Offline Capabilities**: PWA features allow users to install the application and access certain features offline through service workers
- **State Management**: Client-side state management handles user authentication context, game state synchronization, and UI updates

Multiple users simultaneously access the platform through web browsers or as an installed PWA, creating a multi-player gaming environment.

3.1.2.3 Layer 2: API Gateway Layer

The **API Gateway Layer** acts as the single entry point for all client requests, implemented using Spring Cloud Gateway 2025.0.0 (Port 8083). This layer provides:

- **Request Routing**: Intelligent routing of incoming requests to appropriate microservices based on URL patterns:
  - /api/user/* → User Service
  - /api/gomoku/game/* → Game Controller
  - /api/gomoku/room/* → Room Controller
  - /api/gomoku/match/* → Match Controller
  - /api/ranking/* → Ranking Service
- **Security Enforcement**: JWT token validation ensures that only authenticated users can access protected resources
- **Cross-Cutting Concerns**: Handles CORS policies, rate limiting, and request/response logging

- **Protocol Translation**: Bridges between client HTTP/REST requests and internal microservice communication

3.1.2.4 Layer 3: Application Layer (Microservices)

The **Application Layer** comprises five independent microservices, each running on dedicated ports and handling specific business domains:

User Service (Port 8081)

- **Authentication & Authorization:**
  - Manages user registration, login, and JWT token generation using HS256 algorithm
- **Security Features:**
  - Implements RSA-2048 encryption for password transmission and Bcrypt hashing for password storage
- **Profile Management:**
  - Handles user profile updates, password resets, and email verification

Game Controller (Port 8080)

- **Core Game Logic:**
  - Processes player moves, validates game rules, and detects win conditions
- **Chain of Responsibility Pattern:**
  - Implements 34 specialized chains (18 validation chains + 16 execution chains) for handling various game actions (move, undo, restart, surrender, draw, timeout)
- **AI Integration:**
  - Integrates with OpenAI GPT-4o-mini API for AI-powered move suggestions in practice mode
- **Game State Management:**
  - Maintains real-time game state and synchronizes with clients via polling

Room Controller (Port 8085)

- **Room Lifecycle Management:**
  - Handles room creation, joining, and leaving operations
- **Room Code System:**
  - Generates and validates unique 6-character room codes with 30-minute TTL
- **Room Configuration:**
  - Manages room settings including board size, time limits, and game modes

Match Controller (Port 8084)

- **Matchmaking Queue:**
  - Implements Redis-based queue for pairing players with similar skill levels
- **Player Pairing Algorithm:**
  - Matches players based on ELO ratings and queue wait time
- **Match Settlement:**
  - Finalizes match results and triggers ranking updates

Ranking Service (Port 8082)

- **Leaderboard Management:**
  - Aggregates and ranks players based on ELO ratings
- **Score Calculation:**
  - Implements ELO rating system with K-factor adjustments based on player level
- **Level System:**
  - Manages player progression through Bronze, Silver, Gold, Platinum, Diamond, Master tiers

3.1.2.5 Layer 4: Business Logic Layer

The **Business Logic Layer** encapsulates domain-specific logic separated from controllers and data access:

- **User Management**: Password hashing, email verification workflows, and token refresh logic
- **Game Engine**: Move validation algorithms, win detection patterns (five-in-a-row with eight directions), and game state transitions
- **Room & Match Management**: Room code generation algorithms, matchmaking strategies, and match settlement rules
- **Score Calculation**: ELO rating formulas, rank aggregation logic, and level progression thresholds

This layer implements core design patterns including Chain of Responsibility (34 implementations), Strategy Pattern (for AI difficulty levels), and Service Layer Pattern (for transaction management).

3.1.2.6 Layer 5: Data Access Layer

The **Data Access Layer** abstracts database interactions through standardized interfaces:

- **MyBatis 3.0.5**:
  - Provides SQL mapping for MySQL operations with dynamic query generation and result set mapping
- **Spring Data MongoDB**:
  - Offers repository abstractions for document-based storage of game states and history
- **Redisson 3.52.0**:
  - Supplies distributed data structures (queues, caches, locks) for Redis operations
- **AOP Validation Framework**:
  - Implements cross-cutting validation concerns using aspect-oriented programming, including BasicCheck and ValueChecker aspects

This layer ensures database independence, simplifies query management, and provides consistent error handling across all data operations.

3.1.2.7 Layer 6: Infrastructure Layer

The Infrastructure Layer comprises the persistent data stores:

- **MySQL 8.0.28:**
  - Relational database storing structured data including user accounts, game room metadata, and ranking information across four primary tables (user, game_room, ranking, user_token)
- **MongoDB 6.0:**
  - Document database storing flexible, schema-less data including complete game states and historical game records in two collections (game_documents, game_history_documents)
- **Redis 7.x:**
  - In-memory data store providing high-performance caching, session management, room codes (with 30-minute TTL), matchmaking queues, and leaderboard caching

3.1.2.8 External Services Integration

The platform integrates with three external services:

- **OpenAI API:**
  - Provides GPT-4o-mini powered AI move suggestions for practice mode, analyzing board state and recommending optimal moves
- **LiveKit Server:**
  - Manages WebRTC media routing through Selective Forwarding Unit (SFU) architecture for voice chat functionality

- **SendGrid:**
  - Delivers transactional emails for user verification, password resets, and account notifications

3.2 Transition from Analysis to Design

During the analysis phase, we identified conceptual entities (Player, Room, Board, Move) and their interactions through use case diagrams and domain modeling. The transition to design phase transformed these abstract concepts into concrete, implementable structures following engineering best practices.

Transition Objectives:

1. Map analysis entities to design classes with appropriate patterns
2. Define clear boundaries between layers (Presentation, Application, Data)
3. Select appropriate technologies for each architectural concern
4. Ensure traceability from requirements to implementation

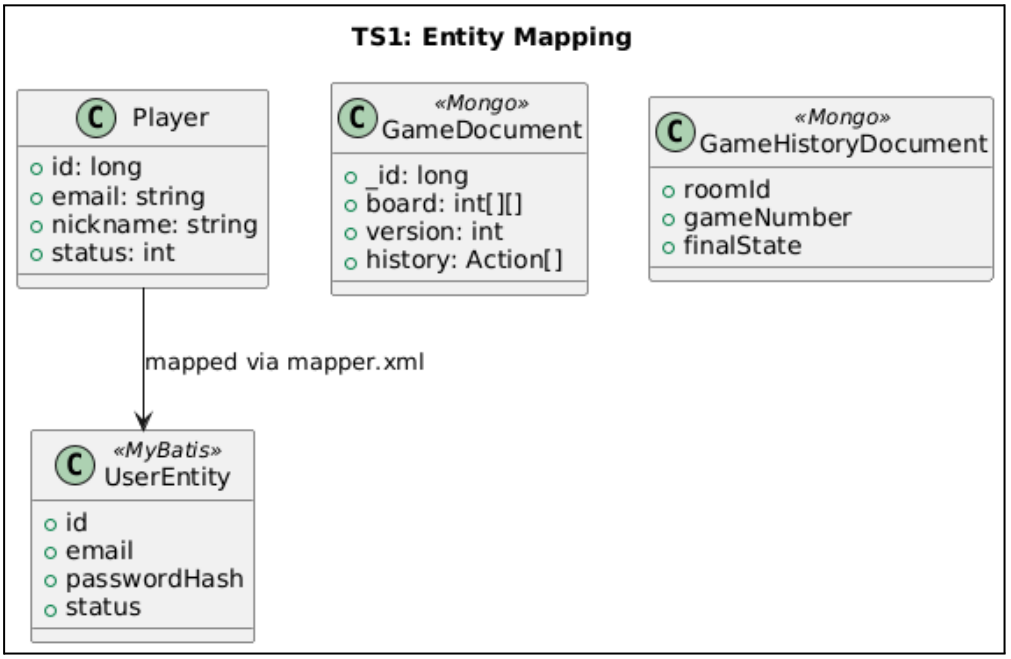
3.2.1 Transition Strategy N

ID	From (Analysis)	To (Design)	Rationale
TS1	Player, GameRecord entities	@Entity JPA classes with Spring Data	ORM persistence, automatic query generation, transaction management
TS2	Actor → System boundary interactions	REST @RestController classes	Define API entry points, request validation, response formatting
TS3	GameFlow, MoveHandler control objects	Service layer classes (@Service)	Encapsulate business logic, improve testability, enable AOP
TS4	Player ↔ Opponent event broadcast	WebSocket broker + Observer pattern	Real-time bidirectional communication, event-driven updates

TS5	Move validation & win detection logic	Chain of Responsibility pattern	Decouple validation steps, extensible rule addition
TS6	Room state management	Redis caching with TTL	Fast in-memory storage, automatic expiration, atomic operations

3.2.1.1 TS1 - Entity → Persistence Models (Player, GameRecord → JPA/MyBatis/Mongo)

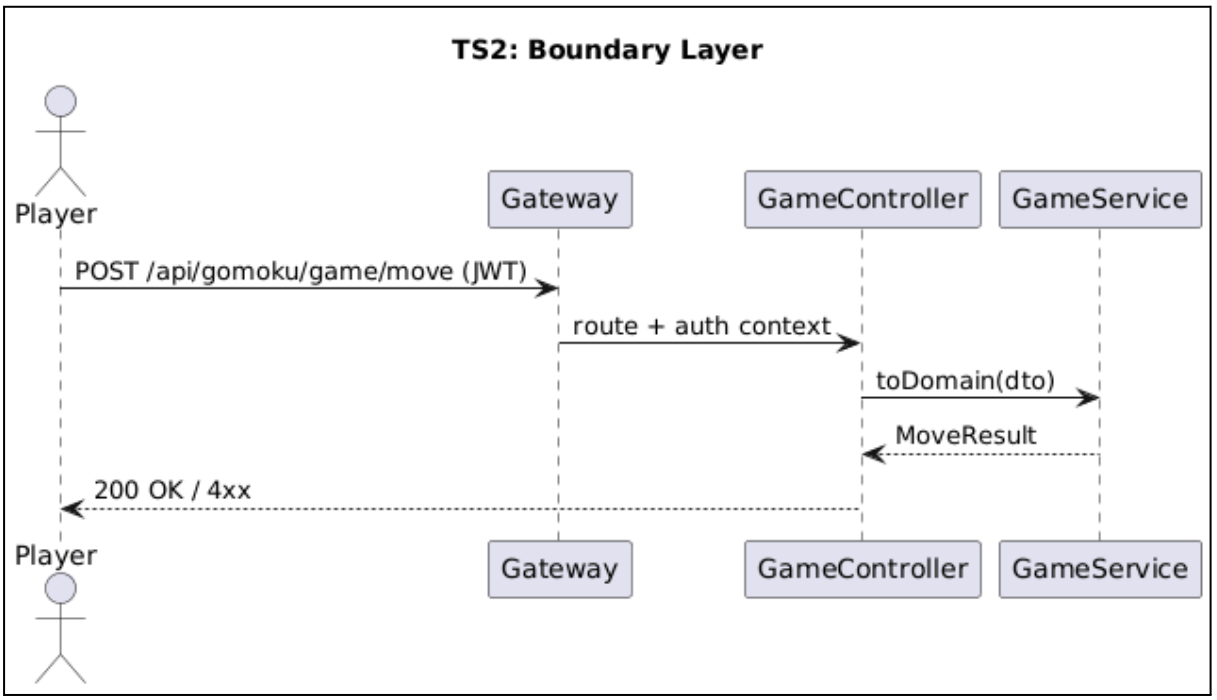
- Static change: analysis entities became MyBatis entities for MySQL (users/rooms/ranking) and Mongo documents for live games/history.
- Rationale: transactional updates for ranks & profile data; flexible, append-only actionHistory in Mongo for replay/undo.
- Dynamic impact: sequences add optimistic version checks during updates to prevent double-moves.



3.2.1.2 TS2 - Actor Interactions → REST Boundaries (@RestController)

- Static change: boundary classes per use case (AuthController, RoomController, GameController…).

- Rationale: keep HTTP concerns (validation, DTOs, 401/403, idempotency) out of business logic.
- Dynamic impact: sequences now start with JWT verification and DTO mapping; errors are standardized problems+json



3.2.1.3 TS3 - Control Objects → Services (@Service)

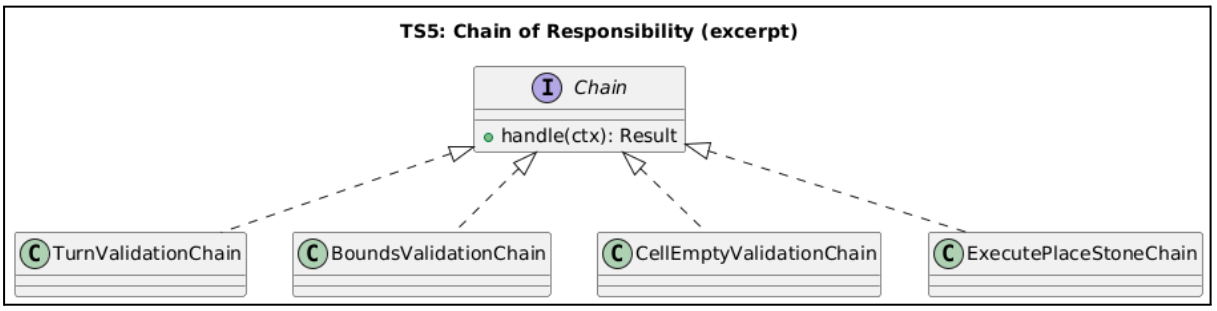
- Static change: analysis “MoveHandler” becomes GameBizService, “Matchmaker” becomes MatchService.
- Rationale: SRP & testability; hide data sources behind repositories/adapters; enable AOP for cross-cutting validations.
- Dynamic impact: services participate in transactions for ranked settlement and emit metrics.

3.2.1.4 TS4 - Player↔Opponent Events → Observer/WebSocket + Polling/SSE

- Decision: deterministic state via polling/SSE (every 1–2 s) to avoid race amplification; WS for chat/voice.
- Trade-off: slightly higher bandwidth but far simpler debugging and replay fidelity.

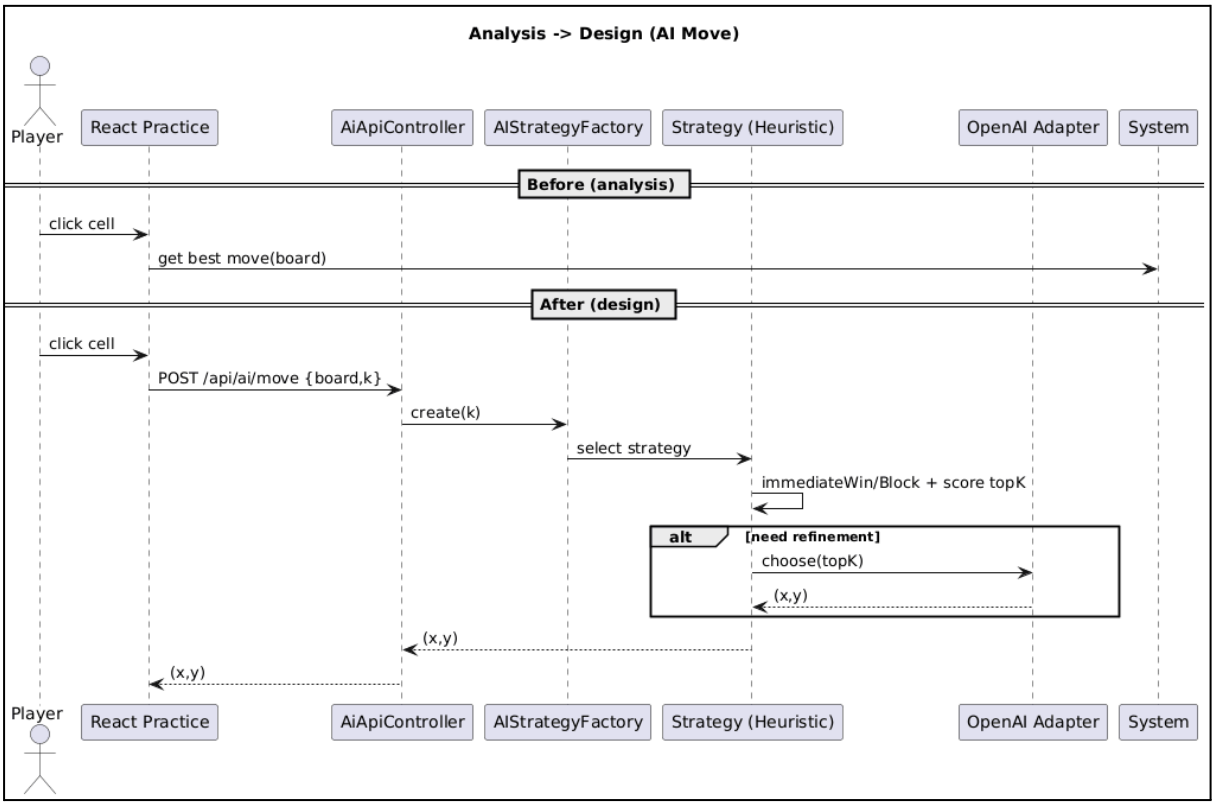
3.2.1.5 TS5 - Move Validation & Execution → Chain of Responsibility

- Static change: 34 chains grouped as “validate*” and “execute*”.
- Rationale: extensible rulebook; unit tests per rule.
- Dynamic impact: fail-fast validation; execution chains manage board updates, history push, win detection.



3.2.1.6 TS6 — Room State & Matchmaking → Redis + TTL/Locks

- Static change: room codes cached with TTL (room:code:*), matchmaking queues (match:q:*), distributed locks (game:lock:{roomId}).
- Rationale: O(1) lookups, back-pressure, auto-expiry to prevent ghost rooms.
- Dynamic impact: sequences add lock-acquire/release around critical sections; graceful degradation to DB if cache fails.



Explanation of the AI Flow Transition (Analysis → Design)

1. Analysis Model (Left Side)

In the analysis phase, we work with conceptual domain objects and actors without specifying frameworks, patterns, technologies, or class responsibilities.

- Player represents the actor initiating a move.
- Game models the board state and game rules at a conceptual level.
- AIEngine is a black-box abstraction that generates an AI move when requested.

At this stage:

- There is no notion of controllers, services, factories, or concrete algorithms.
- The AIEngine simply appears as “something that produces a move,” because analysis describes behaviour, not implementation.

2. Design Model (Right Side)

During design, we refine conceptual objects into technical components with clear responsibilities, interfaces, and interaction patterns.

The analysis model is expanded into:

a. GameController (Boundary Layer)

- Accepts HTTP requests from the frontend.
- Converts incoming requests into domain-friendly structures.
- Triggers the AI move retrieval workflow.

b. GameService (Application Logic Layer)

- Coordinates request processing.

- Manages the board state, validation, and turn updates.
- Decides when the AI move must be computed.

It acts as the high-level “orchestrator,” separating workflow from detailed move generation logic.

c. AIStrategyFactory (Pattern Application)

The single conceptual “AIEngine” becomes a Factory, enabling multiple AI styles:

- DefensiveAI
- AggressiveAI
- BalancedAI

The factory:

- Selects which strategy to use based on difficulty/K-value.
- Ensures the design stays open for extension, closed for modification (OCP).

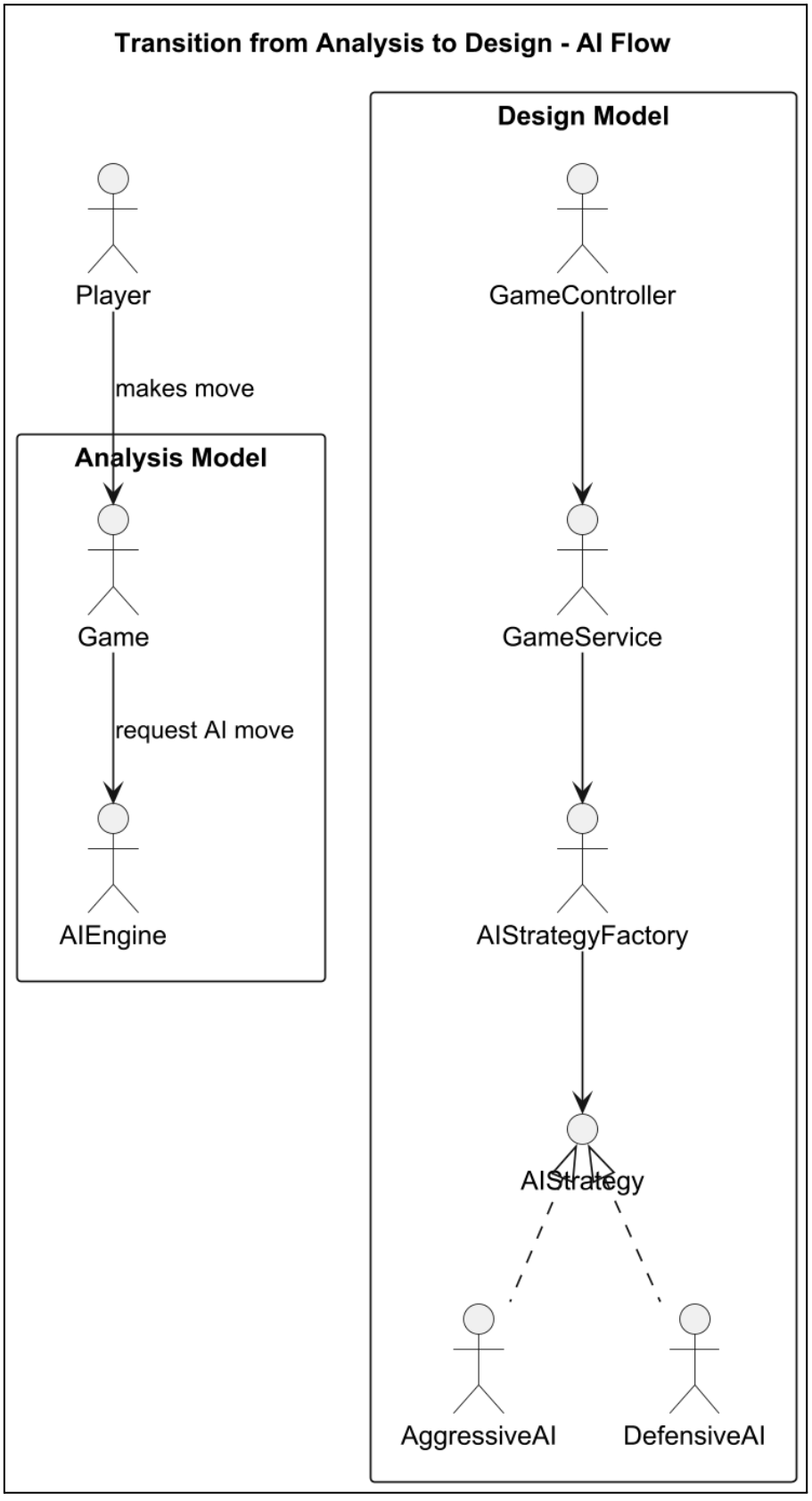
3. Why This Transition Matters

This transition demonstrates how:

- One analysis object (AIEngine) becomes multiple cooperating design components

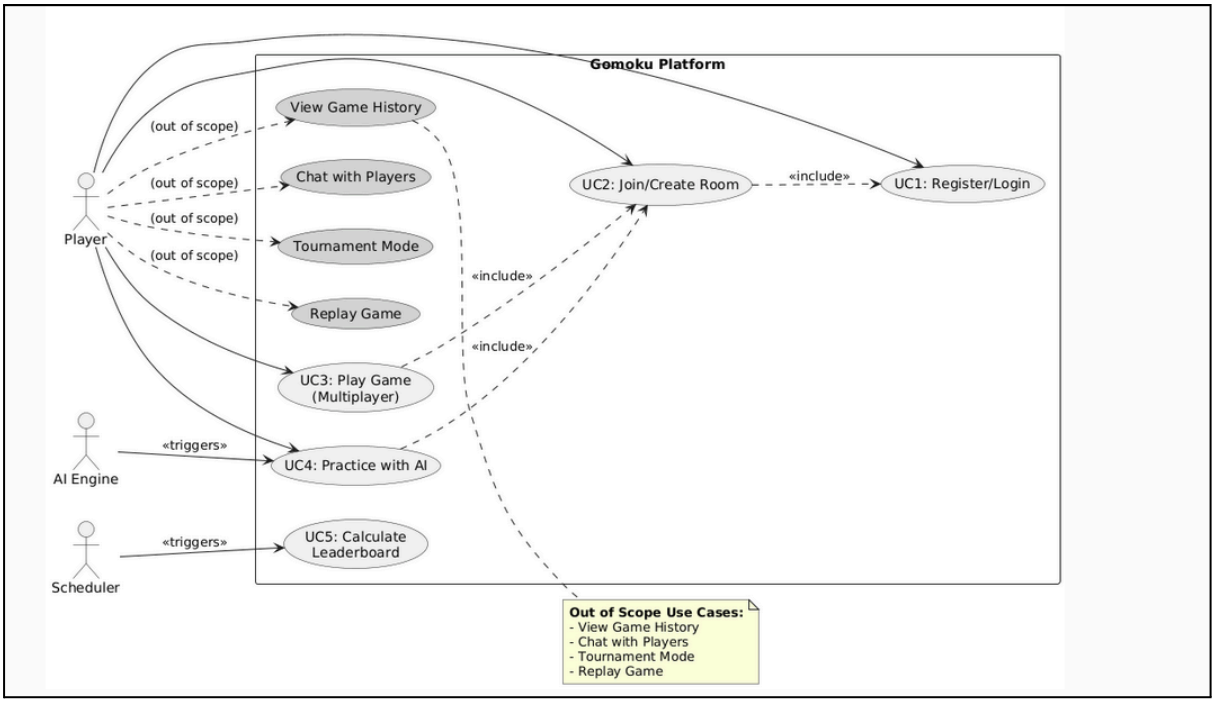
  (Controller → Service → Factory → Strategy).
- Behaviour described in the use case (“player requests an AI move”) becomes a testable and modular workflow.
- Design patterns (Factory + Strategy) reduce coupling and improve extensibility.
- The system remains maintainable, scalable, and ready for future enhancements

  (e.g., adding a PredictiveAI or NeuralAI strategy without modifying existing code).


Transition from Analysis to Design - AI Flow
Design Model
Player
GameController
makes move
Analysis Model
Game
GameService
request AI move
AIEngine
AIStrategyFactory
AIStrategy
AggressiveAI
DefensiveAI

3.3 Use Case Model

3.3.1 Use Case Diagram



3.3.2 Use Case Description

ID	Name	Normal Flow	Exceptional Flow
UC1	Register/Log in	User submits credentials → JWT issued → redirect to Lobby	Invalid credentials → error toast
UC2	Join/Create Room	Player enters code → Redis validates → room established	Room full/invalid → error message
UC3	Play Game (Multiplayer)	Players take turns → WebSocket broadcasts moves → win check → DB record	Network disconnect → room timeout
UC4	Practice with AI	Player selects K value → AIEngine computes move → update board	AIEngine error → fallback defensive mode

UC5	View Leaderboard	REST request → Service fetches top scores → display UI	DB error → empty state

3.3.2.1 UC1

UC1: User creates a new account or authenticates with existing credentials to access the platform.

The normal flow for the UC1:

1. User navigates to the platform homepage
2. User clicks "Login" or "Register" button
3. Branch: Register

   3a. User enters email, nickname, and password

   3b. System validates email format and password strength (min 8 characters)

   3c. System encrypts password with RSA-2048 during transmission

   3d. System hashes password with BCrypt (salt rounds = 10)

   3e. System inserts new user record into MySQL user table with status=0 (inactive)

   3f. System sends verification email via SendGrid

   3g. User clicks verification link in email

   3h. System updates user.status=1 (active)

   3i. Continue to step 4 (Login flow)

4. Branch: Login

   4a. User enters email and password

   4b. System encrypts password with RSA-2048 during transmission

   4c. System queries MySQL user table by email

   4d. System verifies BCrypt password hash matches

   4e. System checks user.status=1 (active)

5. System generates JWT access token (HS256 algorithm, 15-minute expiration)
6. Payload: { userId, email, nickname, exp: now + 15min }
7. System generates JWT refresh token (UUID, 30-day expiration)
8. System inserts/updates user_token table with refresh_token and expires_at
9. System creates Redis session:

   Key: session:{userId}

   Value: { accessToken, loginTime, ipAddress, deviceInfo }

TTL: 24 hours

10. System returns tokens to client:

{

"accessToken": "eyJhbGc...",

"refreshToken": "a1b2c3d4...",

"user": { "id": 12345, "email": "user@example.com", "nickname": "Player1" }

}

11. Client stores tokens in localStorage
12. Client redirects to Lobby page

The exceptional flows:

E1: Invalid Email Format (Step 3b)

System displays error toast: "Invalid email format"

System highlights email input field in red

Return to step 3a

E2: Weak Password (Step 3b)

System displays error toast: "Password must be at least 8 characters with 1 uppercase, 1 lowercase, and 1 number"

Return to step 3a

E3: Email Already Exists (Step 3e)

System catches unique constraint violation on user.email

System displays error toast: "Email already registered"

Return to step 3a

E4: Invalid Credentials (Step 4d)

System detects email not found OR password hash mismatch

System displays error toast: "Invalid email or password"

System logs failed login attempt (for security monitoring)

Return to step 4a

Note: After 5 failed attempts within 10 minutes, temporarily block IP for 15 minutes

E5: Account Inactive (Step 4e)

System detects user.status != 1

If status=0 (inactive): Display "Please verify your email address"

If status=2 (disabled): Display "Account suspended. Contact support."

If status=3 (deleted): Display "Account not found"

Return to step 4a

E6: Network Error (Any step)

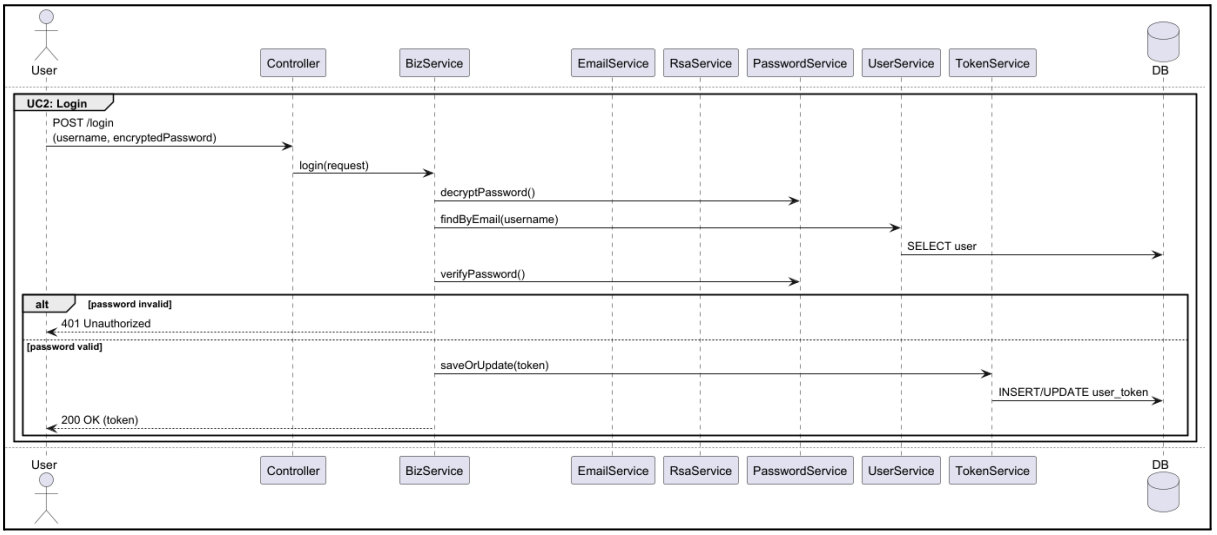
Client detects timeout or connection failure

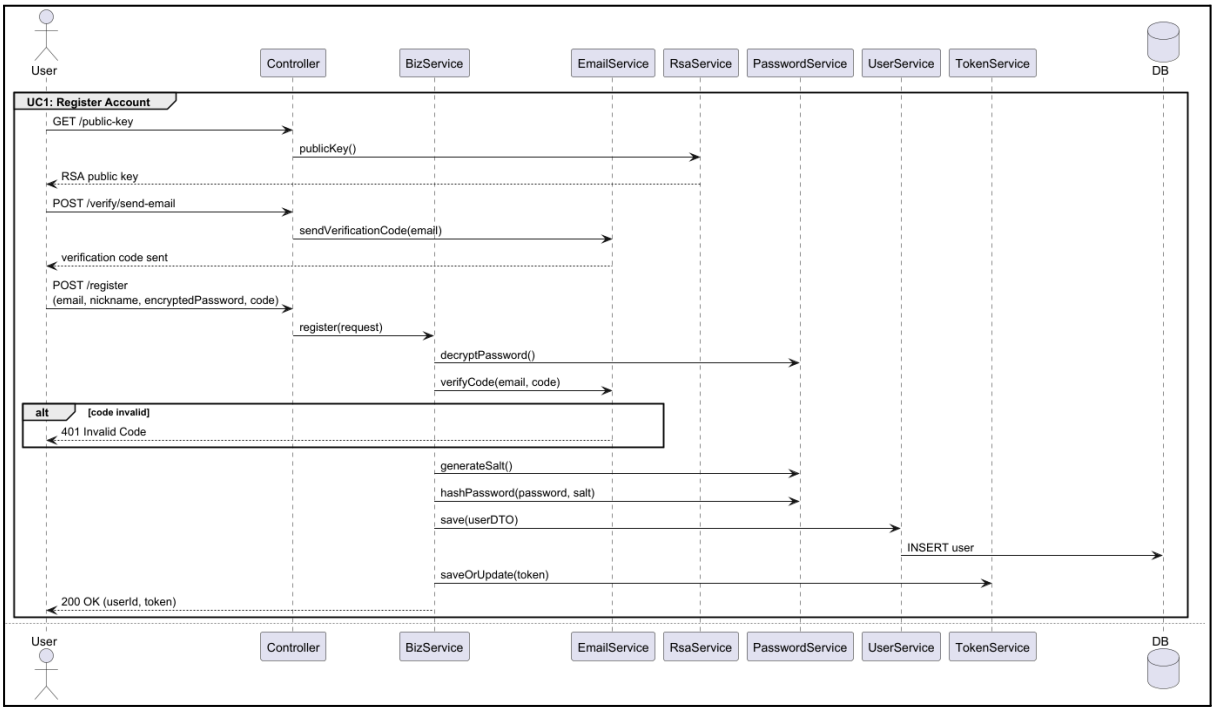
Display error toast: "Network error. Please check your connection."

Allow user to retry

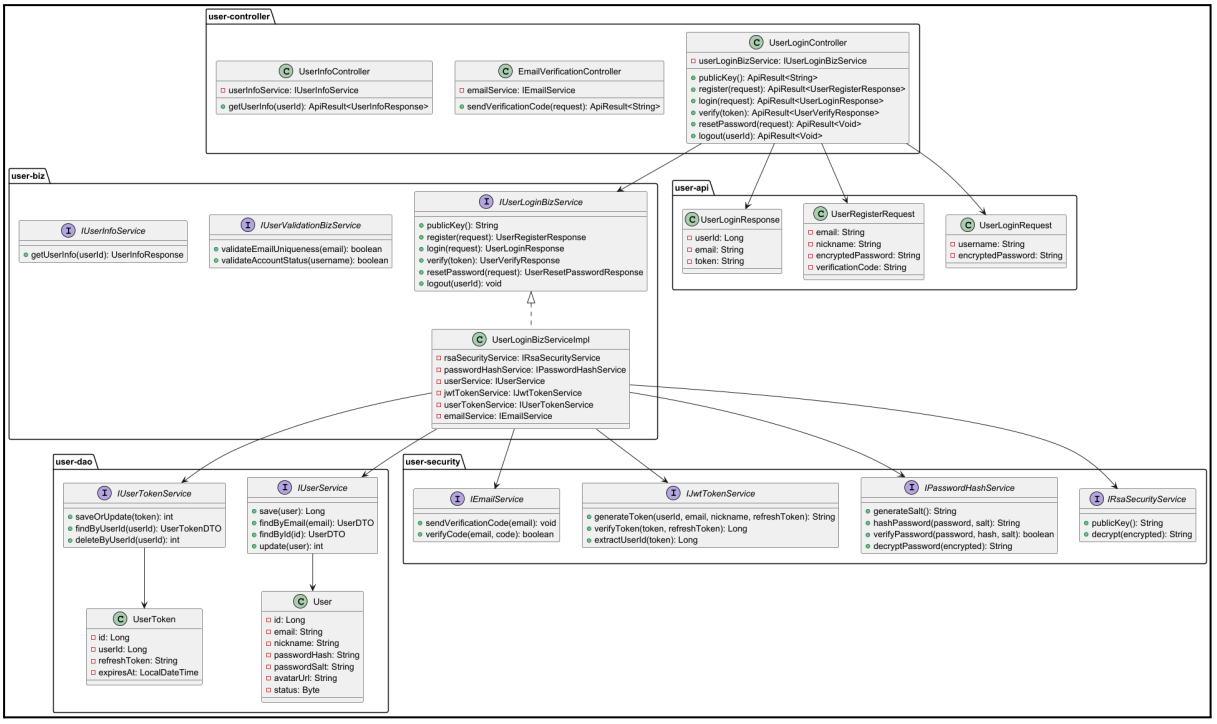
For the register/login Use Case, we have the following diagrams:

Sequence diagram:

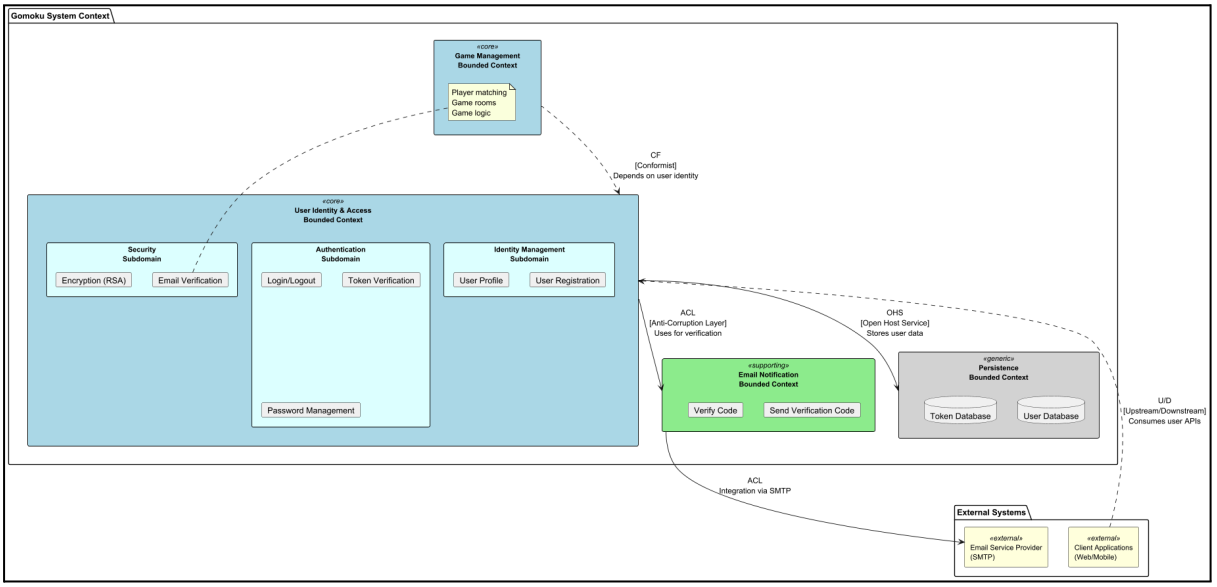




Class diagram:



DDD:



3.3.2.2 UC2

UC2:Player creates a new game room with a unique room code or joins an existing room by entering a room code.

For the create room, the normal flow:

1. Player clicks "Create Room" button
2. System displays room configuration modal:

   Game mode: Casual / Ranked / Private

   Board size: 15×15 (default) / 19×19

   Time limit: 30 sec / 60 sec / No limit
3. Player selects options and clicks "Create"
4. System generates unique 6-character room code:

   roomCode = RandomStringUtils.randomAlphanumeric(6).toUpperCase()

   // Example: "XYZ789"
5. System checks Redis for collision:

   If room:code:XYZ789 exists, regenerate (step 4)

   Else, continue
6. System creates room in MySQL:

   INSERT INTO game_room (room_code, player1_id, type, status, created_at)

   VALUES ('XYZ789', 12345, 0, 0, NOW())

   -- Returns roomId = 98765
7. System stores room code in Redis:

   SET room:code:XYZ789 "98765" EX 1800

   -- TTL: 30 minutes
8. System creates game document in MongoDB:

   GameDocument.createNewGameWithRandomBlack(

   roomId: 98765,

   playerId: 12345,

   modeType: "CASUAL"

)
9. System returns room details to client:

   {

   "roomId": 98765,

"roomCode": "XYZ789",
"mode": "CASUAL",
"status": "WAITING",
"player1": { "id": 12345, "nickname": "Player1" }
}

10. Client redirects to Game Room page (/room/98765)
11. Client displays: "Room Code: XYZ789 - Waiting for opponent..."
12. Client starts polling /api/game/state every 2 seconds

For Join Room, the normal flow:

1. Player clicks "Join Room" button
2. System displays input modal: "Enter Room Code"
3. Player enters room code (e.g., "XYZ789") and clicks "Join"
4. System queries Redis:
   GET room:code:XYZ789
   -- Returns "98765"
5. System queries MySQL to validate room:
   SELECT id, status, player1_id, player2_id, type
   FROM game_room
   WHERE id = 98765
6. System validates:
   Room exists
   Room status = 0 (WAITING)
   player2_id IS NULL (room not full)
7. System updates room in MySQL:
   UPDATE game_room
   SET player2_id = 67890, status = 1, updated_at = NOW()
   WHERE id = 98765
8. System updates game document in MongoDB:
   updateOne(
   { _id: 98765 },
   { $set: { whitePlayerId: 67890 } }
)

9. System refreshes Redis TTL:
   EXPIRE room:code:XYZ789 1800
10. System returns room details to client
11. Client redirects to Game Room page (/room/98765)
12. Both players now see "Ready" button

And the exceptional flow:

E1: Invalid Room Code Format (Step 3)
System detects code is not 6 alphanumeric characters
Display error toast: "Invalid room code format"
Return to step 2

E2: Room Code Not Found (Step 4)
Redis returns NULL for room:code:XYZ789
Display error toast: "Room not found or expired"
Return to step 2

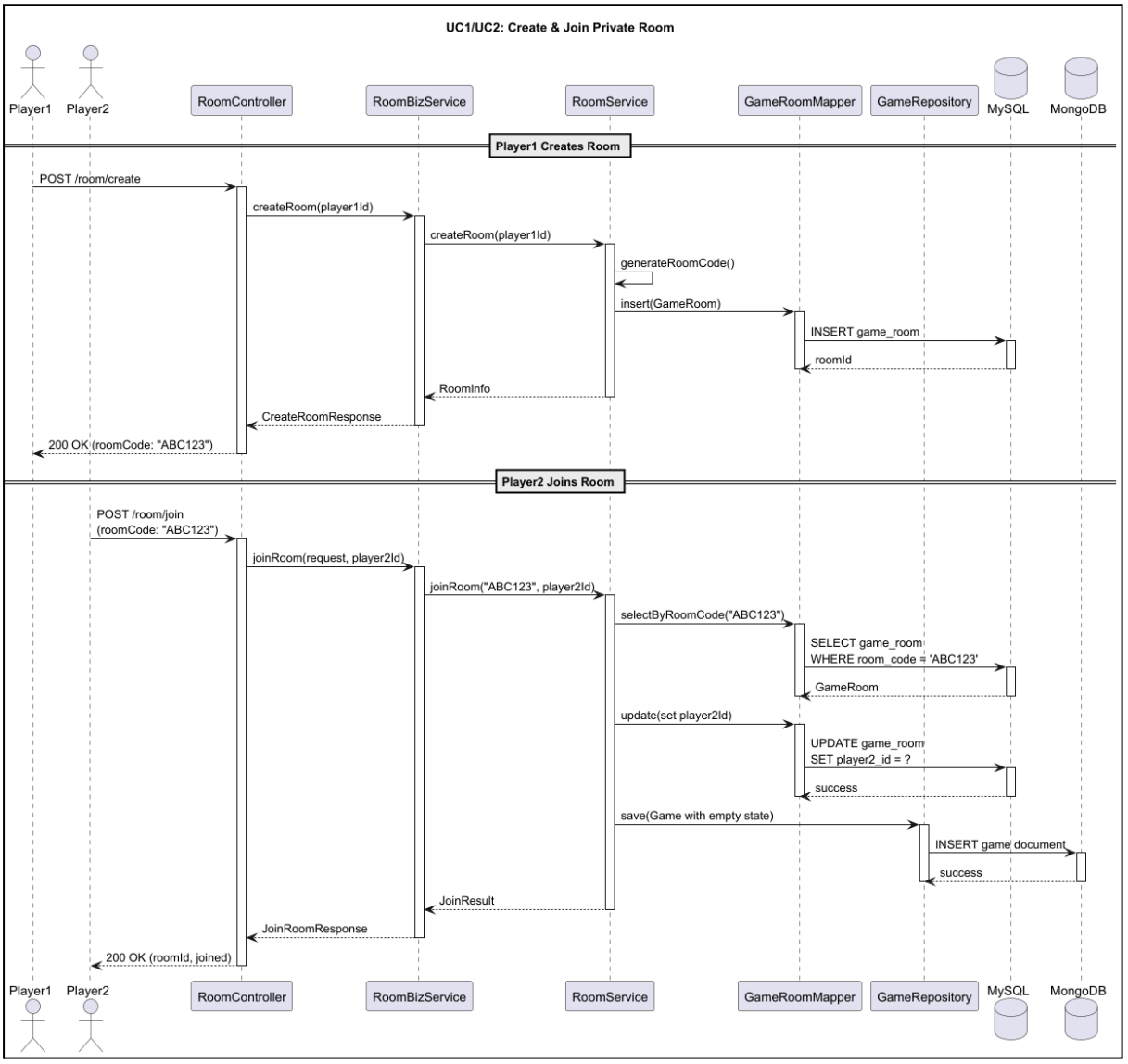
E3: Room Already Full (Step 6)
System detects player2_id IS NOT NULL
Display error toast: "Room is full"
Return to step 2

E4: Room Already Started (Step 6)
System detects status != 0 (not WAITING)
Display error toast: "Game already in progress"
Return to step 2

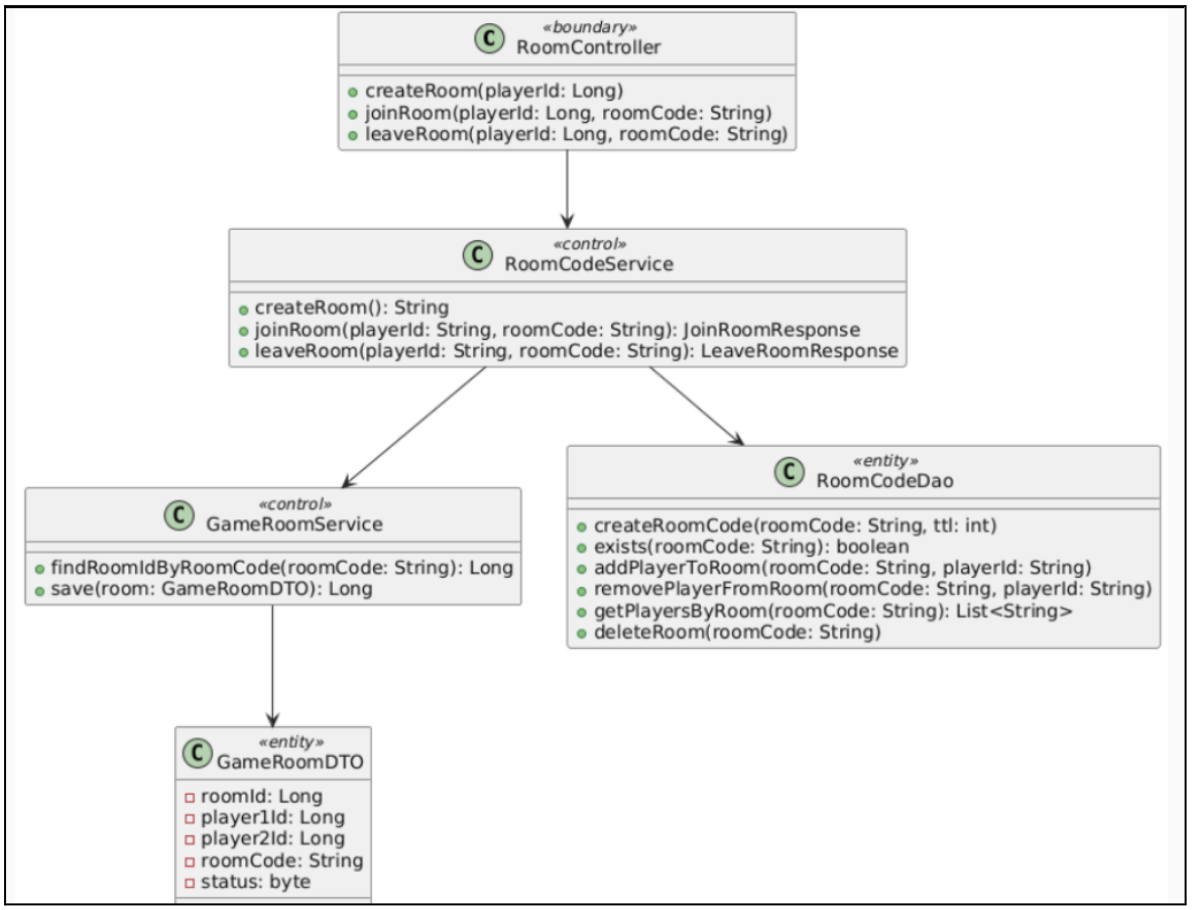
E5: User Already in Another Room (Step 6/7)
System queries if player is already in an active room
Display error toast: "Please leave your current room first"
Offer button: "Leave Current Room"
Return to step 2

For the join/create Use Case, we have the following diagrams:

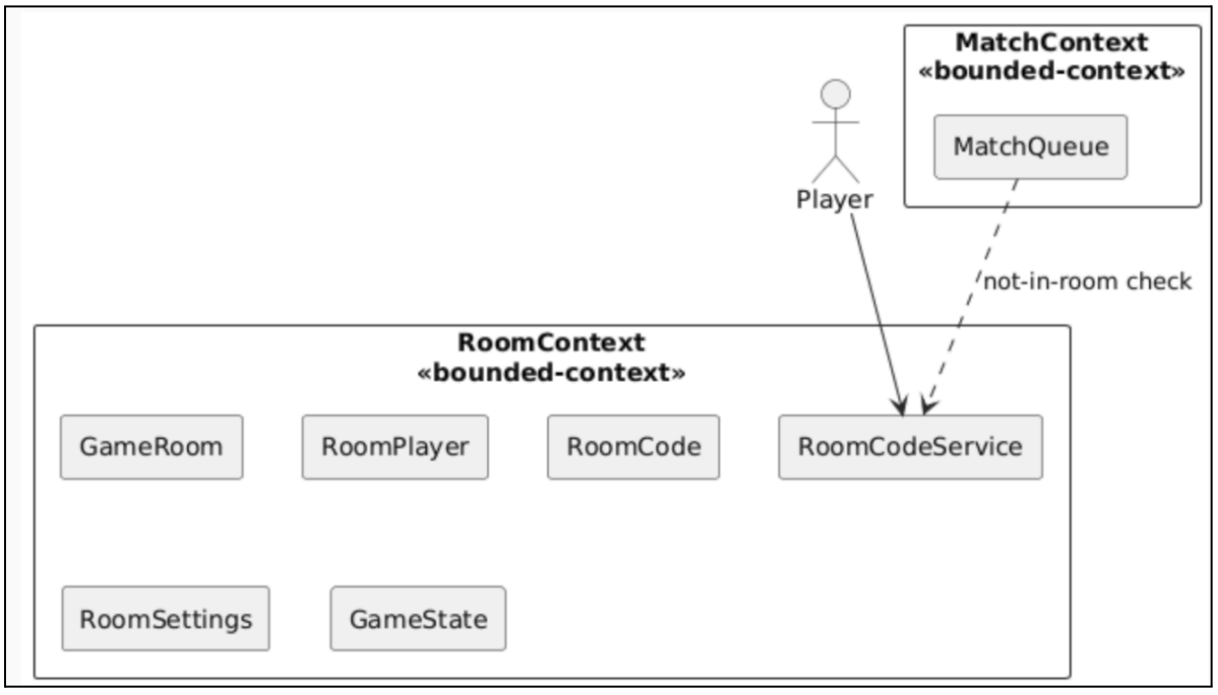
Sequence diagram:


UC1/UC2: Create & Join Private Room
Player1
Player2
RoomController
RoomBizService
RoomService
GameRoomMapper
GameRepository
MySQL
MongoDB
Player1 Creates Room
POST /room/create
createRoom(player1Id)
createRoom(player1Id)
generateRoomCode()
insert(GameRoom)
INSERT game_room
roomId
RoomInfo
CreateRoomResponse
200 OK (roomCode: "ABC123")
Player2 Joins Room
POST /room/join
(roomCode: "ABC123")
joinRoom(request, player2Id)
joinRoom("ABC123", player2Id)
selectByRoomCode("ABC123")
SELECT game_room
WHERE room_code = 'ABC123'
GameRoom
update(set player2Id)
UPDATE game_room
SET player2_id = ?
success
save(Game with empty state)
INSERT game document
success
JoinResult
JoinRoomResponse
200 OK (roomId, joined)

Class diagram:



DDD:



3.3.2.3 UC3

UC3:Two players take turns placing pieces on the board with real-time synchronization, win condition checking, and game result recording.

The normal flow for UC3:

1. Ready Phase:

   1a. Player A clicks "Ready" button

   1b. System updates MongoDB: GameDocument.blackReady = true

   1c. Player B clicks "Ready" button

   1d. System updates MongoDB: GameDocument.whiteReady = true

   1e. System updates room status: game_room.status = 2 (PLAYING)

   1f. System updates game status: GameDocument.status = PLAYING

   1g. Both clients receive status update via polling

2. Game Start:

   2a. System determines starting player (black goes first)

   2b. Client displays: "Your Turn" (for black) / "Opponent's Turn" (for white)

   2c. Client enables board clicks for current player

3. Player A Makes Move (Black's Turn):

   3a. Player A clicks board position (row: 7, col: 8)

   3b. Client sends request:

   POST /api/gomoku/game/move

   Authorization: Bearer {accessToken}

   Body: {

   "roomId": 98765,

   "position": [7, 8],

   "version": 41

   }

4. Server Processes Move:

   4a. Gateway validates JWT token

   4b. GameController routes to GameBizService

   4c. System acquires distributed lock: redis.getLock("game:lock:98765")

   4d. System reads game state from MongoDB (with version check)

   4e. System executes 34 validation chains:

TurnValidationChain: Verify it's black's turn
BoardSizeValidationChain: Position within bounds (0-14)
GameStatusValidationChain: Game is PLAYING
PositionEmptyValidationChain: Cell not occupied
MoveTimeoutValidationChain: Move within time limit
... (18 validation chains total)

5. Update Game State:
   5a. System updates board: currentState.board[7][8] = 1 (black)
   5b. System appends to action history:
   actionHistory.push({
     actionType: "MOVE",
     playerColor: "BLACK",
     position: [7, 8],
     timestamp: 1699999999999
   })
   5c. System sets last action (for frontend animation)
   5d. System switches turn: currentState.currentPlayer = 2 (white)
   5e. System increments total moves: currentState.totalMoves++
6. Check Win Condition:
   6a. System runs win detection algorithm (checks 8 directions from [7,8])
   6b. If five-in-a-row detected:
   Set currentState.winner = 1 (black won)
   Set GameDocument.status = FINISHED
   Go to step 9 (Game End)
   6c. Else, continue
7. Save to MongoDB (Optimistic Locking):
   result = updateOne(
     { _id: 98765, version: 41 }, // Version check
     {
       $set: { currentState, lastAction, updateTime },
       $push: { actionHistory: newAction },
       $inc: { version: 1 } // Increment to 42
     }

)

If result.modifiedCount == 0, throw ConcurrentModificationException (E1)

8. Return Response:

8a. Release distributed lock

8b. Return to client:

```
{
  "success": true,
  "newState": {
    "board": [[0,0,...], [0,1,...], ...],
    "currentPlayer": 2,
    "lastMove": [7, 8],
    "totalMoves": 15
  },
  "version": 42
}
```

9. Opponent Receives Update:

9a. Player B's client polling detects new state

9b. Client animates opponent's move (black piece appears at [7,8])

9c. Client displays: "Your Turn"

9d. Client enables board clicks for Player B

10. Repeat steps 3-9 for Player B's move, alternating turns

11. Game End (Winner or Draw):

11a. System updates MySQL: game_room.status = 3 (FINISHED)

11b. System archives game to MongoDB game_history:

```
GameHistoryDocument.fromGameDocument(
  gameDoc,
  gameNumber: 1,
  endReason: "WIN"
)
```

12. If Ranked Mode → Trigger Match Settlement:

12a. Calculate score changes from score_rule table (+25 winner, -15 loser)

12b. MySQL Transaction BEGIN

12c. INSERT 10 rows into score table (2 players × 5 leaderboards)

12d. UPDATE 10 rows in ranking table (add score_change, exp_value)

12e. MySQL Transaction COMMIT

12f. Redis: DELETE leaderboard:cache: (invalidate all caches)

13. Display Results:

13a. Client shows victory/defeat modal

13b. Client displays score changes (if ranked)

13c. Client offers buttons: "Play Again" / "Leave Room"

The exceptional flow:

E1: Concurrent Move Conflict (Step 7)

Both players attempt to move at exact same time

MongoDB optimistic lock fails for second request (version mismatch)

System returns error: "Game state changed, please retry"

Client auto-retries with new version number

One move succeeds, other is queued for next turn

E2: Invalid Move - Position Occupied (Step 4e)

Validation chain detects board[7][8] != 0

System returns error: { success: false, error: "Position already occupied" }

Client displays error toast

Player can make another move

E3: Invalid Move - Wrong Turn (Step 4e)

Validation chain detects currentPlayer != playerColor

System returns error: "Not your turn"

Client displays error toast

Wait for opponent's move

E4: Move Timeout (Step 4e)

Validation chain detects move time exceeded (e.g., 30 seconds)

System automatically forfeits player's turn

System switches turn to opponent

Client displays: "Timeout! Turn skipped."

E5: Player Disconnects During Game (Any step)

Client stops polling for > 30 seconds

System detects disconnect via timeout

System marks game as abandoned

Opponent sees: "Opponent disconnected. You win by forfeit."

If ranked: Award full points to remaining player

E6: Network Error During Move Submission (Step 3b)

HTTP request times out or fails

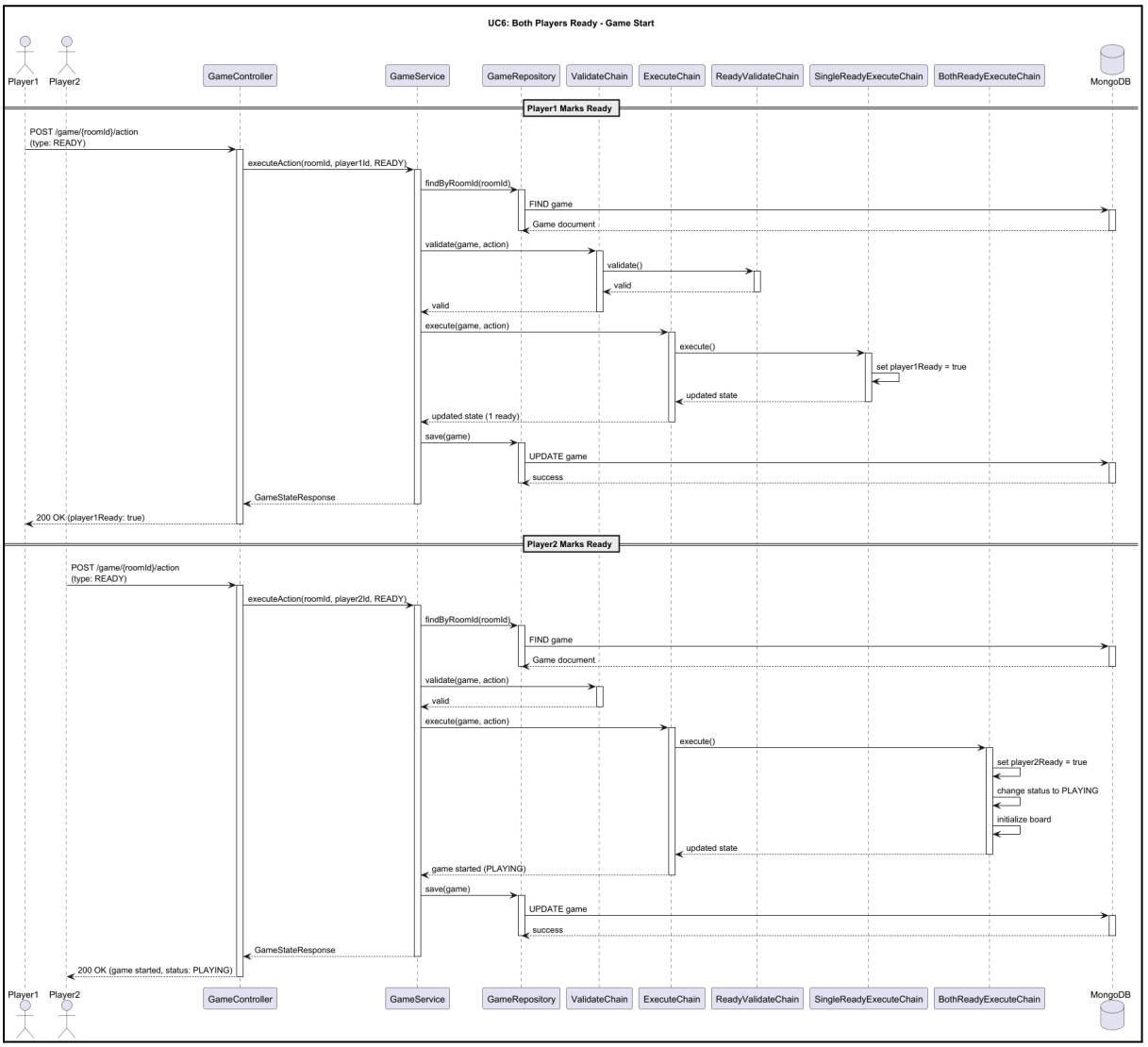
Client displays: "Network error. Retrying..."

Client auto-retries up to 3 times with exponential backoff

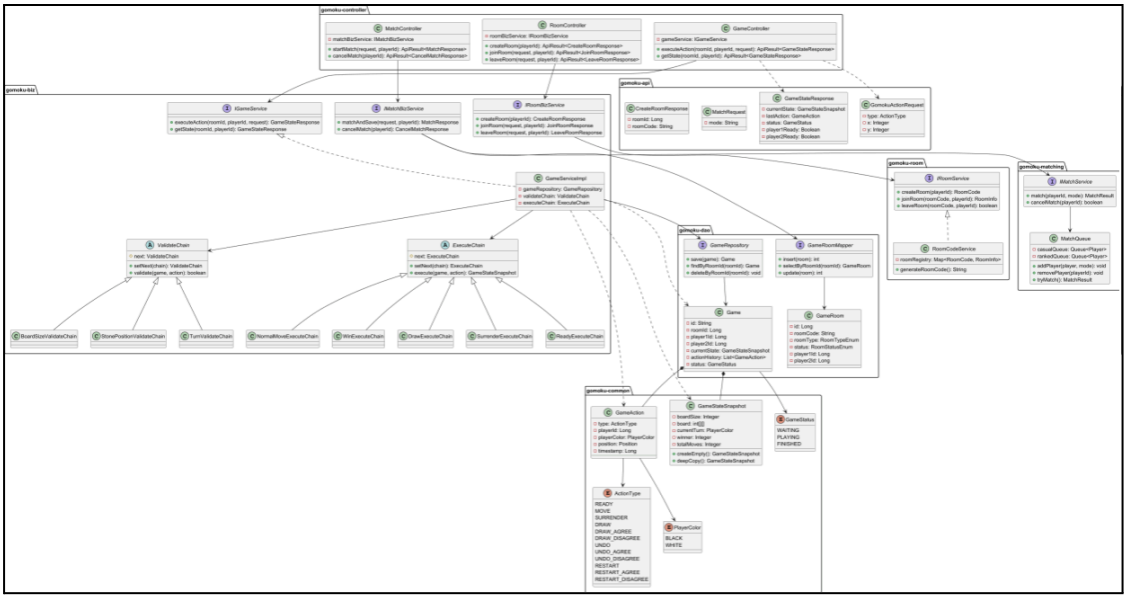
If all retries fail, display: "Connection lost. Please refresh."

For the playgame Use Case, we have the following diagrams:

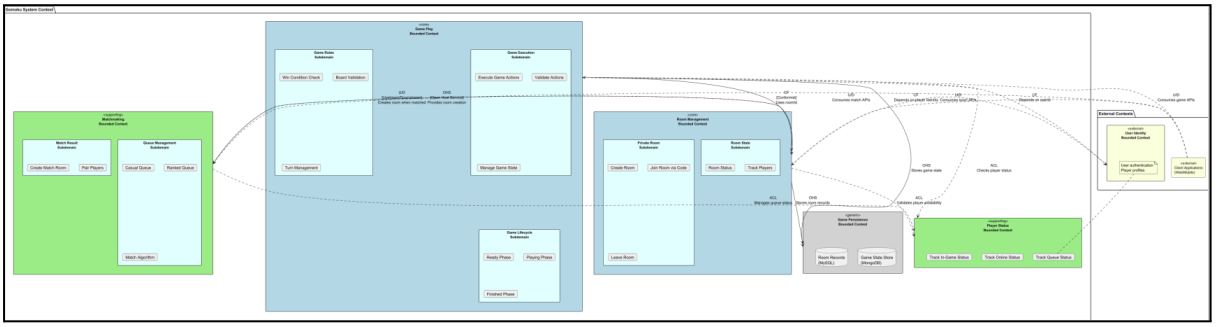
Sequence diagram:


UC6: Both Players Ready - Game Start
Player1
Player2
GameController
GameService
GameRepository
ValidateChain
ExecuteChain
ReadyValidateChain
SingleReadyExecuteChain
BothReadyExecuteChain
MongoDB
Player1 Marks Ready
POST /game/{roomId}/action
(type: READY)
executeAction(roomId, player1Id, READY)
findByRoomId(roomId)
FIND game
Game document
validate(game, action)
validate()
valid
valid
execute(game, action)
execute()
set player1Ready = true
updated state
updated state (1 ready)
save(game)
UPDATE game
success
GameStateResponse
200 OK (player1Ready: true)
Player2 Marks Ready
POST /game/{roomId}/action
(type: READY)
executeAction(roomId, player2Id, READY)
findByRoomId(roomId)
FIND game
Game document
validate(game, action)
valid
execute(game, action)
execute()
set player2Ready = true
change status to PLAYING
initialize board
updated state
game started (PLAYING)
save(game)
UPDATE game
success
GameStateResponse
200 OK (game started, status: PLAYING)

Class diagram:



DDD:



3.3.2.4 UC4

UC4:Player plays against an AI opponent with configurable difficulty (K-value), where the AI engine computes optimal moves using heuristic evaluation or GPT-4o-mini integration.

The normal flow for the UC4:

1. AI Room Creation:

   1a. Player clicks "Practice with AI" button

   1b. System displays AI configuration modal:

   Difficulty: Easy (K=5) / Medium (K=10) / Hard (K=15)

   Board size: 15×15

   Player color: Black (first) / White (second) / Random

   1c. Player selects "Medium, Black" and clicks "Start"

2. Room Setup:

   2a. System creates room in MySQL: type=0 (CASUAL), player1_id=12345, player2_id=NULL

   2b. System creates game document in MongoDB with whitePlayerId=0 (AI marker)

   2c. System sets game status to PLAYING (skip ready phase)

   2d. Client redirects to Game Room page

3. Player's Turn (Black):

   3a. Player makes move at position [7, 7]

   3b. System validates and updates game state (same as UC3 steps 4-7)

   3c. System checks win condition → No winner yet

   3d. Client displays: "AI is thinking..."

4. AI Computation:

   4a. System sends request to AI Engine:

   POST /api/gomoku/ai/move

   Body: {

   "roomId": 98765,

   "boardState": [[0,0,...], [1,0,...], ...],

   "difficulty": 10,

   "lastMove": [7, 7]

   }

   4b. AI Engine evaluates board:

Method 1 (Heuristic): Scan all empty positions, score based on:

Offensive potential: Count open 2-in-a-row, 3-in-a-row, 4-in-a-row

Defensive necessity: Block opponent's threats

Position value: Center positions scored higher

Method 2 (GPT-4o-mini): Send board to OpenAI API:

```
{
  "model": "gpt-4o-mini",
  "messages": [{
    "role": "system",
    "content": "You are a Gomoku AI. Analyze this board and suggest the best move..."
  }],
  "temperature": 0.7
}
```

4c. AI Engine returns move: { "position": [7, 8], "confidence": 0.85 }

5. AI Move Execution:

   5a. System validates AI move (same validation chains)

   5b. System updates game state: board[7][8] = 2 (white/AI)

   5c. System appends to action history

   5d. System checks win condition → No winner yet

   5e. System switches turn back to player

6. Repeat steps 3-5 until game ends
7. Game End:

   7a. System detects five-in-a-row (either player or AI)

   7b. System sets winner and status to FINISHED

   7c. System archives game to game_history with endReason="WIN"

8. Experience Award:

   8a. System queries level_rule table: mode_type='CASUAL', match_result='WIN'

   8b. If player won: +20 exp

   8c. If player lost: +5 exp

   8d. System updates player's total_exp in ranking table (for TOTAL leaderboard)

   8e. System checks if level up threshold crossed

9. Display Results:

9a. Client shows modal: "Victory! +20 EXP" or "Defeat! +5 EXP"

9b. If level up: Display "Level Up! You are now Level 4"

9c. Offer buttons: "Play Again" / "Back to Lobby"

The exceptional flow:

E1: AI Engine Timeout (Step 4b)

OpenAI API takes > 10 seconds to respond

System falls back to heuristic algorithm:

Defensive mode: Block any opponent 3-in-a-row or 4-in-a-row

Random move: If no immediate threats, pick random valid position

Client displays: "AI is taking a while..."

Continue with fallback move

E2: AI Engine Error (Step 4b)

OpenAI API returns error (rate limit, invalid request, etc.)

System logs error for monitoring

System falls back to heuristic algorithm

Client does not notify player (seamless fallback)

E3: AI Selects Invalid Move (Step 5a)

Rare bug: AI returns position already occupied or out of bounds

Validation chain catches error

System retries AI computation with additional constraint: "Previous move invalid, choose different position"

If retry fails, fallback to random valid position

E4: Player Quits Mid-Game (Any step)

Player clicks "Leave Room" or closes browser

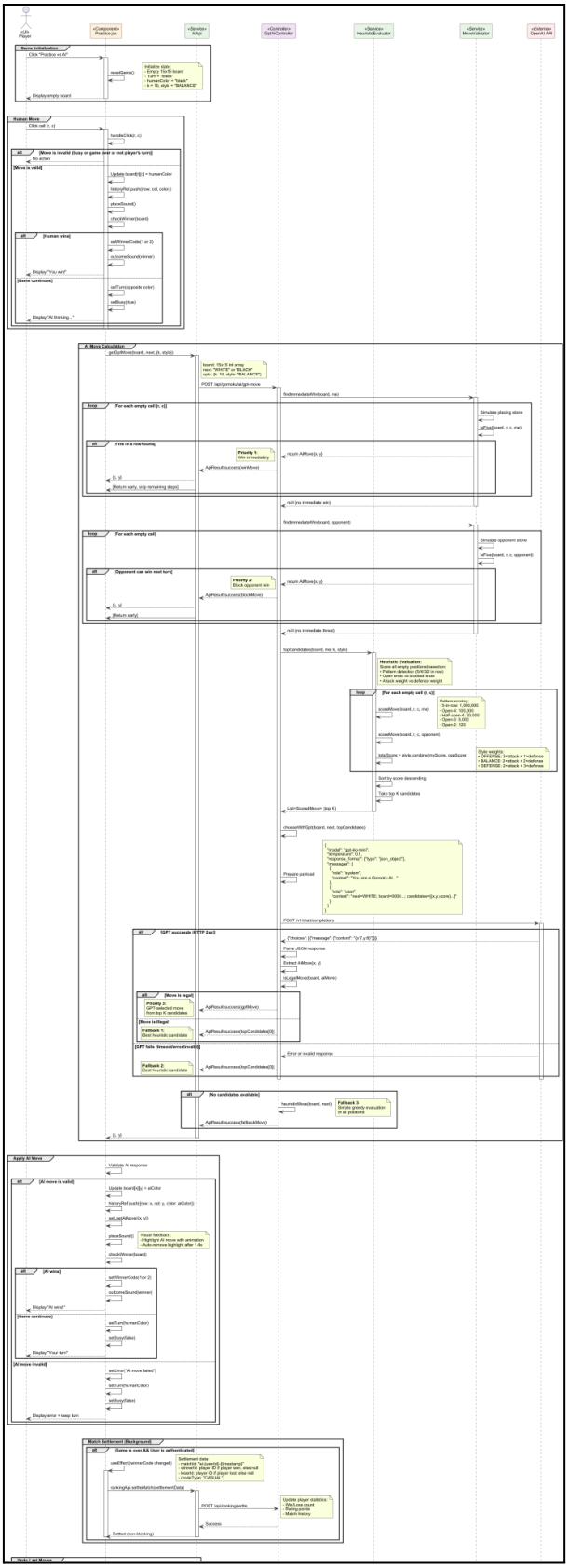
System marks game as abandoned (not recorded in history)

No experience points awarded

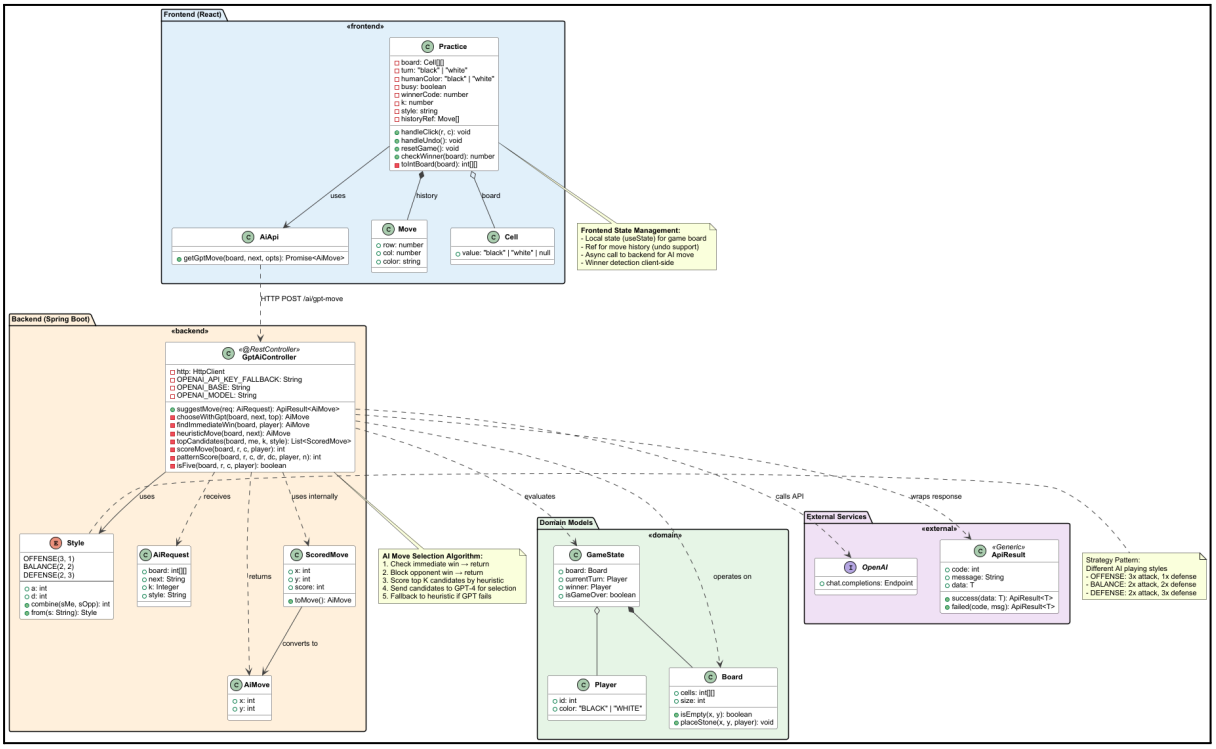
Room deleted from MySQL and MongoDB

For the practice with AI Use Case, we have the following diagrams:

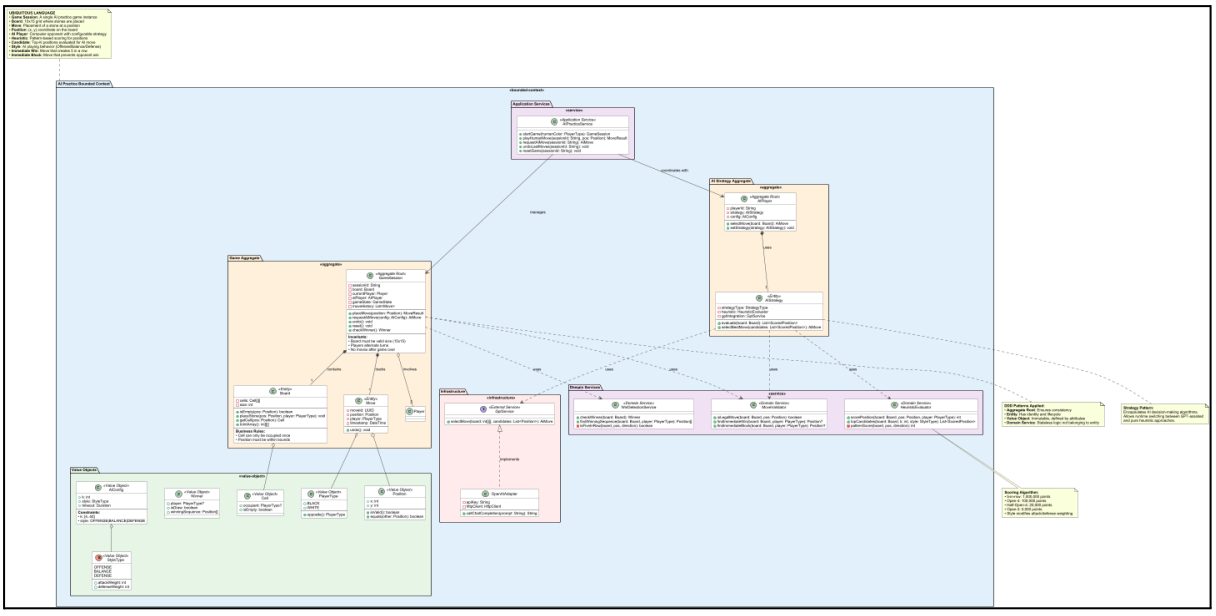
Sequence diagram:



Class diagram:



DDD:



3.3.2.5 UC5

UC5: Scheduled background job recalculates player rankings across multiple leaderboard types (daily, weekly, monthly, seasonal, total) by aggregating scores and updating rank positions.

The normal flow for the UC5:

1. Trigger (Every 5 minutes):

   1a. System cron job triggers at :00, :05, :10, :15, ... of each hour

   1b. System queries active leaderboard rules:

   SELECT id, type FROM leaderboard_rule

   WHERE UNIX_TIMESTAMP(NOW()) BETWEEN start_time AND end_time

   -- Returns: [1 (TOTAL), 2 (MONTHLY), 3 (SEASONAL), 10 (WEEKLY), 113 (DAILY)]

2. For Each Leaderboard (Parallel Execution):

   2a. Process TOTAL leaderboard (ID=1)

   2b. Process MONTHLY leaderboard (ID=2)

   2c. Process SEASONAL leaderboard (ID=3)

   2d. Process WEEKLY leaderboard (ID=10)

   2e. Process DAILY leaderboard (ID=113)

3. Recalculate Ranks (Example: TOTAL leaderboard):

   UPDATE ranking r

   JOIN (

   SELECT user_id,

   ROW_NUMBER() OVER (ORDER BY current_total_score DESC) as new_rank

   FROM ranking

   WHERE leaderboard_rule_id = 1

) ranked ON r.user_id = ranked.user_id

   SET r.rank_position = ranked.new_rank,

   r.updated_time = NOW()

   WHERE r.leaderboard_rule_id = 1;

   This updates rank_position for all players (1st, 2nd, 3rd, ...)

4. Verify Updates:

4a. System counts updated rows

4b. System logs: "Updated 1,523 player ranks for TOTAL leaderboard"

5. Populate Redis Cache:

   5a. System queries top 1000 players from MySQL:

   SELECT user_id, current_total_score, rank_position

   FROM ranking

   WHERE leaderboard_rule_id = 1

   ORDER BY current_total_score DESC

   LIMIT 1000

   5b. System clears old cache: DEL leaderboard:cache:1

   5c. System populates Redis Sorted Set:

   ZADD leaderboard:cache:1

   1550 "userId:12345"

   1525 "userId:67890"

   1510 "userId:11223"

   …

   EXPIRE leaderboard:cache:1 300 // 5 minutes

6. Repeat steps 3-5 for other leaderboards (MONTHLY, SEASONAL, WEEKLY, DAILY)
7. Generate Future Leaderboards (If needed):

   7a. Check if tomorrow's DAILY leaderboard exists:

   SELECT id FROM leaderboard_rule

   WHERE type = 'DAILY'

   AND DATE(FROM_UNIXTIME(start_time)) = DATE_ADD(CURDATE(), INTERVAL 1 DAY)

   7b. If not exists, INSERT new leaderboard_rule for tomorrow

   7c. Repeat for next 30 days (maintain rolling 30-day window)

8. Cleanup Expired Leaderboards:

   8a. Query leaderboards older than 1 year:

   SELECT id FROM leaderboard_rule

   WHERE type IN ('DAILY', 'WEEKLY')

   AND end_time < UNIX_TIMESTAMP(DATE_SUB(NOW(), INTERVAL 1 YEAR))

   8b. For each expired leaderboard:

Archive ranking rows to cold storage (optional)

DELETE from ranking table

DELETE from leaderboard_rule table

9. Log Completion:

9a. System logs: "Leaderboard calculation completed in 3.2 seconds"

9b. System updates Prometheus metrics: leaderboard_calculation_duration_seconds

The exceptional flow:

E1: Database Lock Timeout (Step 3)

UPDATE query takes > 30 seconds due to high concurrent load

MySQL throws lock timeout error

System logs error: "Leaderboard calculation failed due to lock timeout"

System retries after 1 minute

If retry succeeds, continue

If retry fails again, alert monitoring system

E2: Redis Connection Error (Step 5b)

Redis is unavailable or overloaded

System logs error: "Failed to populate Redis cache for leaderboard 1"

System continues with other leaderboards

Leaderboard API falls back to direct MySQL queries (slower but functional)

Alert monitoring system

E3: No Ranked Matches Since Last Run (Step 1b)

Query returns 0 updated records in all leaderboards

System logs: "No new ranked matches, skipping calculation"

System still refreshes Redis caches with existing data

Use case ends early

E4: Concurrent Calculation Detected (Step 1a)

Another cron job instance is still running from previous trigger

System detects distributed lock: redis.exists("leaderboard:calculation:lock")

System logs: "Leaderboard calculation already in progress, skipping"

Use case ends without action

E5: MySQL Replication Lag (Step 3)

Slave database has replication lag > 10 seconds

System detects via SHOW SLAVE STATUS

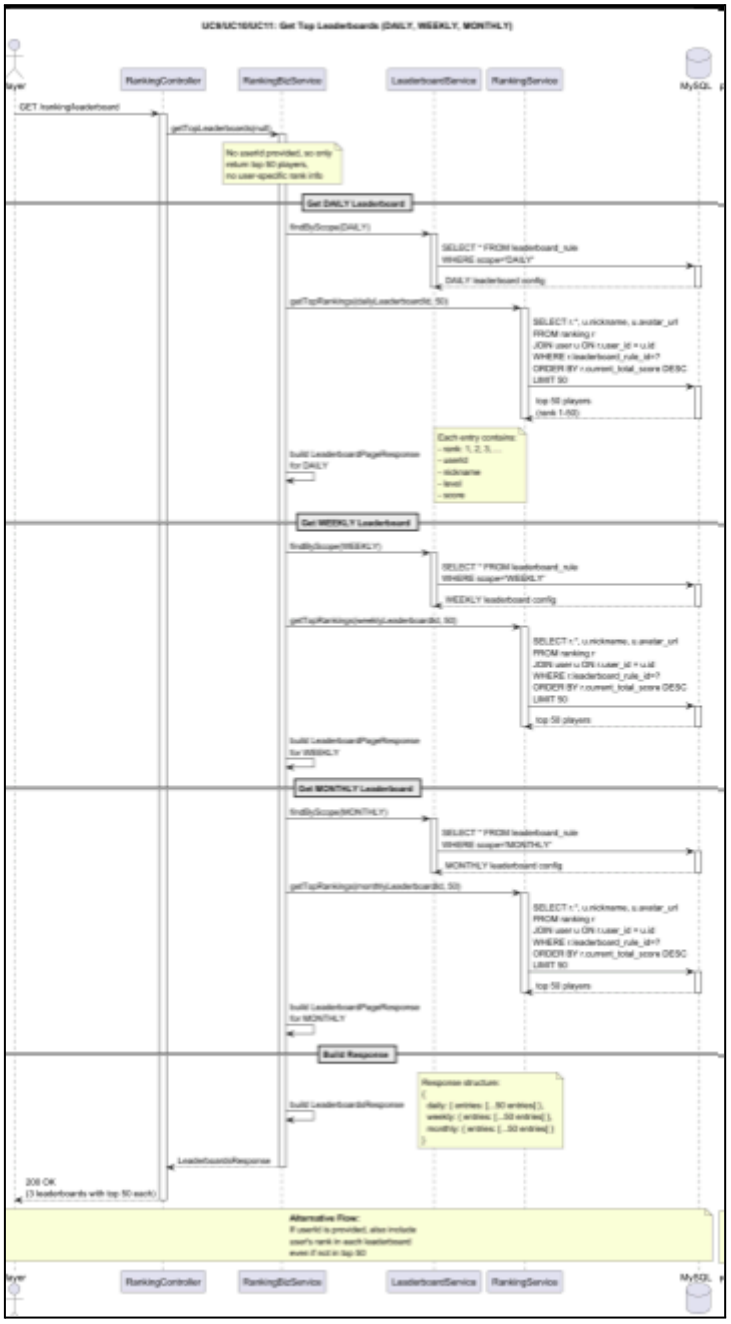
System logs warning: "Replication lag detected (15s), calculation may use stale data"

System proceeds with calculation (eventual consistency acceptable)

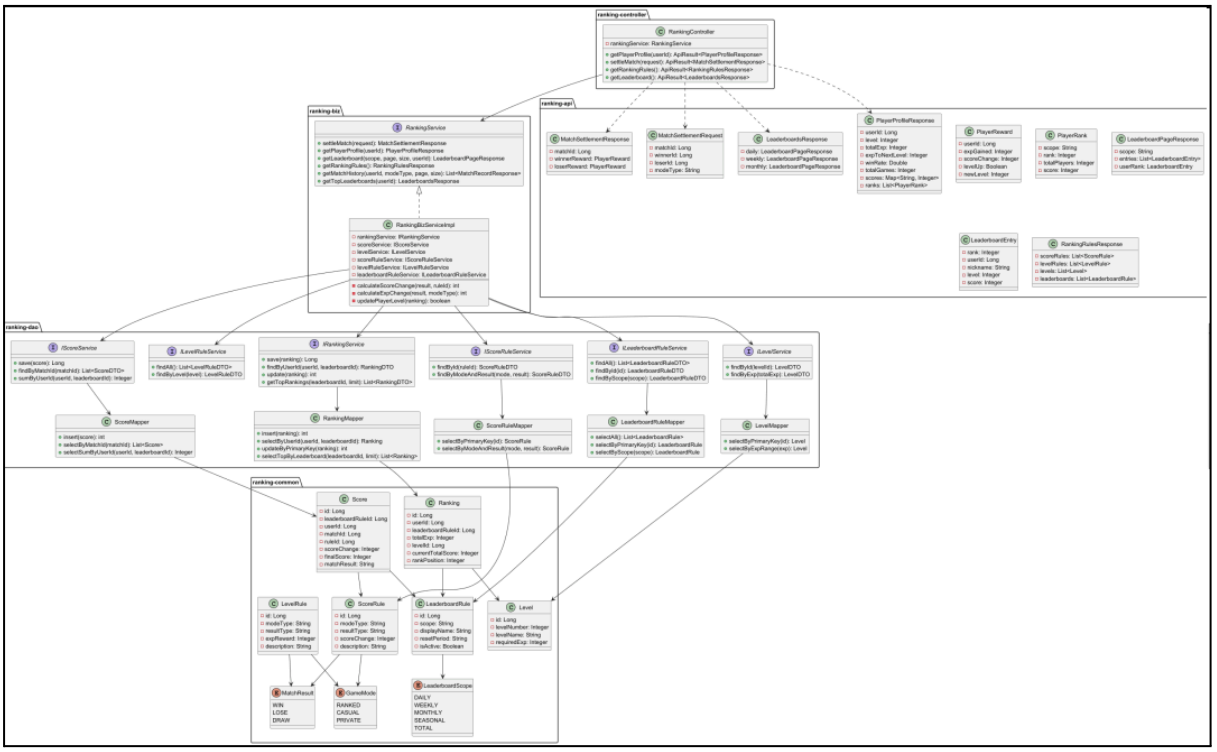
Alert monitoring system if lag exceeds threshold

For the viewLeaderBoard Use Case, we have the following diagrams:

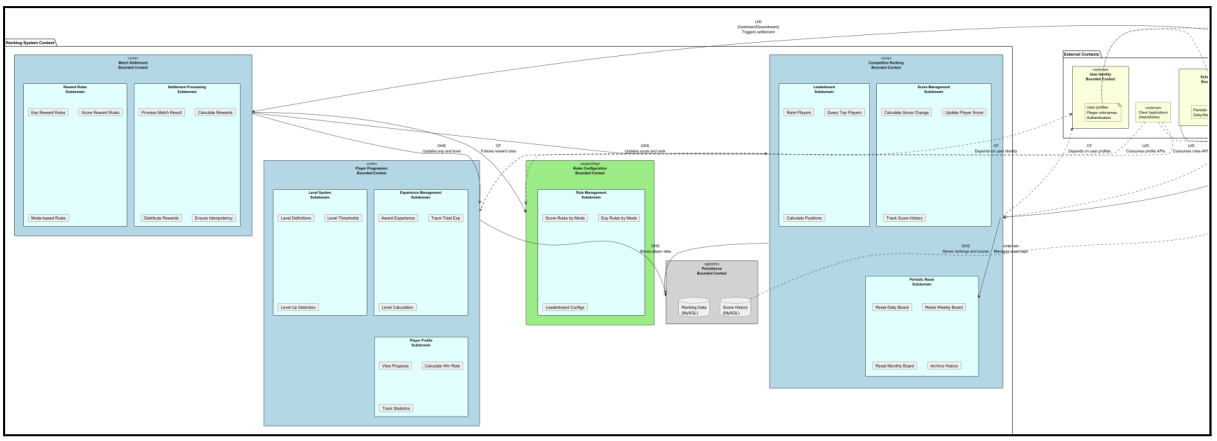
Sequence diagram:



Class diagram:



DDD:

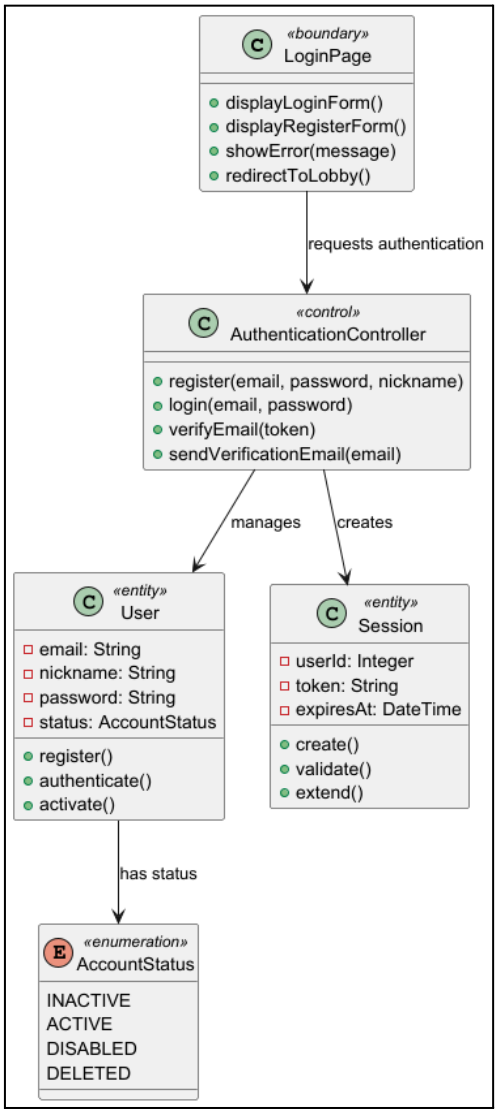


3.4 Analysis & Design Models

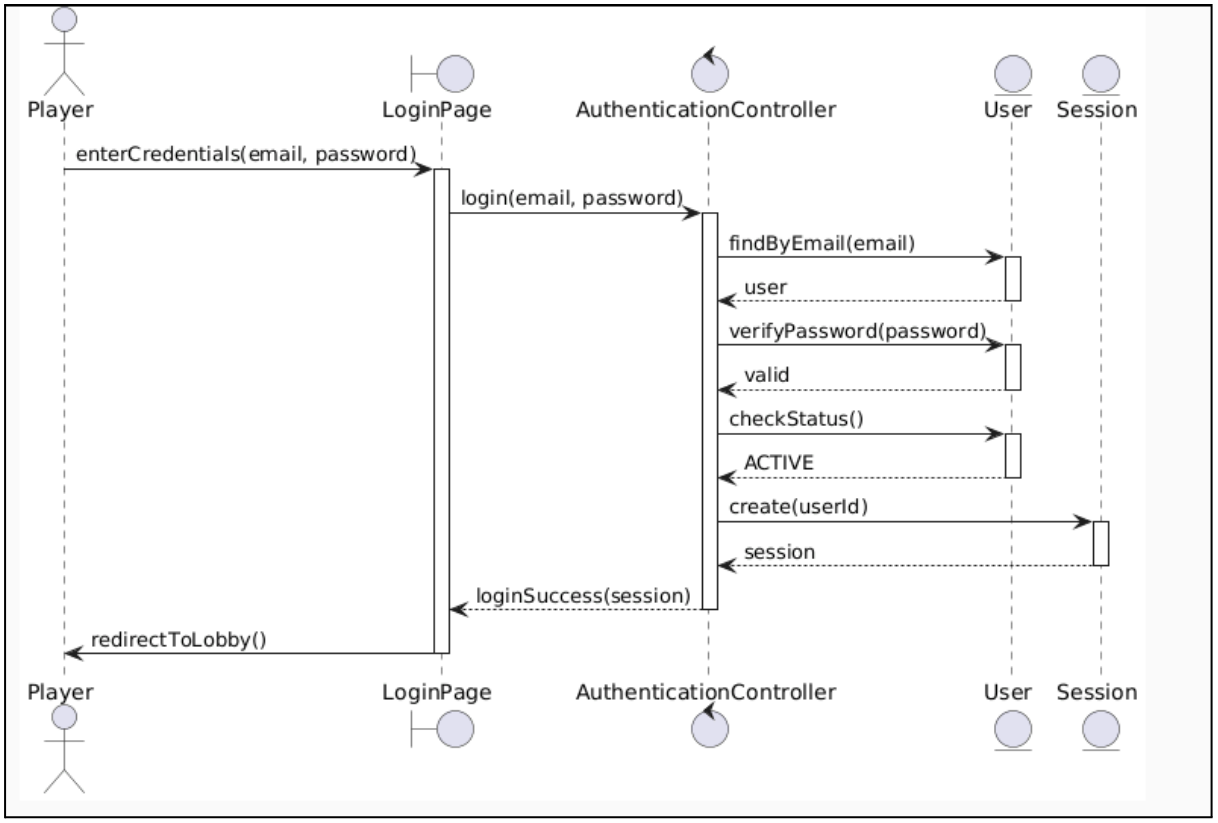
3.4.1 Use Case 1 by Hao Tian

For the analysis model:

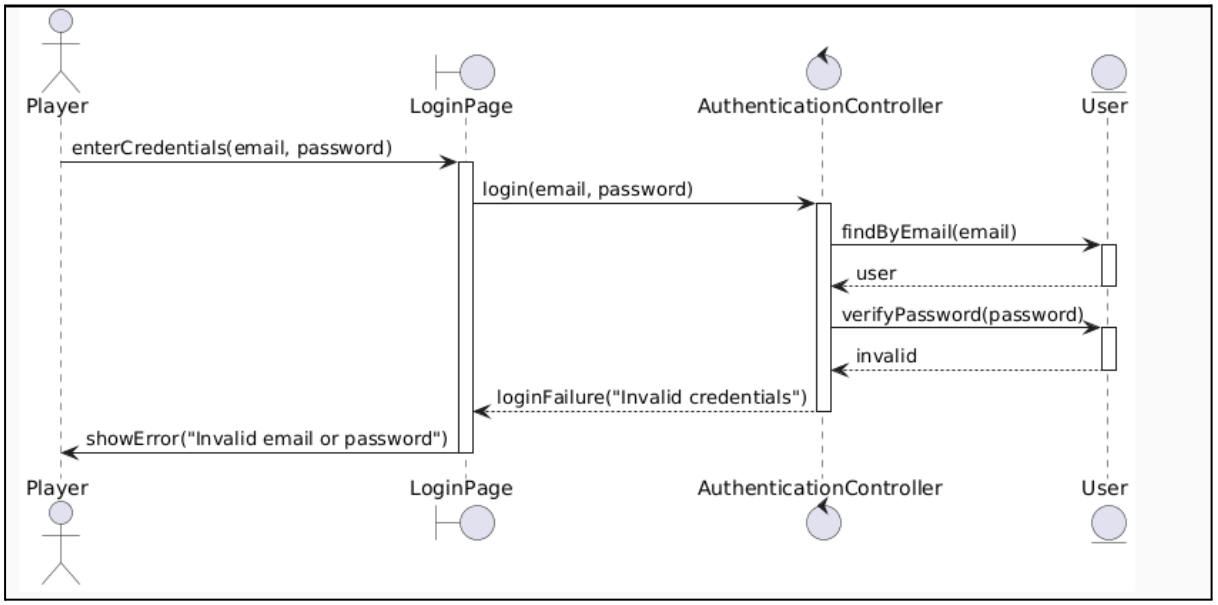
Class diagram:



Sequence diagram - normal flow:

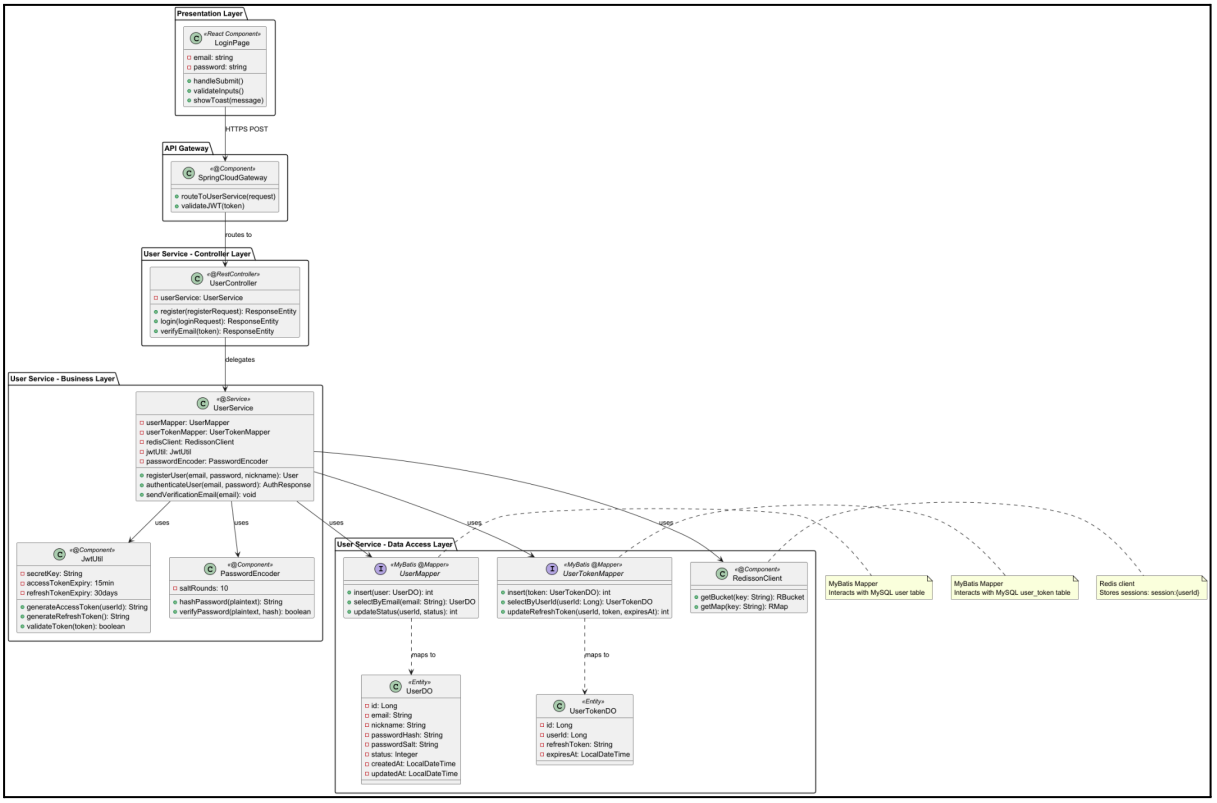


Sequence diagram - exceptional flow:

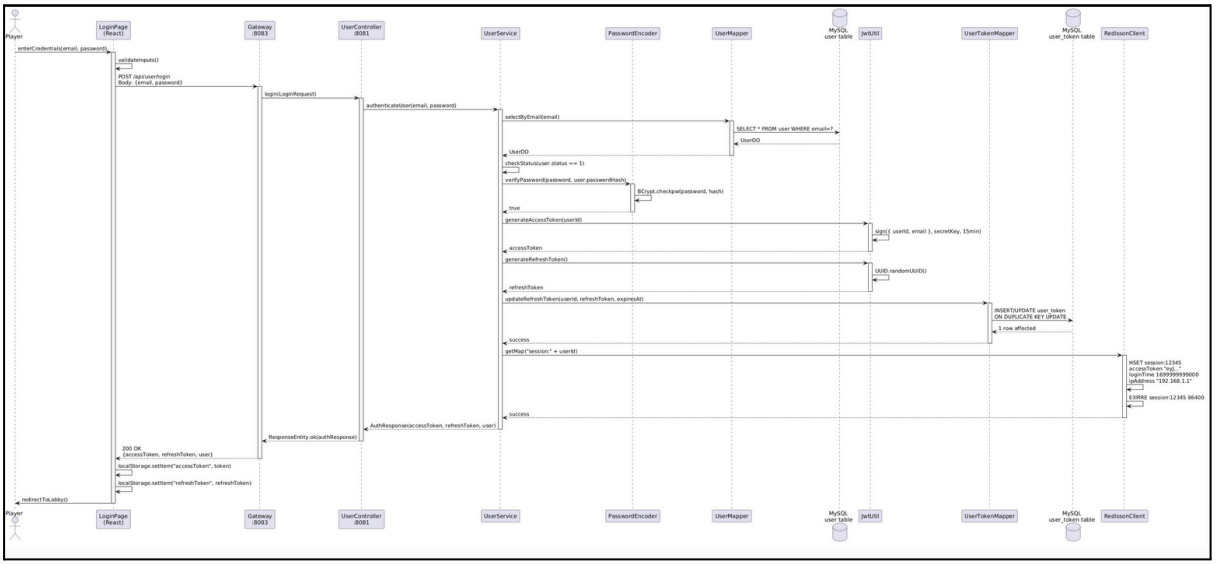


For the design model:

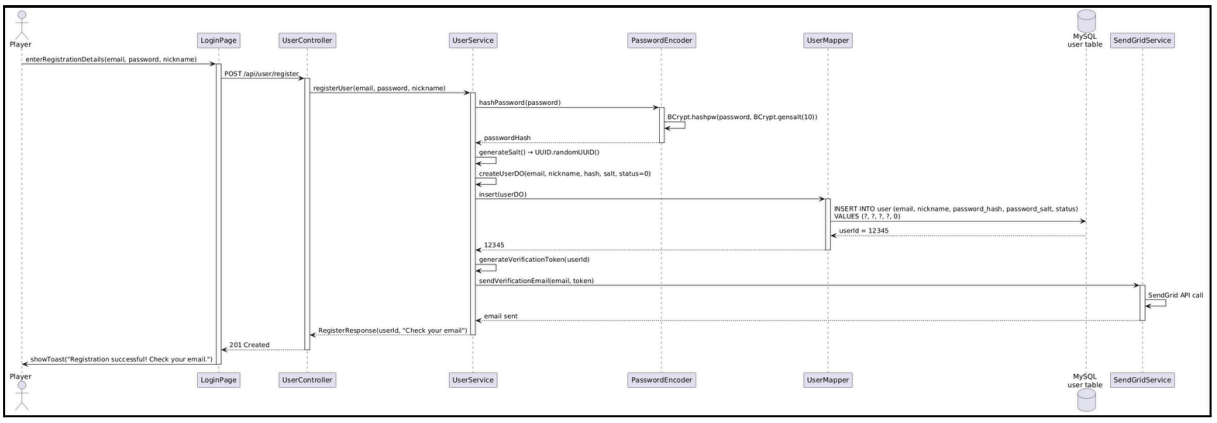
Class diagram:



Sequence diagram - login flow:



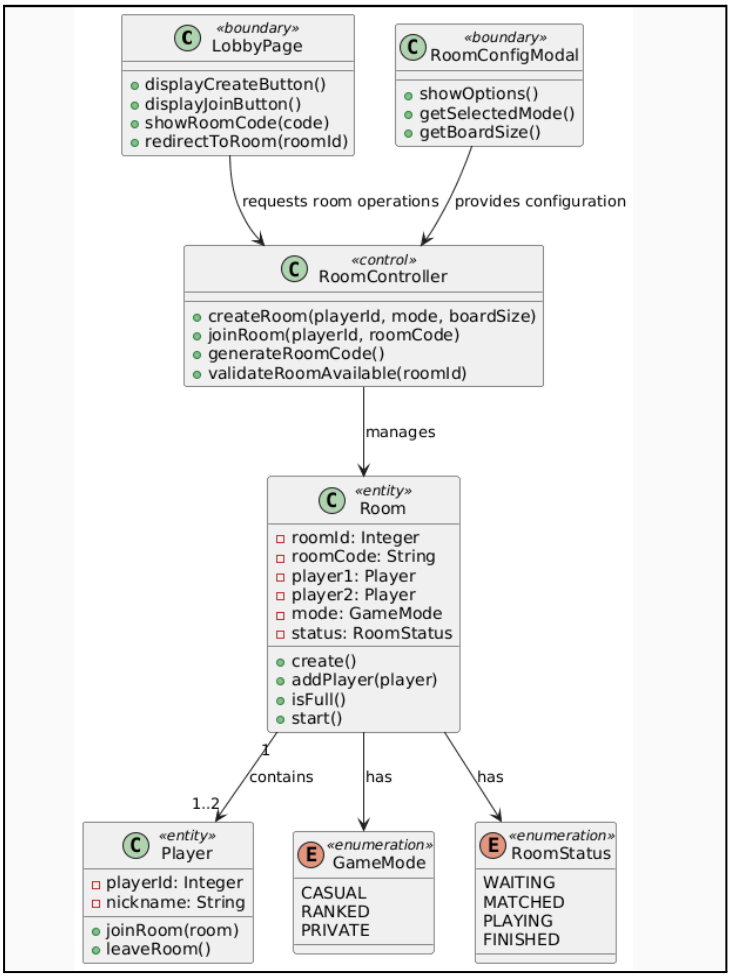
Sequence diagram - registration flow:



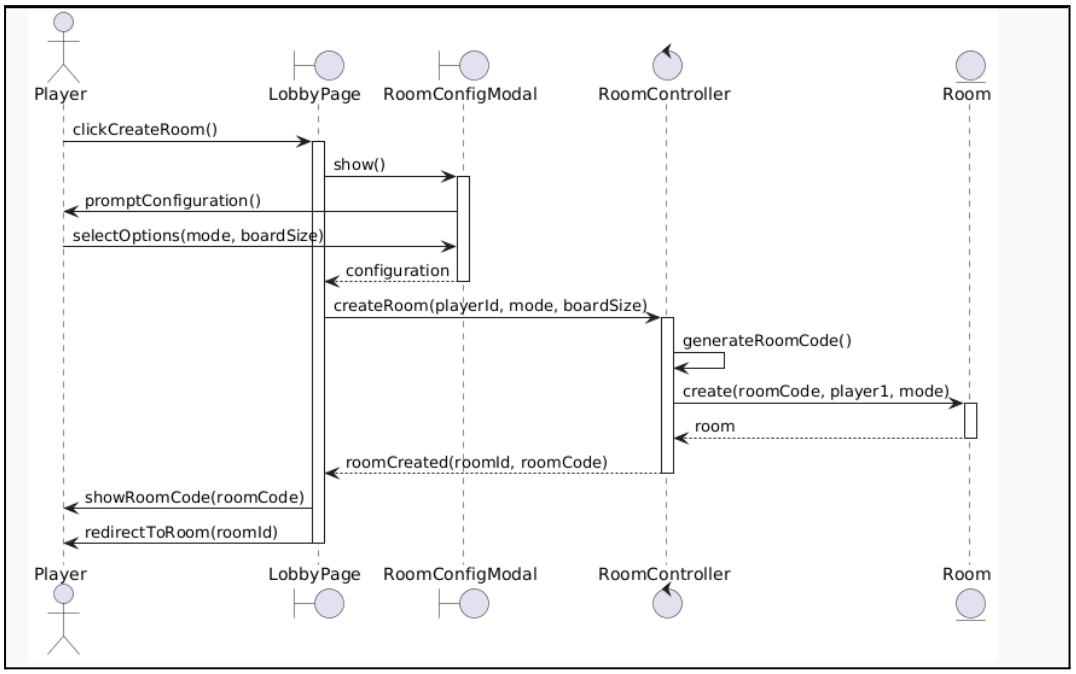
3.4.2 Use Case 2 by Li YuanXing

For the analysis model:

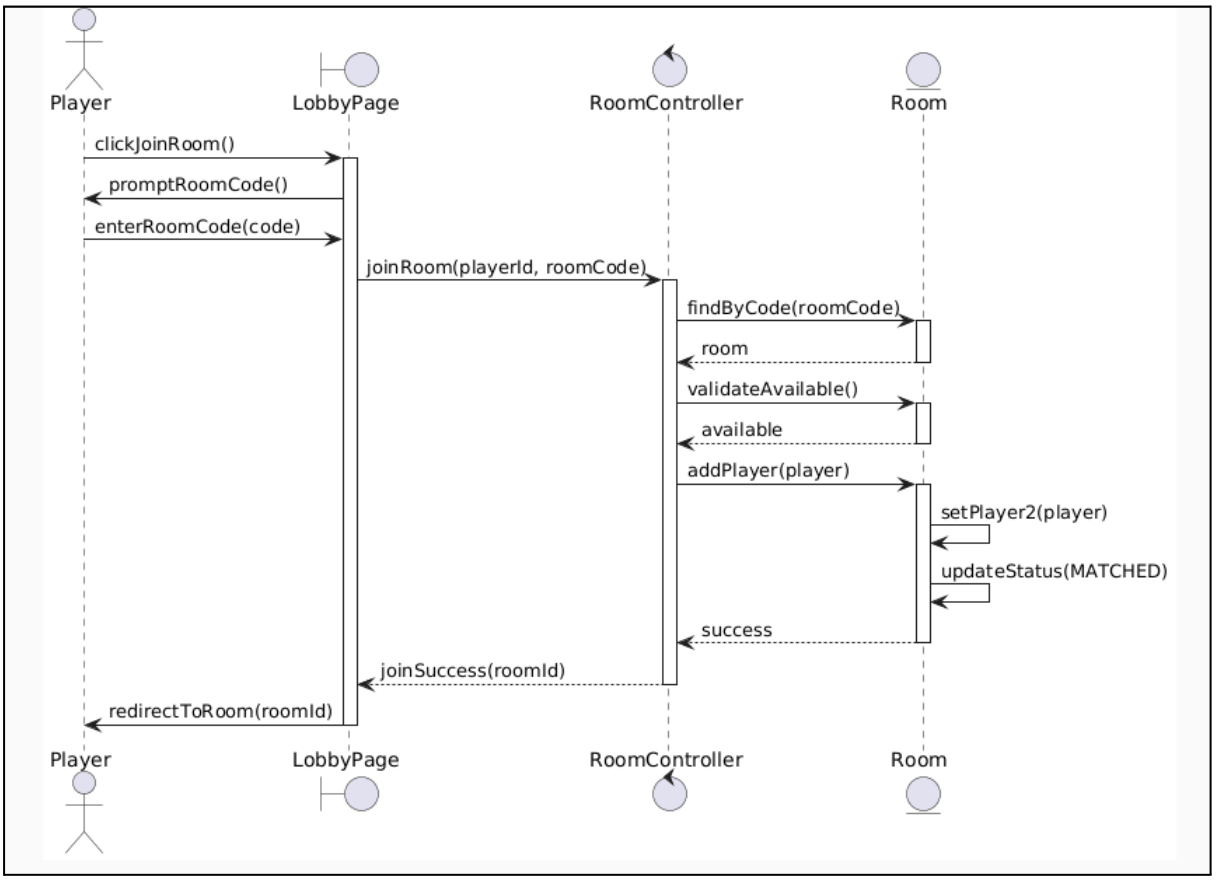
Class diagram:



Sequence diagram - create room:

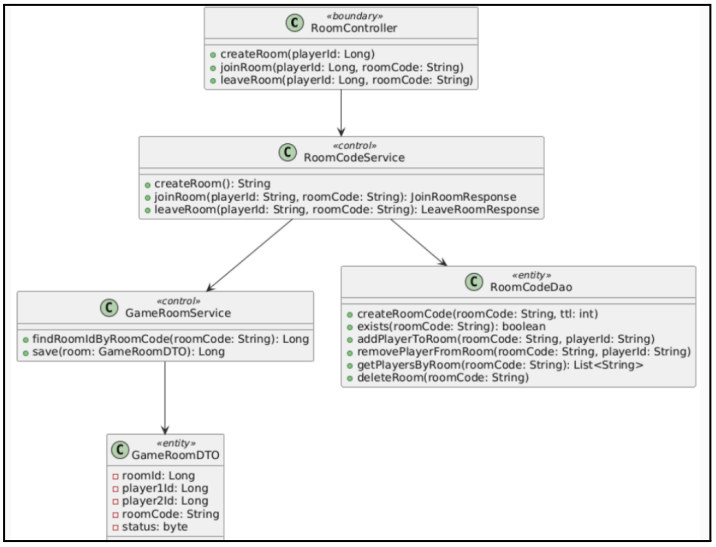


Sequence diagram - join room:

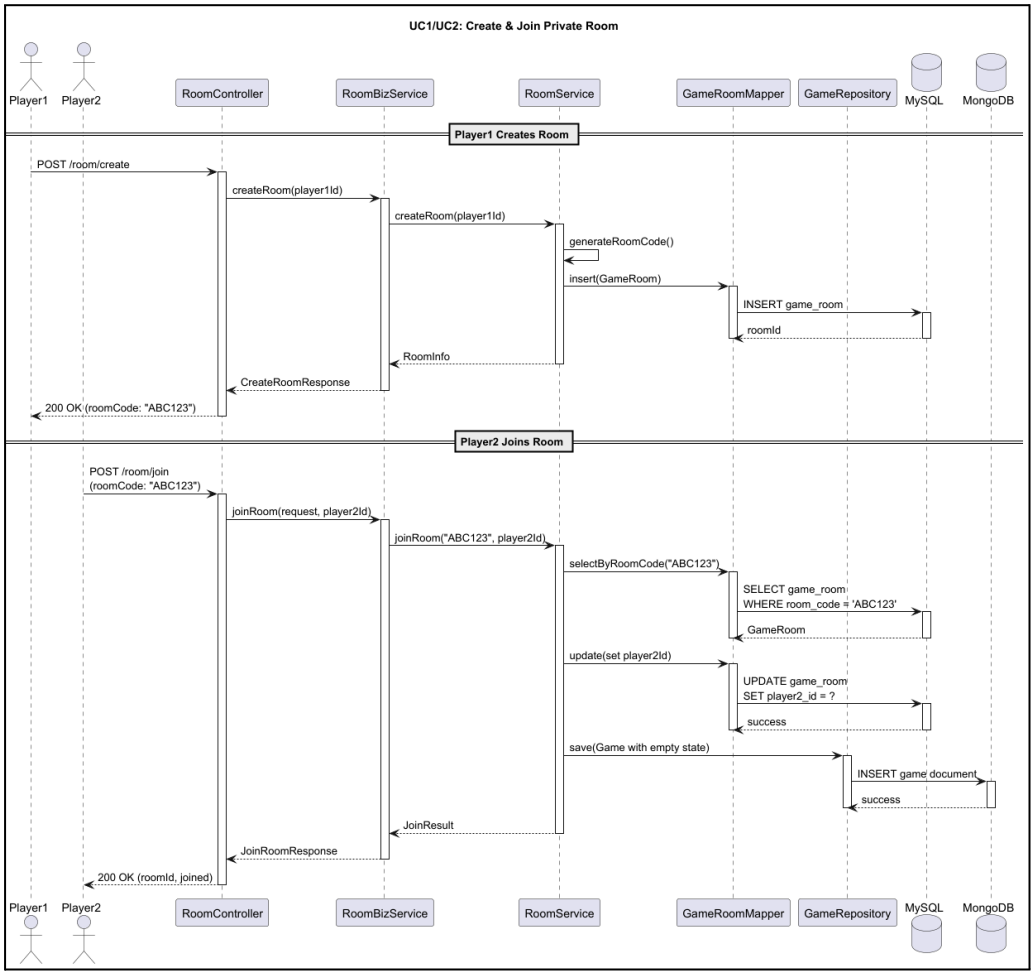


For the design model:

Class diagram:



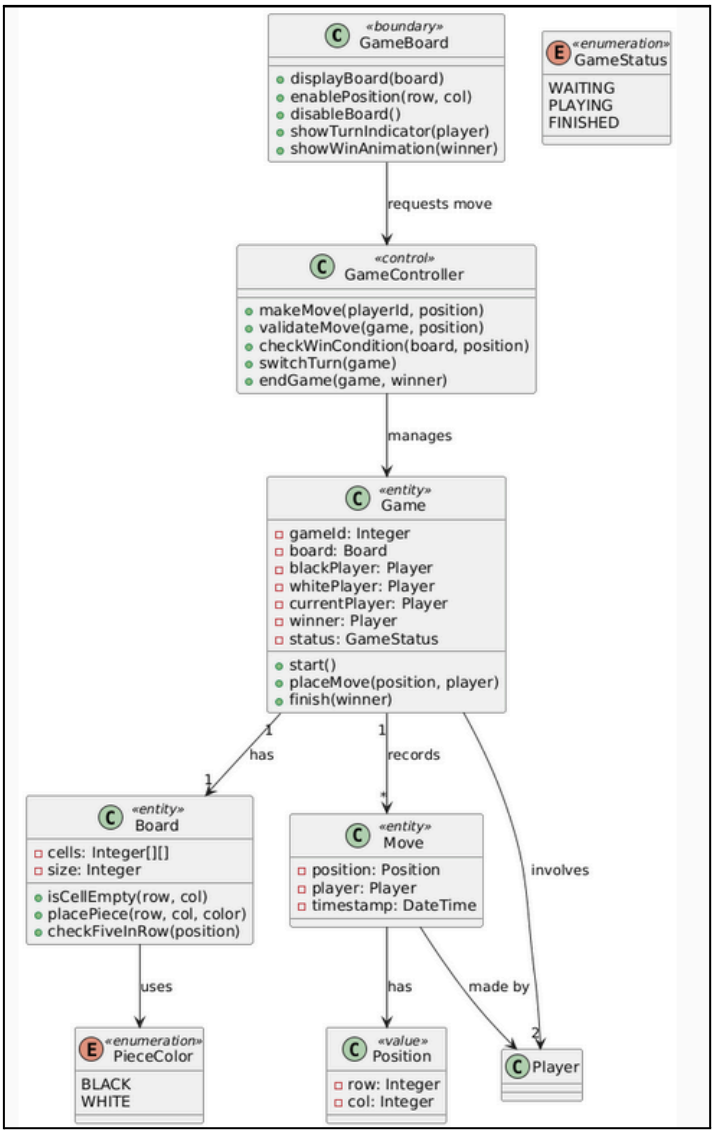
Sequence diagram:



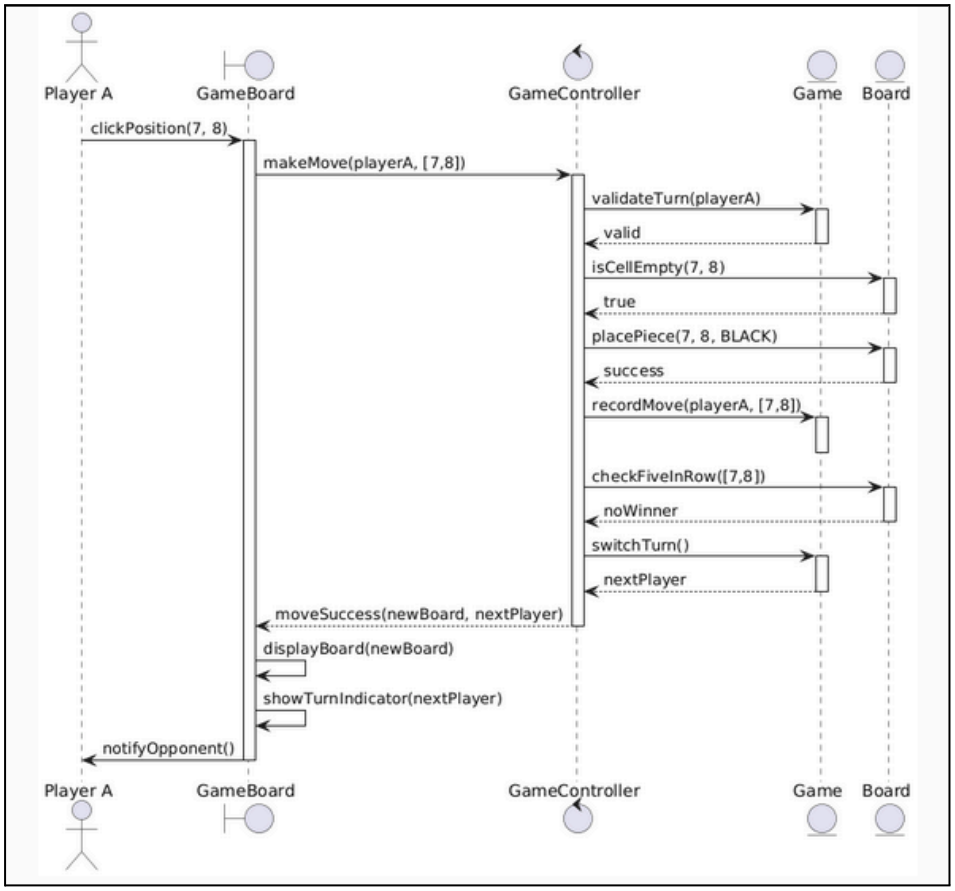
3.4.3 Use Case 3 by Hao Tian

For analysis model:

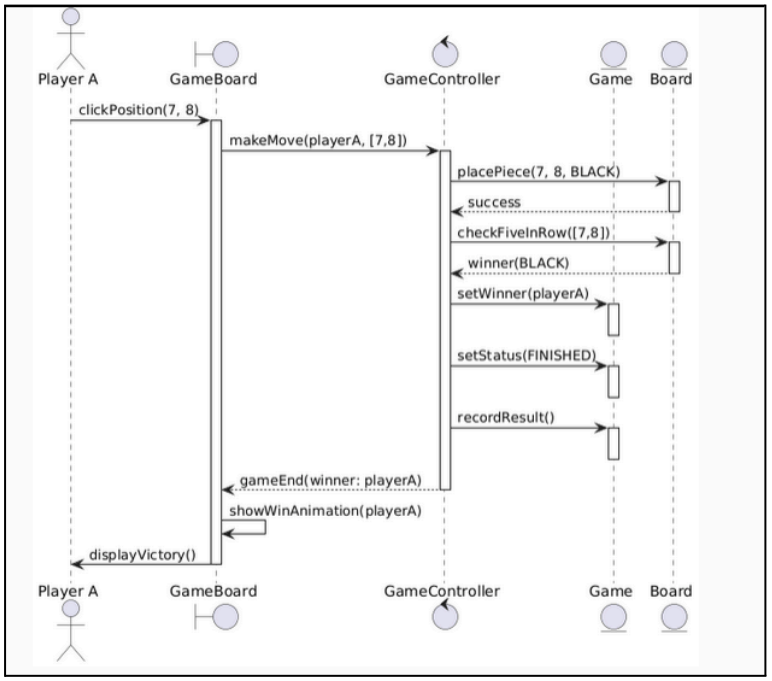
Class diagram:



Sequence diagram - Normal Move:

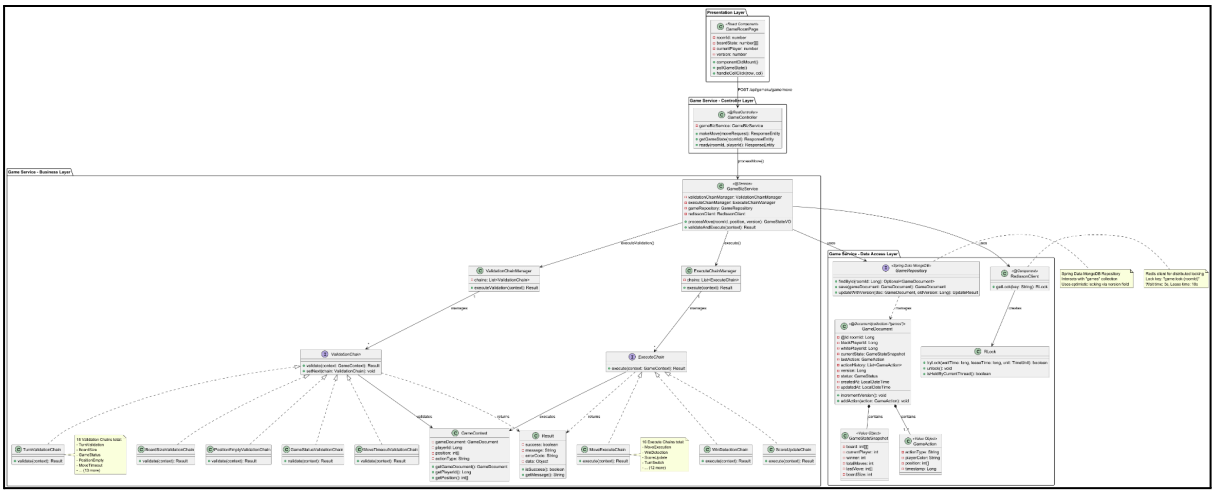


Sequence diagram - Winning Move

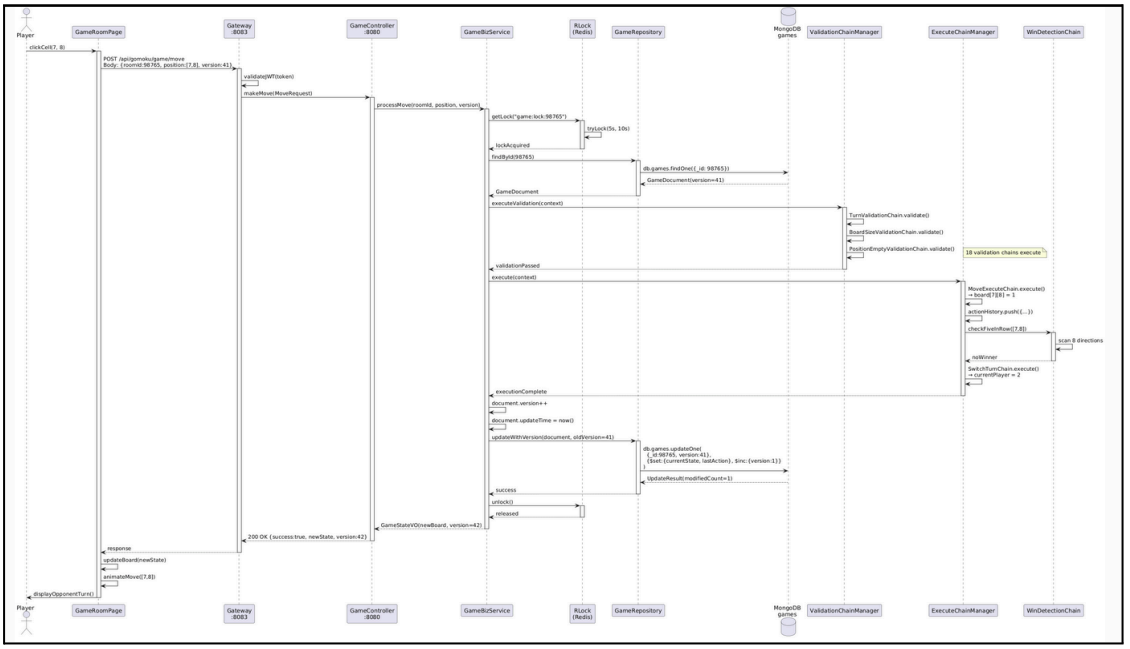


For design model:

Class diagram:



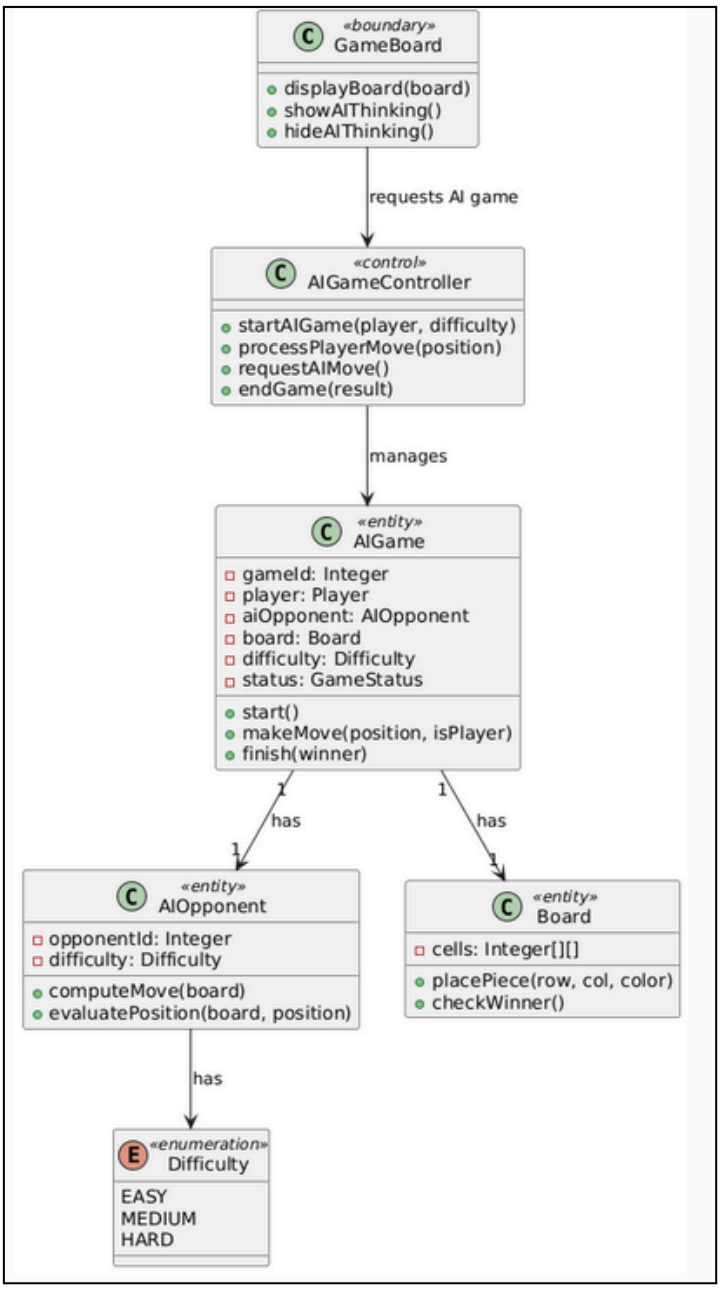
Sequence diagram - Make Move:



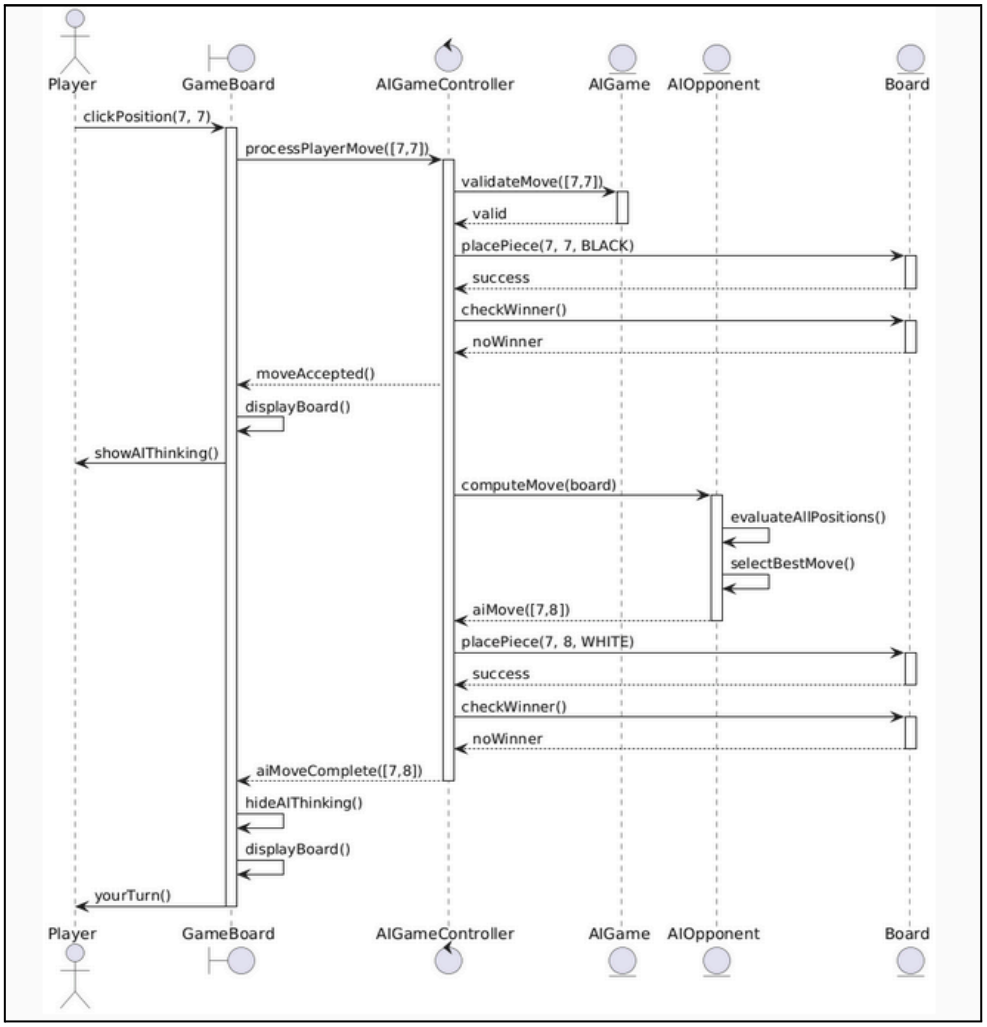
3.4.4 Use Case 4 by Shashank Bagda

For the analysis model:

Class diagram:

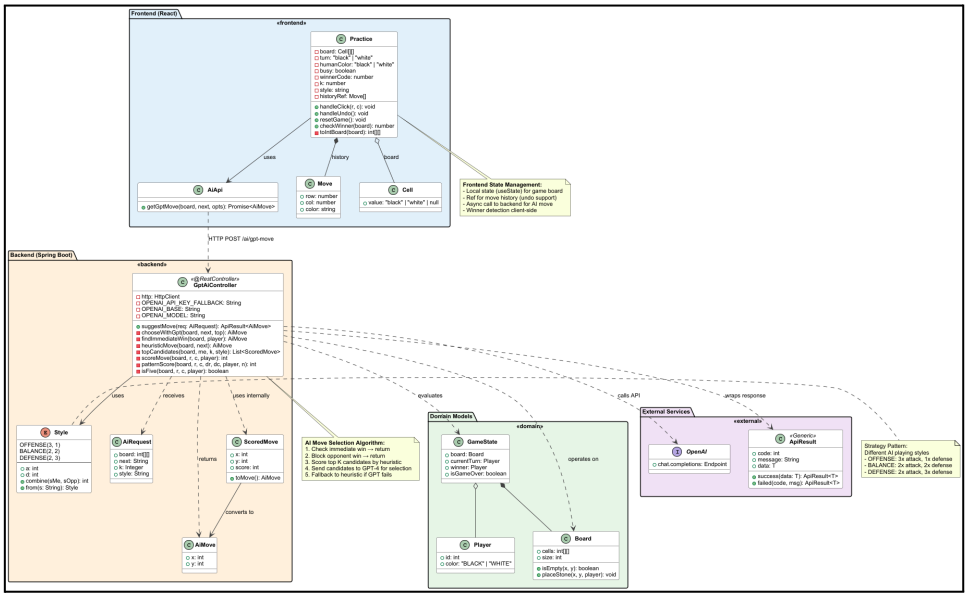


Sequence diagram - Player Move then AI Move:

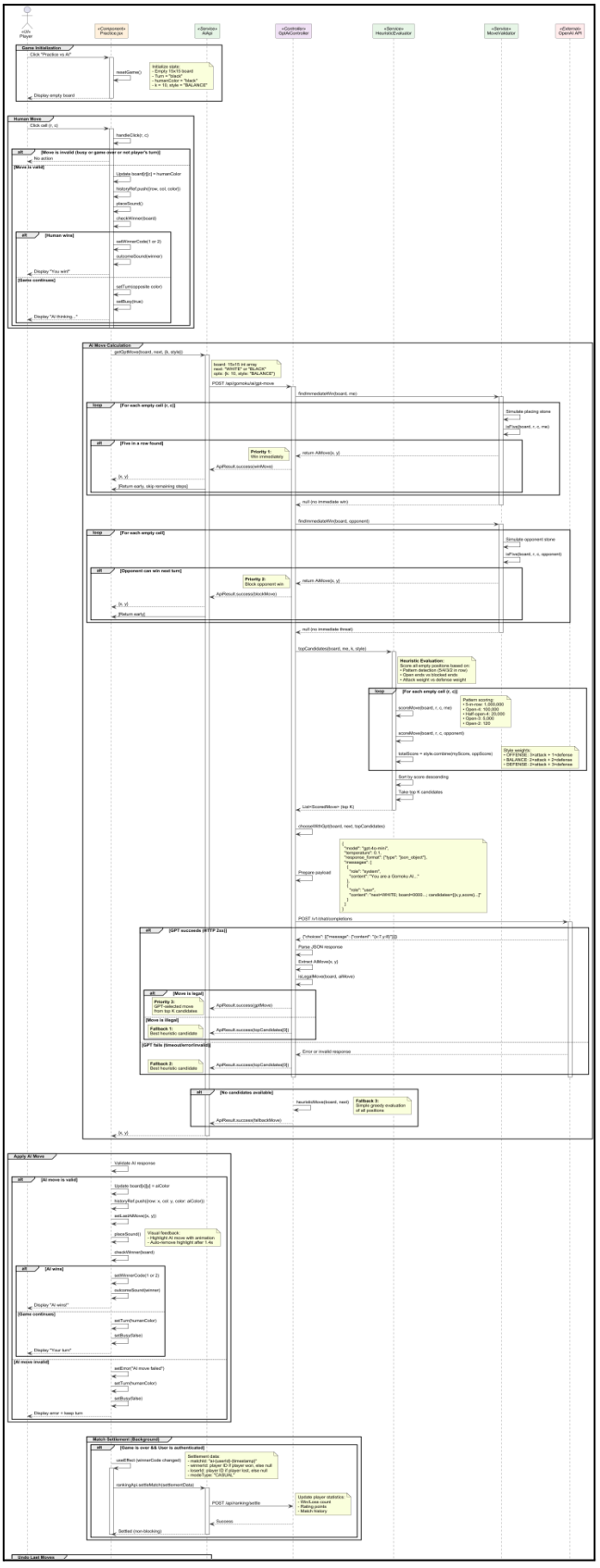


For design model:

Class diagram:



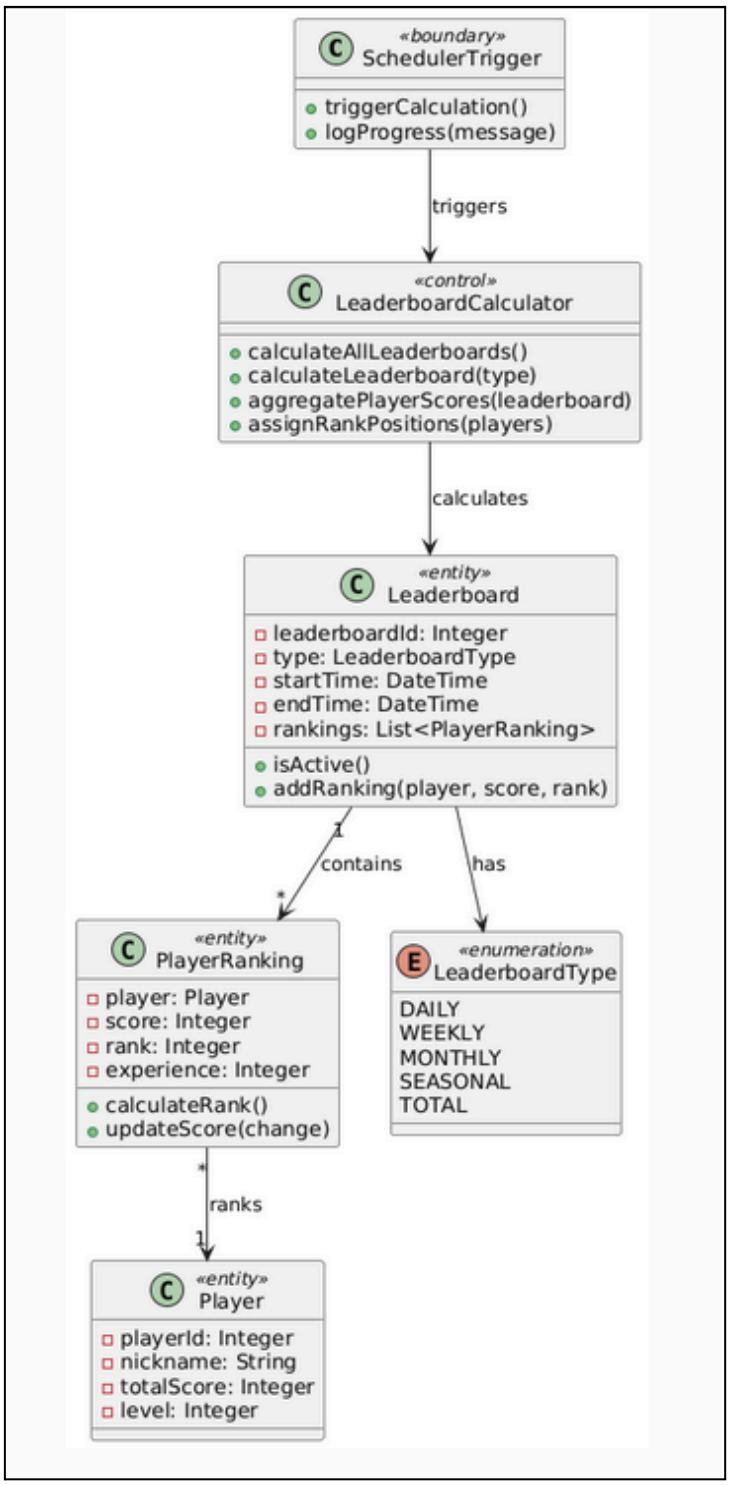
Sequence diagram:



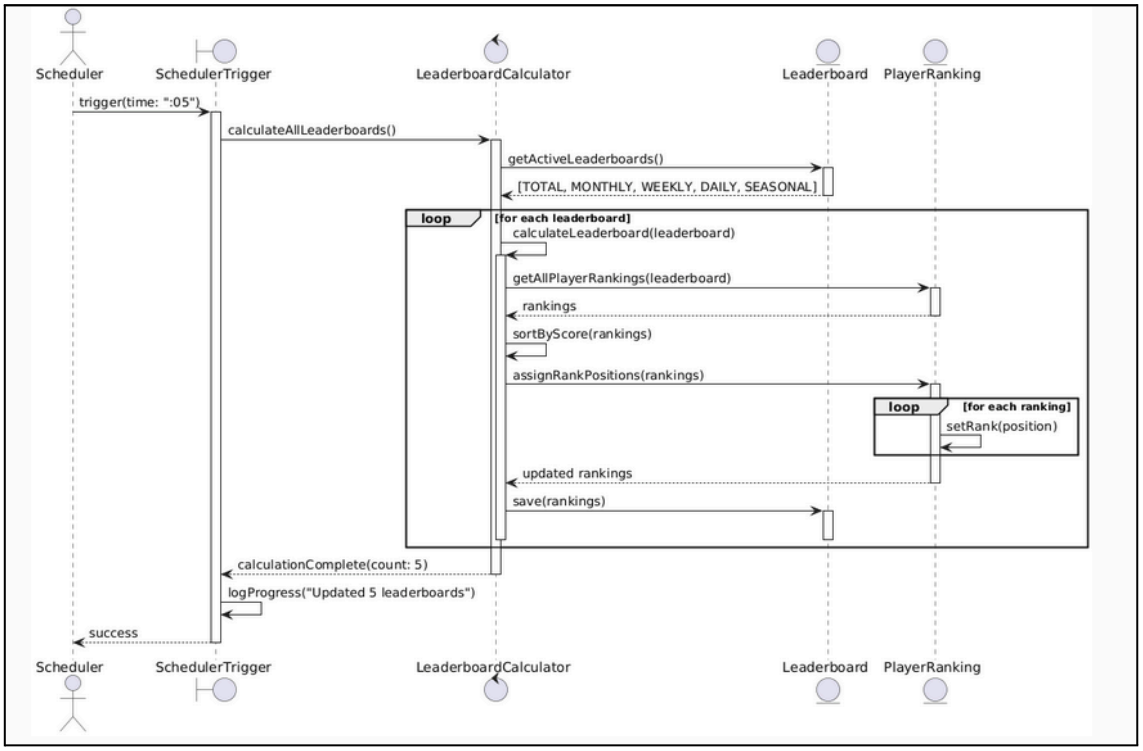
3.4.5 Use Case 5 by Cheng Muqin

For analysis model:

Class diagram:

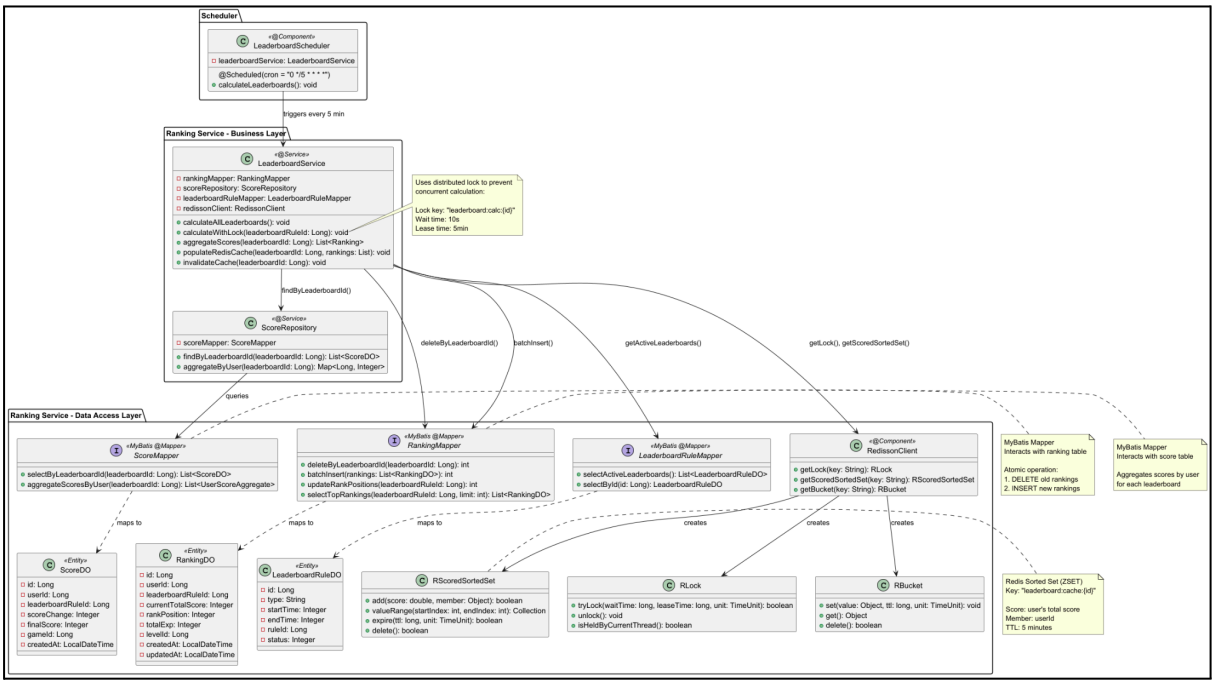


Sequence diagram:

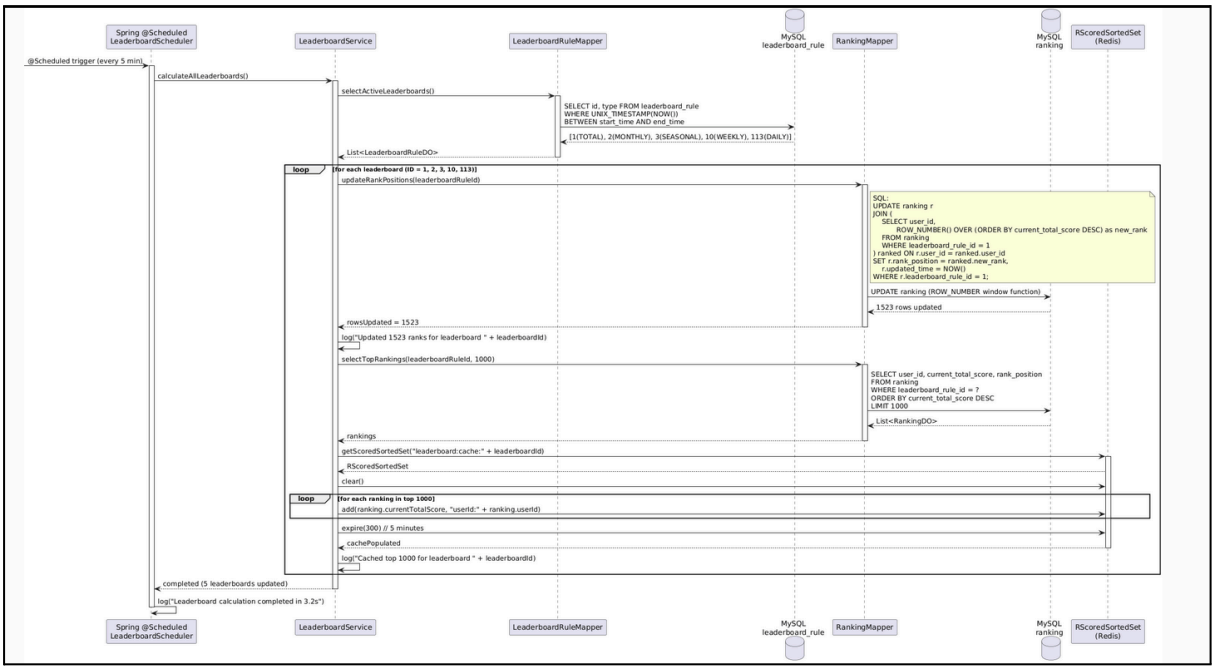


For design model:

Class diagram:



Sequence diagram:



3.5 Design Problems and Patterns

3.5.1 Design Problem 1 by Hao Tian(UC1: Register/Login)

Description: The authentication system must securely store passwords and generate/validate JWT tokens for user sessions. Storing plaintext passwords is a critical security vulnerability, and directly managing JWT logic in controllers leads to code duplication.

Brute Force Design: Controller directly handles password storage and JWT generation.

Problems with Brute Force:

Security Risk: Plaintext passwords vulnerable to database breaches

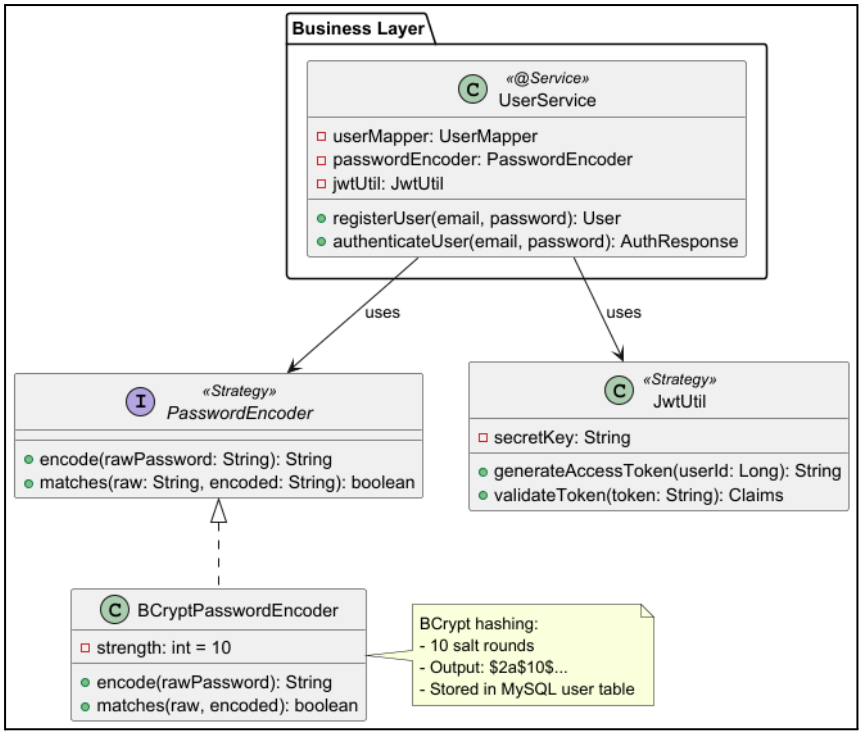
Code Duplication: JWT generation logic repeated in register/login/refresh endpoints

Violation of SRP: Controller handles both HTTP concerns and cryptography

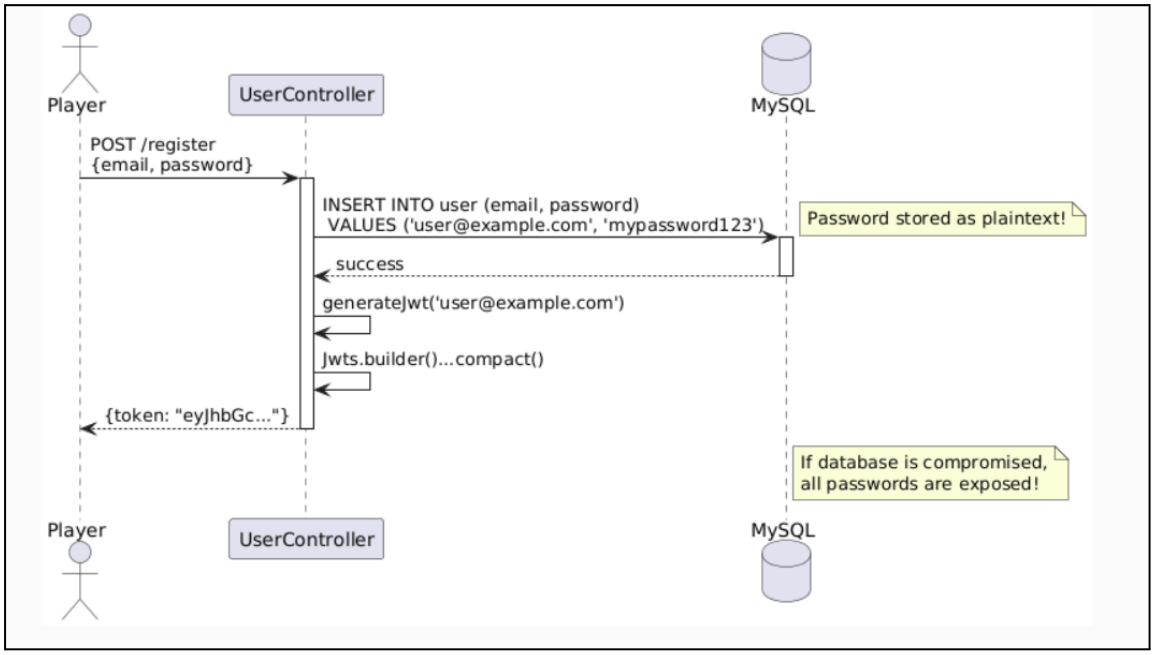
Hard to Test: Cannot unit test password hashing without hitting controller

Configuration Scattered: JWT secret, expiration time hardcoded

Class diagram:



Sequence diagram:



Candidate Design Patterns

1. Strategy Pattern(Chosen)

   Pros:

   Encapsulates password hashing algorithm (BCrypt, Argon2, PBKDF2)

   Encapsulates JWT generation/validation logic

   Easy to swap algorithms without changing business logic

   Testable in isolation

   Cons:

   Requires creating strategy interfaces and implementations

2. Facade Pattern

   Pros:

   Simplifies complex cryptography API

   Hides implementation details

   Cons:

   Doesn't provide algorithm flexibility

   Not ideal for "choose your hashing algorithm" scenario

3. Template Method Pattern

   Pros:

   Common authentication flow with customizable steps

   Cons:

Requires inheritance (less flexible than composition)

Overkill for simple password hashing

Motivation to Choose Strategy Pattern:

We selected Strategy Pattern for these reasons:

1. Algorithm Flexibility: Different parts of the system use different strategies:

   Password hashing: BCrypt with 10 salt rounds

   RSA encryption: 2048-bit keys for sensitive data transmission

   JWT signing: HS256 for access tokens

2. Security Best Practices: BCrypt automatically handles:

   Random salt generation (prevents rainbow table attacks)

   Configurable cost factor (10 rounds = ~100ms, protects against brute force)

   Built-in salt storage in hash output

3. Testability: Each strategy can be unit tested independently:

   ```
   @Test
   public void testBCryptStrategy_MatchesPassword() {
      PasswordEncoder encoder = new BCryptPasswordEncoder(10);
      String hashed = encoder.encode("password123");
      assertTrue(encoder.matches("password123", hashed));
   }
   ```

4. JWT Centralization: JwtUtil strategy encapsulates:

   Access token generation (15-minute expiry)

   Refresh token generation (30-day expiry)

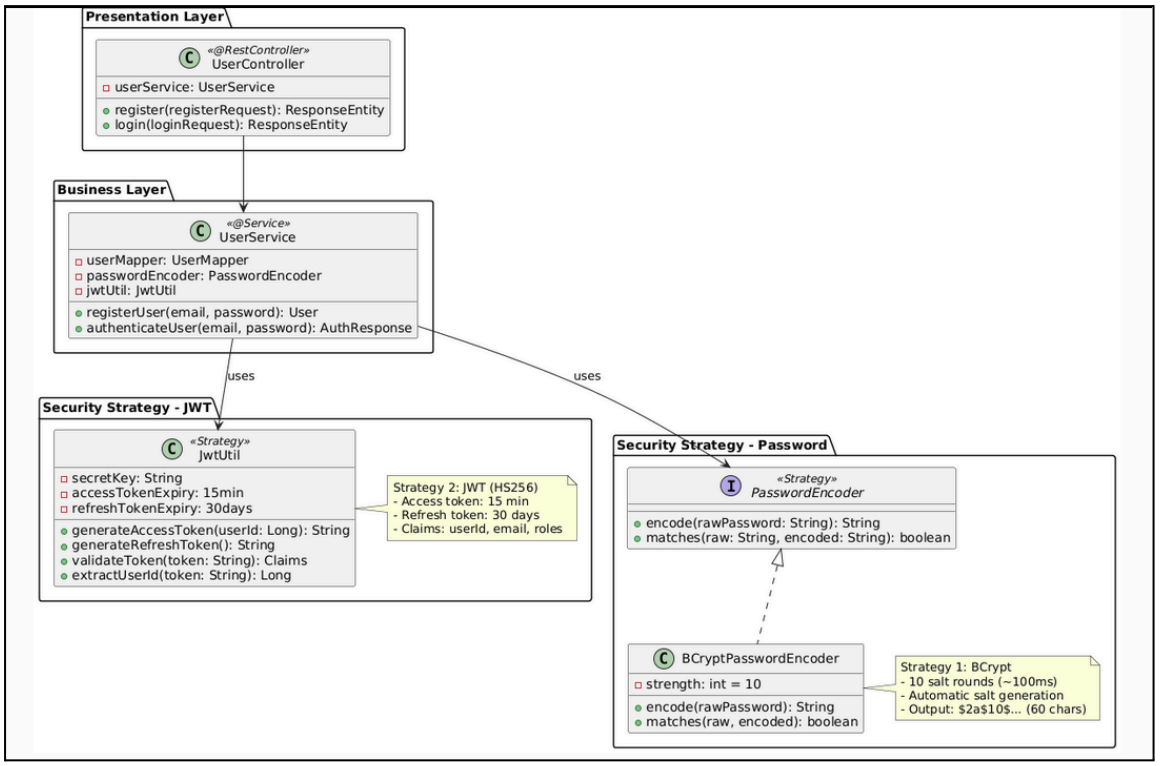
   Token validation with signature verification

   Claims extraction

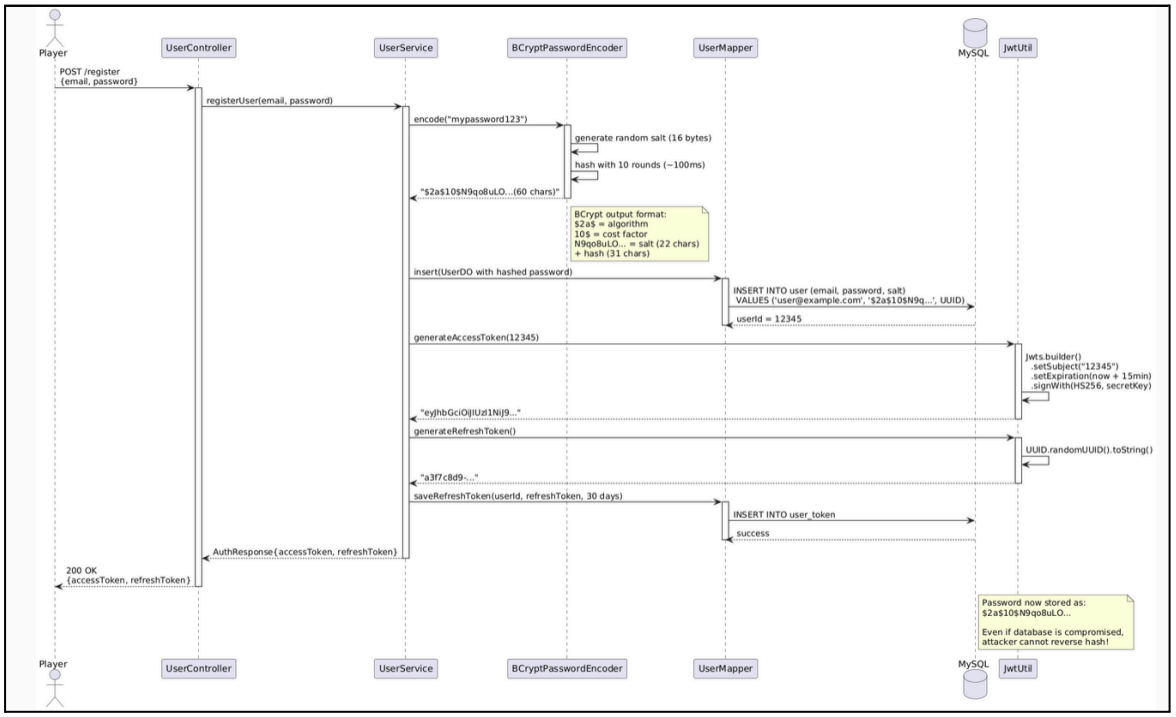
5. Spring Security Integration: Spring Boot provides PasswordEncoder interface:

   ```
   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder(10); // Strategy selection
   }
   ```

Class diagram:



Sequence diagram:



Implementation Decisions

1. BCrypt with 10 Salt Rounds

   Decision: Use BCrypt with cost factor = 10 (2^10 = 1024 iterations).

   Rationale:

   Security: 10 rounds takes ~100ms to hash, making brute force attacks impractical

   Performance: Fast enough for user registration/login (acceptable latency)

   Future-Proof: Can increase rounds as CPU power grows (currently 10-12 recommended)

2. Dual-Token System (Access + Refresh)

   Decision: Generate two types of tokens:

   Access Token: Short-lived (15 minutes), includes user claims

   Refresh Token: Long-lived (30 days), random UUID stored in database

   Rationale:

   Security: Short access token expiry limits damage if token is stolen

   User Experience: Refresh token allows seamless re-authentication without re-login

   Revocability: Refresh tokens in database can be invalidated (logout, password change)

3. Single-Device Enforcement via Database

   Decision: Store only one active refresh token per user in user_token table.

   Rationale:

   Security: Prevents token theft by limiting concurrent sessions

   Logout Support: Deleting refresh token from database logs out user

   Account Takeover Detection: If user logs in from new device, old session is invalidated

4. RSA-2048 for Password Transmission

   Decision: Frontend encrypts password with RSA-2048 public key before sending to backend.

   Rationale:

   HTTPS Defense-in-Depth: Even with HTTPS, RSA adds extra protection layer

Man-in-the-Middle Protection: Password encrypted even if HTTPS is compromised

Client-Side Security: Password never exists in plaintext in browser's network logs

3.5.2 Design Problem 2 by Li YuanXing(UC2: Join/Create Room)

Problem Description: When players create game rooms, the system must generate unique room codes for players to share. Using sequential IDs (1, 2, 3...) is predictable and allows players to guess valid room codes. Random codes must be checked for collisions, and the code-to-room-ID mapping must be fast.

Brute Force Design: Sequential room IDs exposed directly.

Problems with Brute Force:

Predictable Codes: Players can guess codes (1, 2, 3...) and join random games

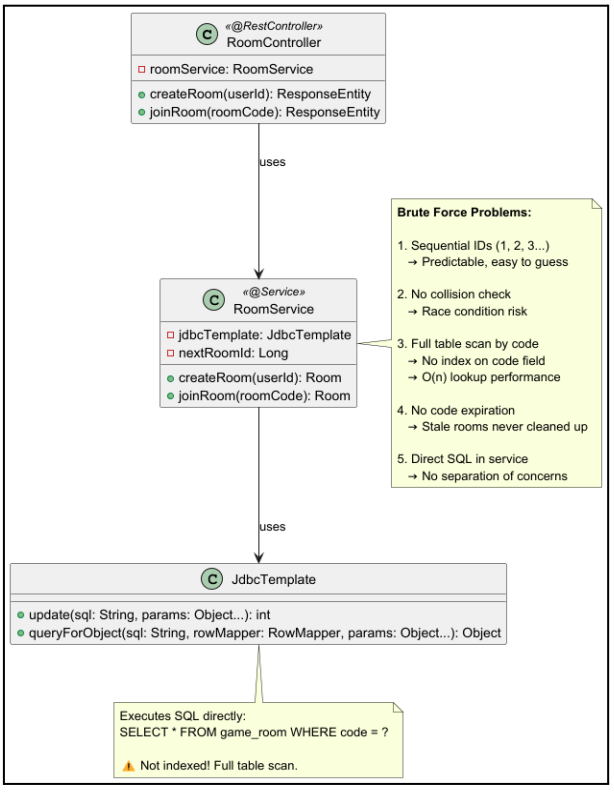
No Collision Prevention: If two requests generate same random code, constraint violation

Slow Lookup: Query game_room table by code (not indexed primary key)

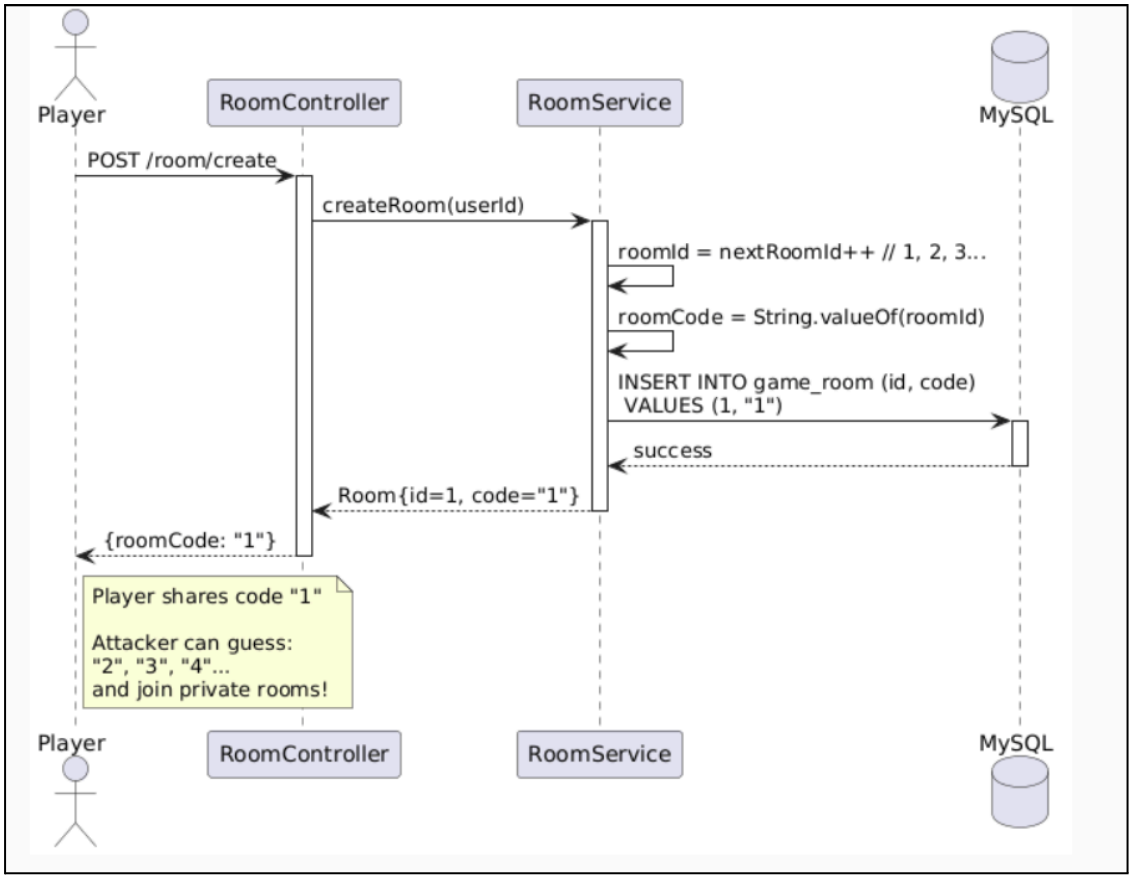
No Expiration: Old room codes remain valid forever

Database Load: Every join request hits database

Class diagram:



Sequence diagram:



Candidate Design Patterns

1. Factory Pattern + Caching (Chosen)

   Pros:

   Encapsulates room code generation logic

   Can add collision retry logic

   Redis cache for fast code lookup

   TTL for automatic expiration

   Cons:

   Requires Redis infrastructure

2. UUID Generation

   Pros:

   Guaranteed unique (no collision check needed)

   Cons:

   36 characters (too long for users to type)

   Not user-friendly ("a3f7c8d9-4e2b-...")

3. Hash-Based Codes

   Pros:

   Derived from room ID (deterministic)

   Cons:

   Still requires collision check

   Hash collisions possible

Motivation to Choose Factory Pattern and Caching:

We selected Factory Pattern with Redis Caching for these reasons:

1. User-Friendly Codes: 6-character alphanumeric codes (e.g., "A3F7C8") are:

   Easy to read aloud over voice chat

   Short enough to type on mobile

   36^6 = 2.1 billion possible combinations (collision-resistant)
2. Fast Lookup: Redis hash map provides O(1) lookup:

   Redis Key: "room:code:A3F7C8"

   Redis Value: 12345 (room ID)

   TTL: 30 minutes
3. Automatic Expiration: Redis TTL auto-deletes expired codes (30 minutes):

   Reduces namespace pollution

   Forces players to create new rooms (prevents stale sessions)

   No cron job needed for cleanup
4. Collision Prevention: Factory retries code generation up to 3 times:

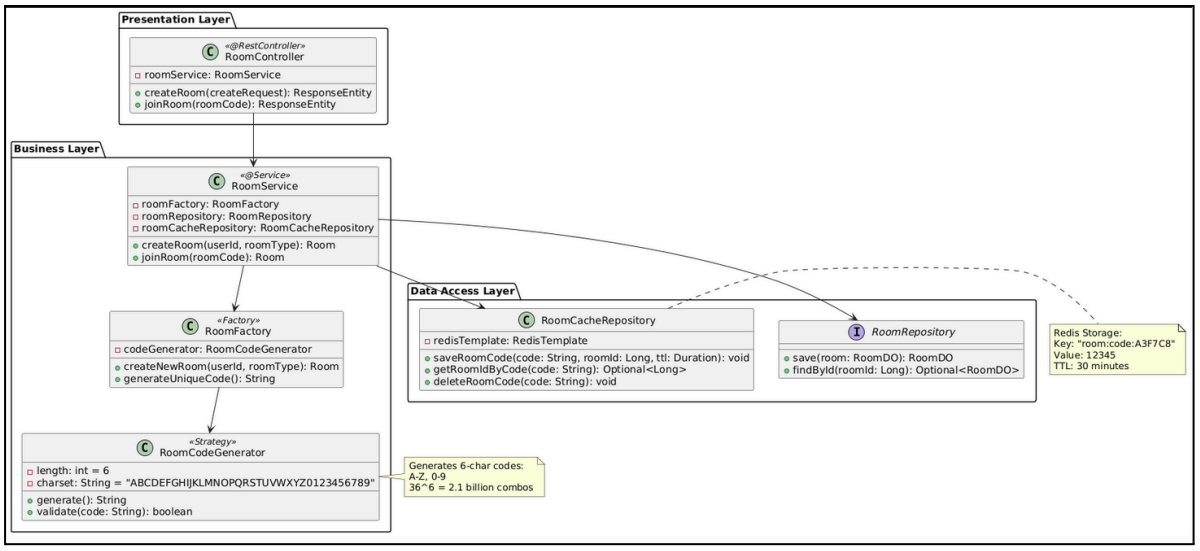
```
for (int attempt = 0; attempt < 3; attempt++) {
   String code = generateRandomCode(6);
   boolean exists = redisTemplate.hasKey("room:code:" + code);
   if (!exists) return code; // Success
}
throw new CodeGenerationException();
```

5. Database Offloading: 99% of join requests hit Redis cache (not MySQL):

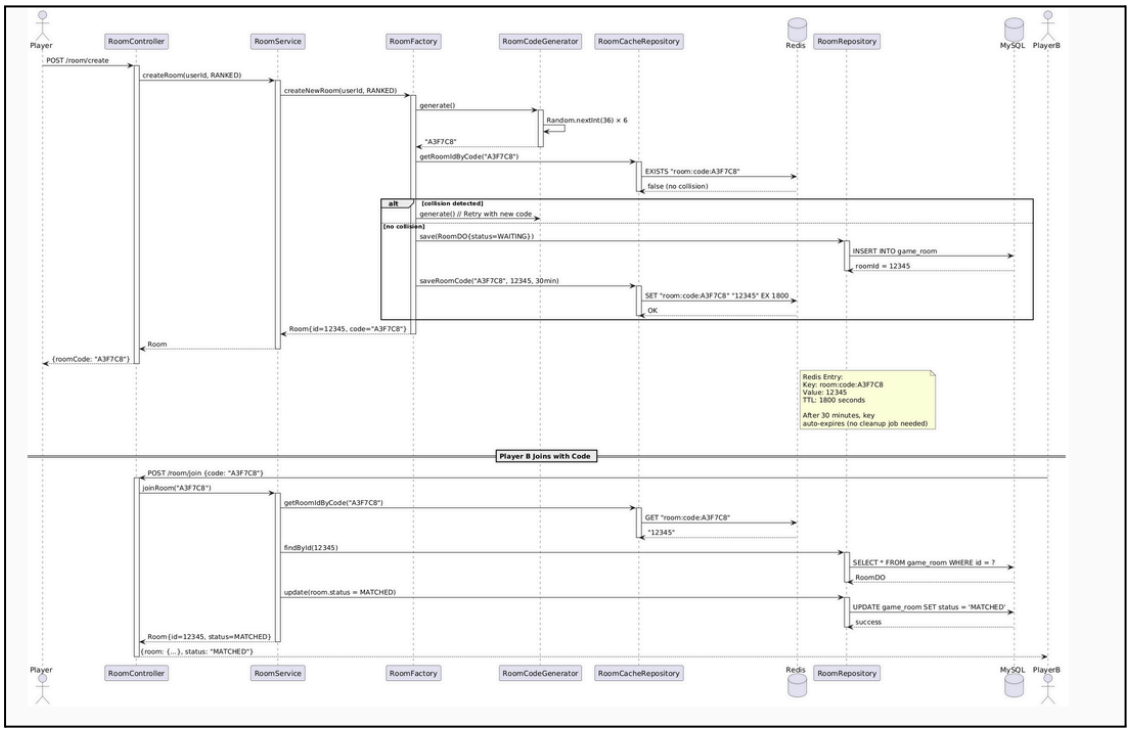
   Cache hit: Redis O(1) lookup (~1ms)

Cache miss: Fallback to MySQL query (~10ms)

Class diagram:



Sequence diagram:



Implementation Decisions

1. 6-Character Alphanumeric Codes

   Decision: Use charset A-Z0-9 (36 characters) with length 6.

   Rationale:

   Collision Resistance: 36^6 = 2,176,782,336 possible codes

   User-Friendly: No ambiguous characters (removed "0O", "1Il")

Voice Chat Compatibility: Letters and numbers easy to communicate verbally

2. Redis TTL of 30 Minutes

   Decision: Room codes expire after 30 minutes of inactivity.

   Rationale:

   Prevents Stale Codes: If creator leaves, code becomes invalid automatically

   Namespace Cleanup: No manual deletion job needed (Redis handles it)

   Reasonable Window: 30 minutes is enough for players to share code and join

3. Collision Retry Logic (Max 3 Attempts)

   Decision: If generated code already exists in Redis, retry up to 3 times.

   Rationale:

   Collision Probability: With 2.1 billion codes and ~1000 active rooms, collision rate < 0.0001%

   Three Strikes: After 3 failures, throw exception (indicates system issue)

   Fast Failure: Don't retry forever (prevents infinite loops)

4. Cache-Aside Pattern for Fallback

   Decision: If Redis is down, fall back to MySQL query.

   Rationale:

   High Availability: System remains functional even if Redis fails

   Graceful Degradation: Performance degrades but service doesn't crash

   Monitoring: Log cache misses to detect Redis issues

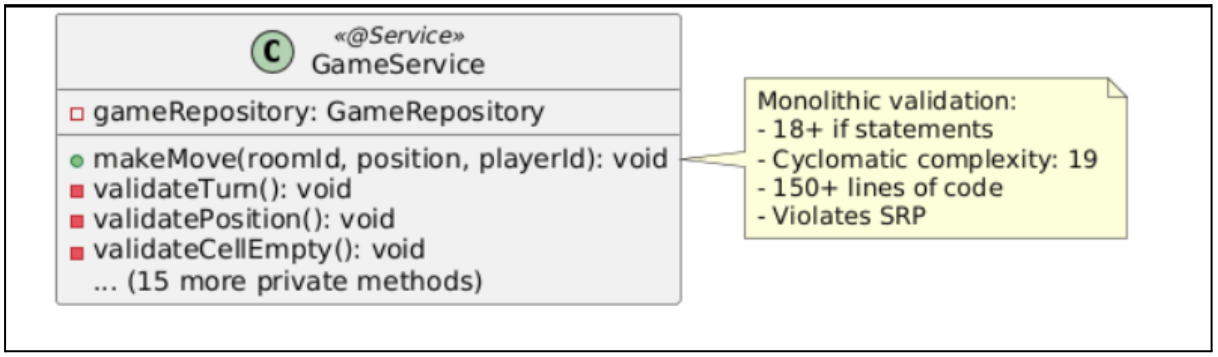
3.5.3 Design Problem 3 by Hao Tian

Problem Description: Processing a game move requires 18+ independent validations (turn check, bounds check, cell empty, game status, etc.) before executing the move. A monolithic method with deeply nested if-else statements violates SRP and is difficult to test/maintain.

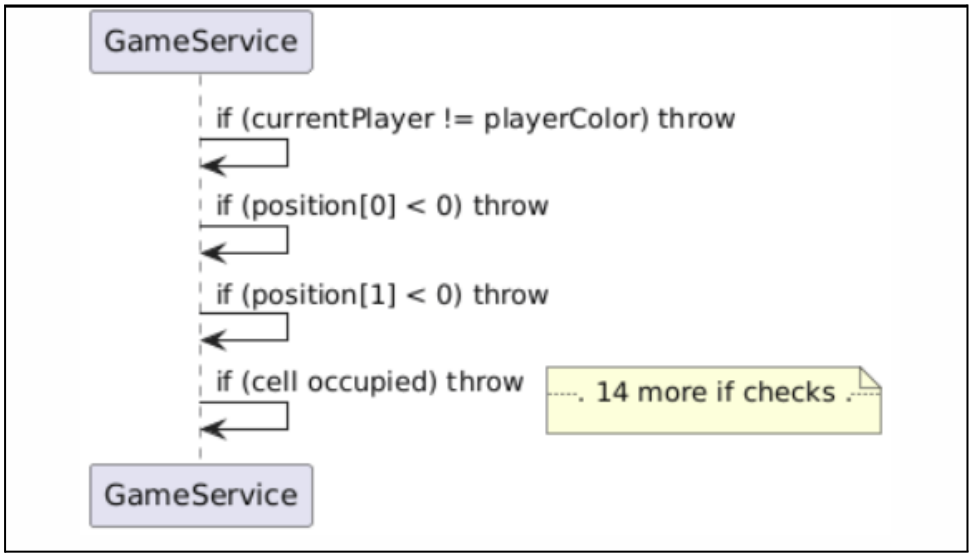
Brute Force Design: Monolithic makeMove() method with 18+ if statements.

Problems: Cyclomatic complexity 19, violates SRP, difficult to test, duplicated validation logic.

Class diagram:



Sequence diagram:



Candidate Design Patterns

1. Chain of Responsibility Pattern (Chosen)

   Pros: Decouples validations, easy to add/remove, single responsibility per chain

   Cons: Many small classes

2. Strategy Pattern

   Pros: Encapsulates validation algorithms

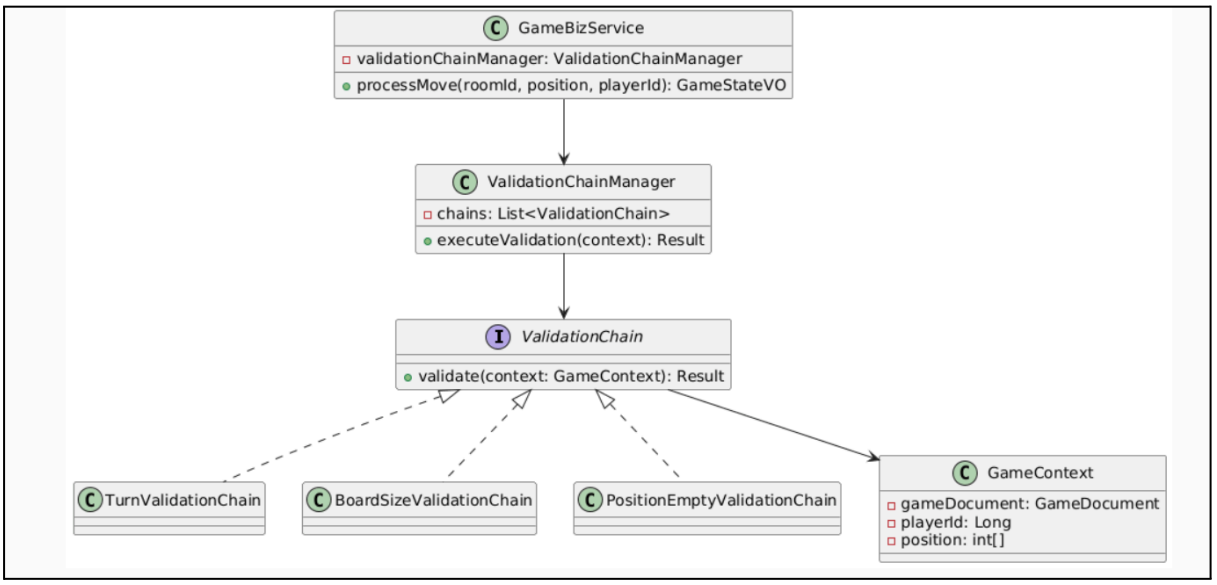
Cons: Doesn't fit "run all validations" scenario

Motivation to Choose Chain of Responsibility:

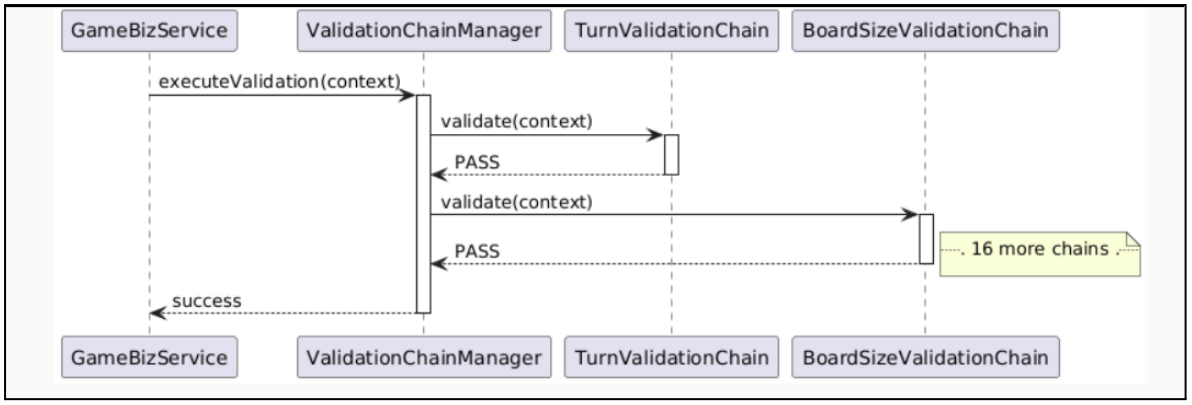
We selected Chain of Responsibility Pattern because:

1. Sequential Validation: Each validation must pass before proceeding (natural chain behavior)
2. Open/Closed Principle: Add new validations without modifying existing code
3. Single Responsibility: Each chain handles one rule (TurnValidationChain, BoardSizeValidationChain, etc.)
4. Testability: Each chain tested independently
5. Separation: 18 validation chains (read-only) + 16 execution chains (state changes)

Class diagram:



Sequence diagram:



Implementation Decisions

1. Two Separate Chain Managers

   ValidationChainManager (18 chains): Read-only checks

   ExecuteChainManager (16 chains): State changes
2. Shared GameContext Object

   Reduces parameters, easy to extend.
3. Early Exit on Failure

   Fail-fast: stop chain on first validation failure.

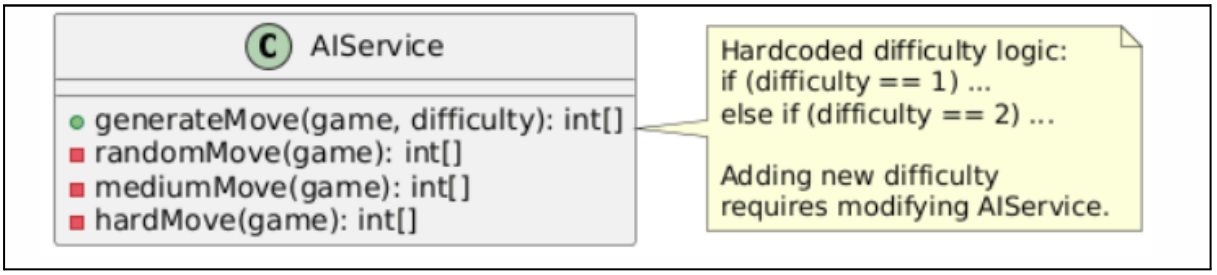
3.5.4 Design Problem 4 by Shashank Bagda

Problem Description: The AI opponent must provide multiple difficulty levels (easy, medium, hard) with different move evaluation strategies. Hardcoding if-else logic for each difficulty level violates OCP and makes it difficult to add new difficulty levels.

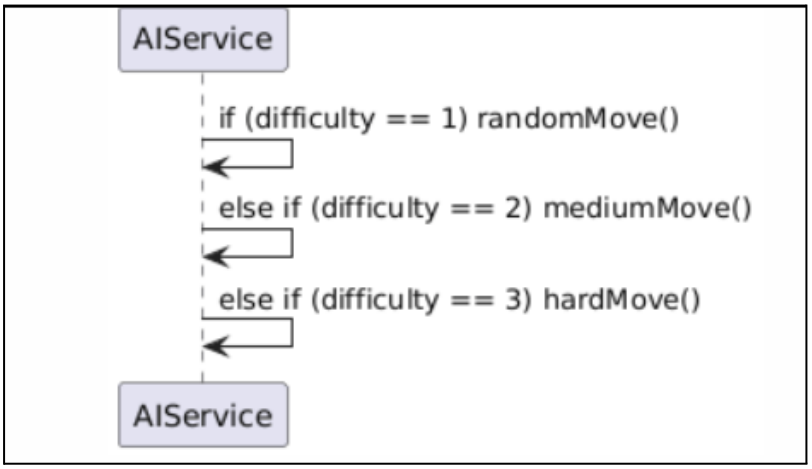
Brute Force Design: Hardcoded difficulty logic.

Problems: Violates OCP, difficulty logic scattered, hard to add new levels.

Class diagram:



Sequence diagram:



Candidate Design Patterns

1. Strategy Pattern (Chosen)

   Pros: Encapsulates difficulty algorithms, easy to add levels, OCP compliant

   Cons: Requires strategy interface

2. State Pattern

   Pros: Encapsulates difficulty states

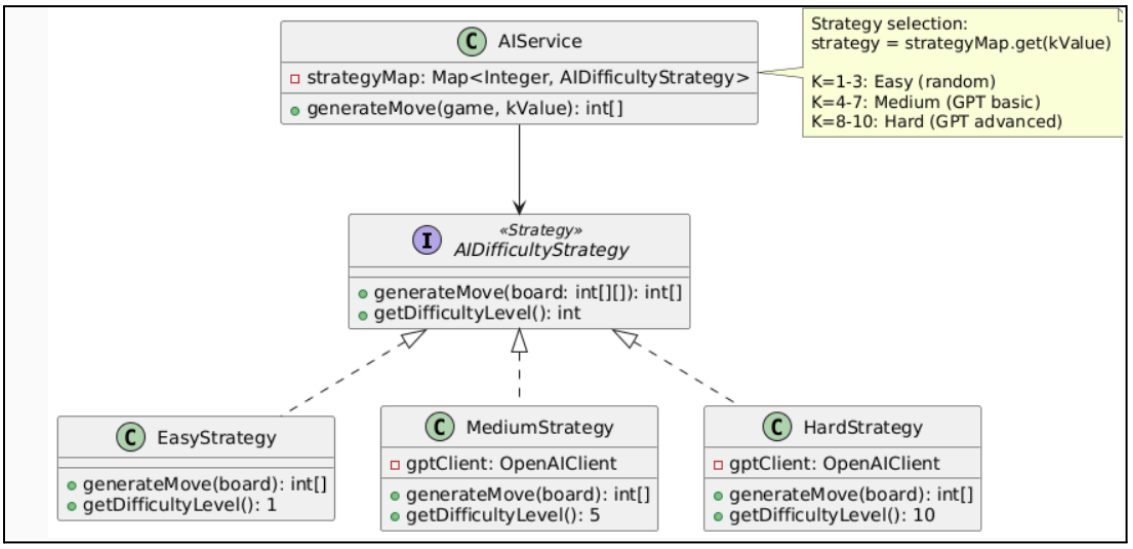
   Cons: Overkill for static difficulty selection

Motivation to Choose Strategy Pattern:

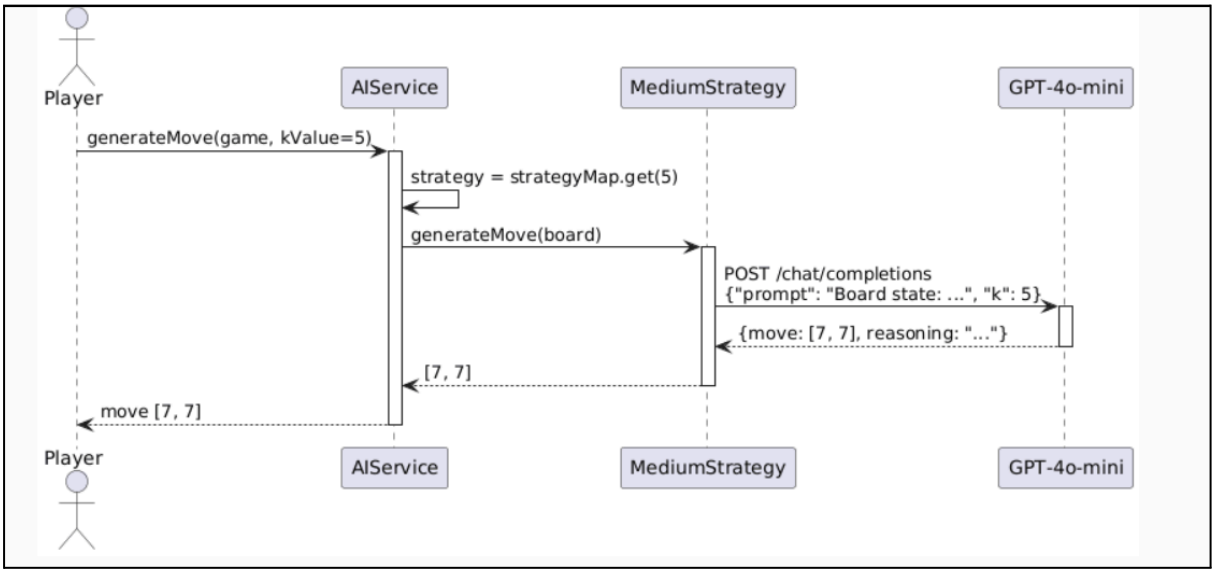
We selected Strategy Pattern because:

1. Algorithm Family: Each difficulty level is a different move evaluation algorithm
2. Runtime Selection: Player chooses difficulty (K-value) when creating AI room
3. Open/Closed: Add new difficulty levels without modifying existing code
4. Encapsulation: Each strategy encapsulates its evaluation logic (GPT-4o-mini prompt engineering)

Class diagram:



Sequence diagram:



Implementation Decisions

1. Three Strategy Implementations

   Decision: Use three concrete strategies for K-value ranges:

   EasyStrategy (K=1-3): Random valid move

   MediumStrategy (K=4-7): GPT-4o-mini with basic prompt

HardStrategy (K=8-10): GPT-4o-mini with advanced prompt (threat analysis)

2. Strategy Registry Pattern

   Decision: Use a Map<Integer, AIDifficultyStrategy> to register strategies.

   Rationale:

   O(1) strategy lookup by K-value

   Easy to add new strategies via Spring configuration

   Clear mapping between K-value and strategy

3. GPT-4o-mini Integration with K-Value

   Decision: Pass K-value to GPT as part of the prompt to control difficulty.

   Rationale:

   K-value guides GPT's evaluation depth

   Higher K = more detailed board analysis

   Consistent difficulty scaling

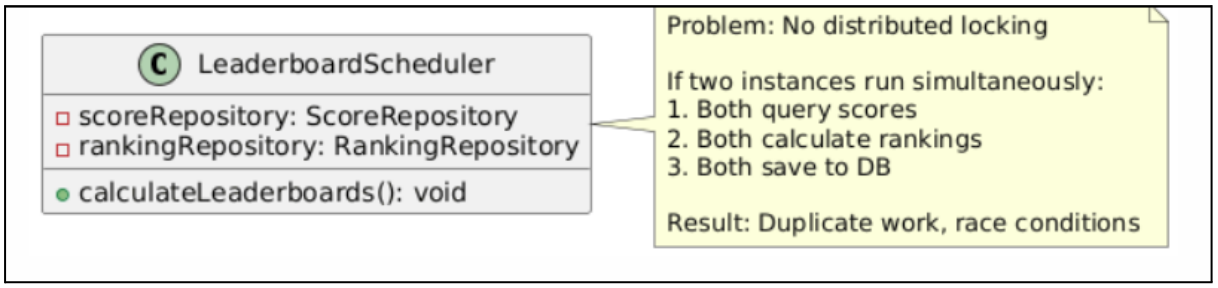
3.5.5 Design Problem 5 by Cheng Muqin

Problem Description: The leaderboard calculation runs as a scheduled cron job every 5 minutes. Multiple job instances might run concurrently (in distributed deployment), leading to duplicate calculations and inconsistent rankings.

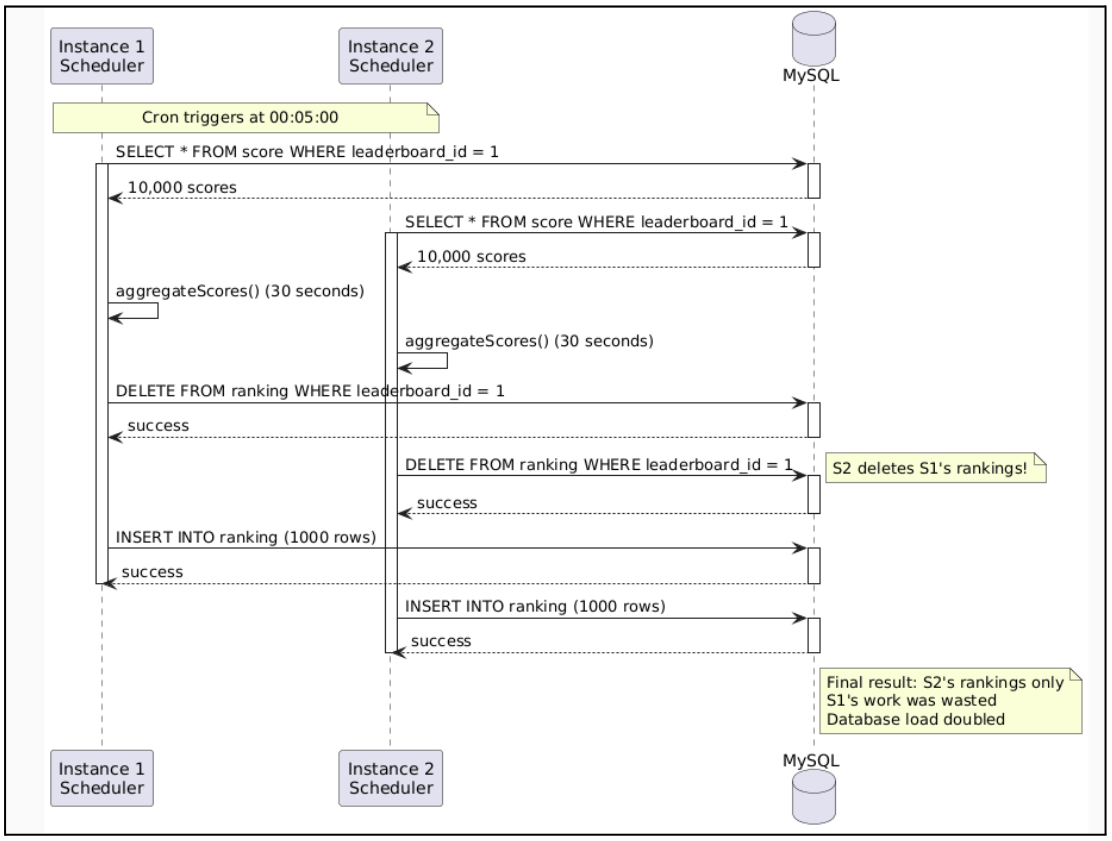
Brute Force Design: Cron job without locking.

Problems: Duplicate calculations, race conditions, inconsistent rankings, database contention.

Class diagram:



Sequence diagram:



Candidate Design Patterns

1. Distributed Locking (Redis) (Chosen)

   Pros: Prevents concurrent execution, works in distributed systems

Cons: Requires Redis infrastructure

2. Database Locking (SELECT FOR UPDATE)

   Pros: No external dependency

   Cons: Doesn't prevent duplicate work (both instances still query)

3. Leader Election (Zookeeper)

   Pros: Only one instance runs job

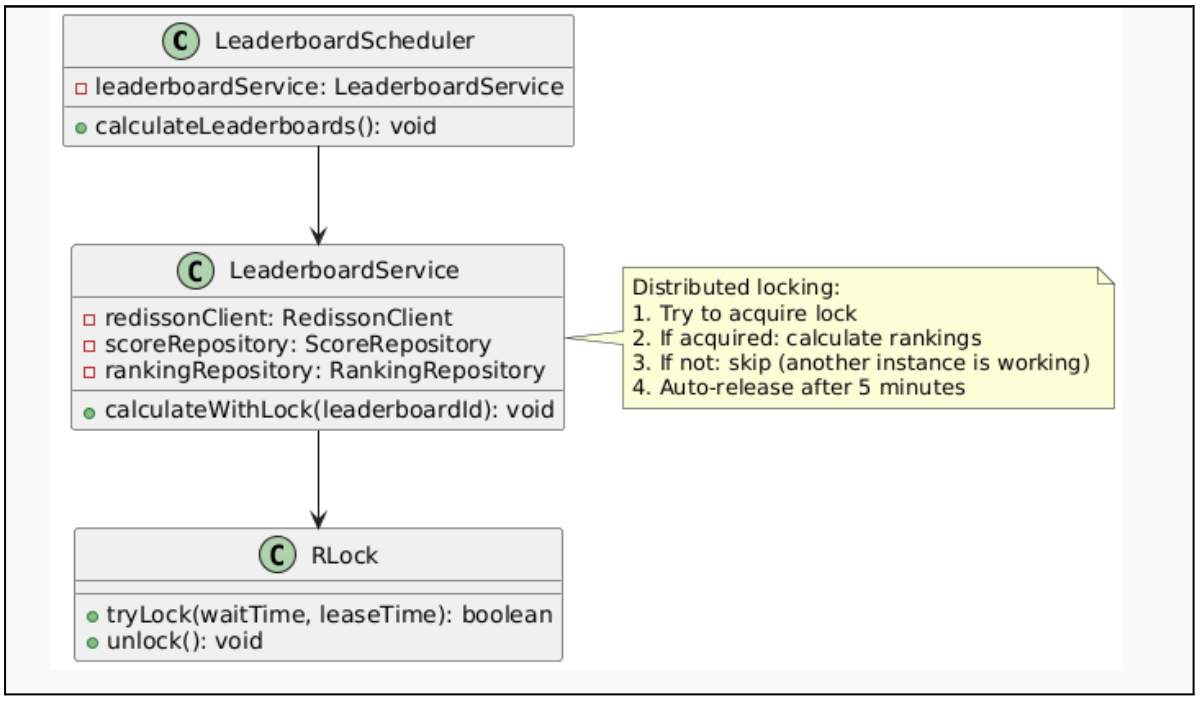
   Cons: Complex infrastructure, overkill for simple locking

Motivation to Choose Distributed Locking:

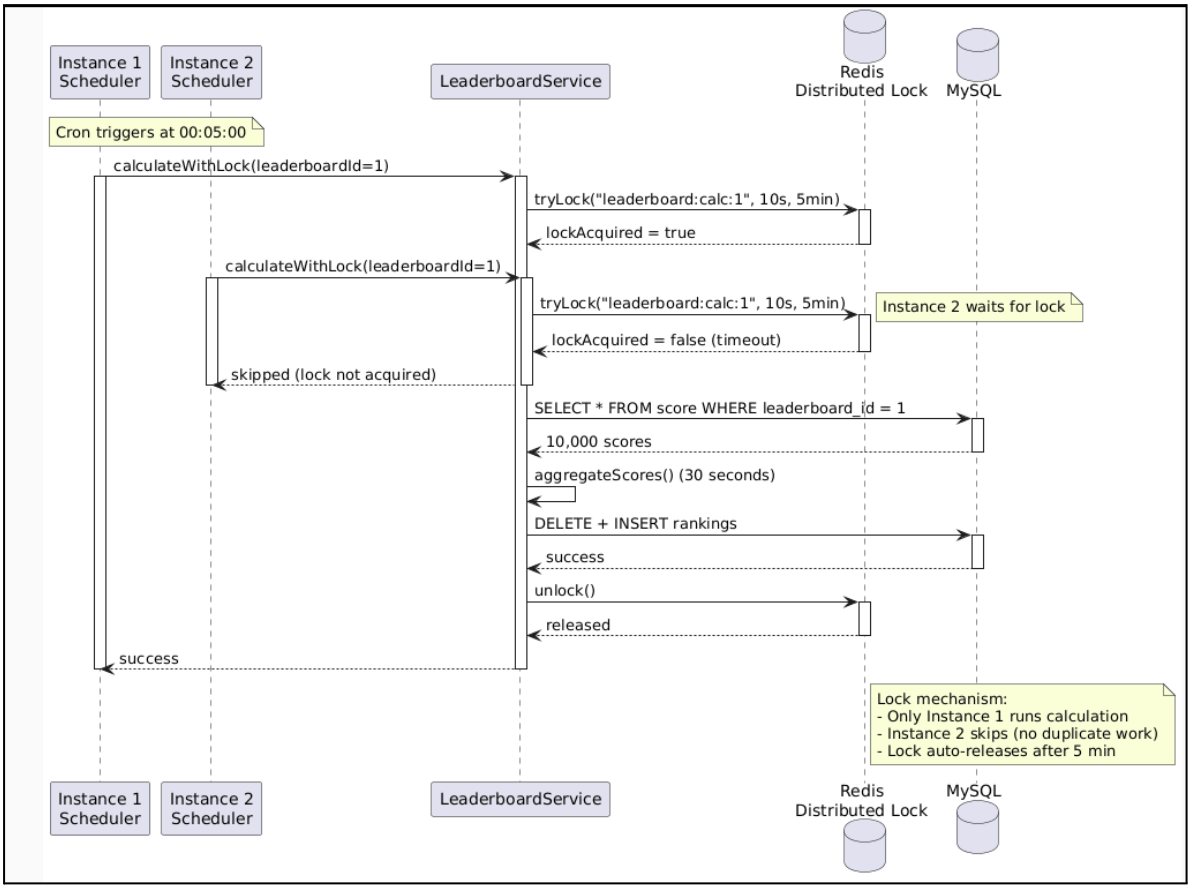
We selected Distributed Locking with Redis because:

1. Prevents Duplicate Work: Only one instance acquires lock and runs calculation
2. Automatic Timeout: Lock auto-releases after TTL (prevents deadlock if instance crashes)
3. Fair Queuing: Instances retry acquiring lock (eventual consistency)
4. Redis Already Used: Leverages existing Redis infrastructure (no new dependencies)

Class diagram:



Sequence diagram:



Implementation Decisions

1. Per-Leaderboard Locking

   Decision: Use separate locks for each leaderboard (leaderboard:calc:{id}).

   Rationale:

   Parallel calculation for different leaderboards (daily, weekly, monthly)

   Reduces lock contention (finer-grained locking)

   One leaderboard failure doesn't block others

2. Lock Timeout Configuration

   Decision:

   Wait time: 10 seconds (how long to wait for lock)

   Lease time: 5 minutes (auto-release if holder crashes)

   Rationale:

   10s wait: Long enough to account for calculation finishing soon, short enough to not block cron job

5min lease: Typical calculation takes 30-60s, 5min buffer prevents deadlock

Auto-release: If instance crashes mid-calculation, lock auto-releases after 5min

3. Cron Job Integration

   Decision: Scheduler tries to acquire lock, skips if unavailable.

4. Cache Invalidation After Calculation

   Decision: Invalidate Redis leaderboard cache after successful calculation.

   Rationale:

   Ensures frontend fetches fresh rankings

   Cache-aside pattern: next request populates cache with new data

3.6 Database Schemas

3.6.1 Overview

The Gomoku Platform employs a **polyglot persistence architecture** with three complementary databases:

- **MySQL 8.0.28**: Transactional data (users, rooms, rankings)
- **MongoDB 6.0**: Flexible game state and history
- **Redis 7.x**: High-performance caching and queues

3.6.2 ER Diagram

3.6.2.1 user

Purpose:

Core user account storage with secure authentication.

Key Logic:

- **Password Security:**
  - Uses BCrypt hashing with 60-character hash storage. The password_salt (32-character UUID) is stored separately but always used together with the hash during verification.
- **Email as Primary Identifier:**
  - Email serves as the unique login credential (not username). This ensures one account per email and simplifies password recovery.
- **Avatar Flexibility:**
  - Supports both avatar_url (external links) and avatar_base64 (embedded images) to handle different upload scenarios.
  - Base64 is used for small avatars uploaded directly; URLs for externally hosted images.
- **Account Status Management:**
  - The status field (0=inactive, 1=active, 2=disabled, 3=deleted) enables soft deletion and account suspension without losing user data.
  - New accounts start as inactive (0) until email verification, then transition to active (1).

Usage Pattern:

- Registration → Insert with status=0 (inactive)
- Email Verification → Update status=1 (active)
- Admin Ban → Update status=2 (disabled)
- Account Deletion → Update status=3 (deleted, not physically deleted)

3.6.2.2 user_token

Purpose:

Manages JWT refresh tokens for extending user sessions beyond the short-lived access token expiration.

Key Logic:

- **One Token Per User:**
  - The unique constraint on user_id ensures each user has only one active refresh token. When a user logs in from a new device, the old token is replaced (single-session enforcement).
- **Token Rotation:**
  - On refresh, both the refresh_token and expires_at are updated, invalidating the old token immediately.
  - This prevents replay attacks if an old token is compromised.
- **Expiration Strategy:**
  - Access tokens expire in 15 minutes; refresh tokens expire in 30 days.
  - This balances security (short-lived access) with usability (infrequent re-login).

Refresh Flow:

1. Client sends expired access token + valid refresh token
2. Backend validates refresh_token exists and expires_at > NOW()
3. Generate new access token (15 min) and new refresh token (30 days)
4. Update user_token table with new refresh_token and expires_at
5. Return both tokens to client

3.6.2.3 game_room

Purpose:

Central registry for all game rooms across casual, ranked, and private modes.

Key Logic:

- **Room Code System:**

  - The room_code (6-character alphanumeric) enables easy room sharing without exposing internal IDs. Codes are generated randomly and stored in both MySQL (permanent) and Redis (30-min TTL for fast lookup).

  - Player Assignment: player1_id is always the room creator; player2_id is filled when someone joins (private/casual) or when matchmaking finds a pair (ranked).

- **Status Lifecycle:**
  - **WAITING (0)**: Room created, waiting for second player
  - **MATCHED (1)**: Two players assigned, waiting for both to ready up
  - **PLAYING (2)**: Game active (both players readied)
  - **FINISHED (3)**: Game ended, ready for archival
- **Type Differentiation:**
  - type field (0=casual, 1=ranked) determines whether the match affects ELO rankings.
  - Only ranked matches (type=1) trigger score updates in the score and ranking tables.

Room Creation Scenarios:

- **Casual Mode:**
  - User creates room → room_code generated → friend joins via code
- **Ranked Matchmaking:**
  - System creates room → assigns both players → no room code
- **Private Room:**
  - User creates room → sets password (stored in Redis) → friends join

3.6.2.4 level

Purpose:

Defines the 10-level progression system with experience thresholds.

Key Logic:

- **Non-Linear Growth:**
  - Experience requirements increase non-linearly (0 → 10 → 30 → 60 → 100…) to provide achievable early levels and challenging high-tier progression.

- **Configuration Data:**
  - This table is pre-populated during system initialization and rarely changes. It's essentially a lookup table for level calculations.
- **Level Calculation:**
  - To determine a player's level, sum their total experience and find the highest level.id where total_exp >= exp_required.

Example:

Player has 75 total_exp:

- Level 1: 0 required ✓ (75 >= 0)
- Level 2: 10 required ✓ (75 >= 10)
- Level 3: 30 required ✓ (75 >= 30)
- Level 4: 60 required ✓ (75 >= 60)
- Level 5: 100 required ✗ (75 < 100)

→ **Player is Level 4, needs 25 more exp for Level 5**

3.6.2.5 level_rule

Purpose:

Defines experience point rewards based on game mode and match outcome.

Key Logic:

- **Asymmetric Rewards:**
  - Winners get significantly more experience than losers to incentivize improvement. Even losses grant some experience to avoid player frustration.
- **Mode-Based Scaling:**
  - **RANKED (highest):** 50 exp for win, 10 for loss (competitive emphasis)
  - **CASUAL (medium):** 20 exp for win, 5 for loss (fun, low pressure)
  - **PRIVATE (lowest):** 15 exp for win, 3 for loss (informal play)
  - **Draw Handling:** Draws grant moderate experience (20/10/8) between win and loss amounts, treating it as a neutral outcome.

3.6.2.6 score_rule

Purpose:

Defines ELO score changes for ranked matches (flattened structure for all 9 combinations).

Key Logic:

- **Flattened Design:**
  - Instead of a normalized structure with separate rows, it uses 9 columns (ranked_win_score, casual_win_score, etc.) for simpler queries. One row defines all scoring rules.
- **Ranked-Only Scoring:**
  - Only ranked_ columns have non-zero values (+25 win, -15 loss, 0 draw). Casual and private modes don't affect scores, encouraging players to try ranked mode for progression.
- **Zero-Sum Alternative:**

- The current setup is NOT zero-sum (+25/-15 means net +10 per match). This intentionally inflates scores over time to make players feel progression. A true zero-sum ELO would use +25/-25.

- **Seasonal Variation:**
  - While currently one “Standard” rule exists, the design supports creating “Season1”, “Season2” rules with different scoring (e.g., Season1 could use +30/-20 for faster progression).

3.6.2.7 score

Purpose:

Immutable audit log of every ranked match’s score changes (like a blockchain ledger).

Key Logic:

- **Historical Record:**
  - Every ranked match generates TWO score records (one per player). This creates a complete audit trail of how players’ scores evolved over time.
- **Cumulative Tracking:**
  - final_score stores the player’s total score AFTER this match, enabling score history charts: [1000 → 1025 → 1010 → 1035...].
- **Leaderboard Linking:**
  - leaderboard_rule_id associates scores with time periods. A match played on 2025-11-13(example) generates scores for:
    - DAILY leaderboard (Nov 13)
    - WEEKLY leaderboard (Week 46)
    - MONTHLY leaderboard (November)
    - SEASONAL leaderboard (Q4 2025)
    - TOTAL leaderboard (all-time) (experience)

This means ONE match creates 5 score records per player (10 total inserts).

Multi-Leaderboard Logic:

Match ends at 2025-11-13 14:30:

For Player A (winner, +25 points):

- Insert into score: leaderboard_rule_id=1 (TOTAL), score_change=+25, final_score=1525
- Insert into score: leaderboard_rule_id=2 (MONTHLY), score_change=+25, final_score=1525
- Insert into score: leaderboard_rule_id=3 (SEASONAL), score_change=+25, final_score=1525
- Insert into score: leaderboard_rule_id=10 (WEEKLY), score_change=+25, final_score=1525
- Insert into score: leaderboard_rule_id=113 (DAILY Nov 13), score_change=+25, final_score=1525

For Player B (loser, -15 points):

- Same 5 inserts with score_change=-15, final_score=985

Result:

- 10 new rows in score table

3.6.2.8 leaderboard_rule

Purpose:

Defines time periods for different leaderboard types (daily, weekly, monthly, seasonal, total).

Key Logic:

- **Time-Bounded Leaderboards:**
  - Each leaderboard has start_time and end_time (Unix timestamps). Matches are attributed to leaderboards where match_time BETWEEN start_time AND end_time.
- **Pre-Generated Future Leaderboards:**
  - System generates 30 DAILY leaderboards and 4 WEEKLY leaderboards in advance. A cronjob runs nightly to create tomorrow's leaderboards, ensuring they always exist when matches occur.
- **Permanent TOTAL Leaderboard:**
  - ID=1 has end_time set to year 2099, making it effectively permanent for all-time rankings.
- **Overlapping Periods:**
  - A single match on Nov 13, 2025, 14:30 matches FIVE leaderboards simultaneously:
    - **DAILY**: Nov 13, 2025 (00:00 - 23:59)
    - **WEEKLY**: Week 46, 2025 (Nov 11 - Nov 17)
    - **MONTHLY**: November 2025 (Nov 1 - Nov 30)
    - **SEASONAL**: Q4 2025 (Oct 1 - Dec 31)
    - **TOTAL**: All-time (1970 - 2099)

Cronjob Logic:

Every day at 00:01 UTC:

1. Check if DAILY leaderboard exists for today + 30 days
2. If not, INSERT new leaderboard_rule:
   - type = 'DAILY'
   - start_time = UNIX_TIMESTAMP(DATE_ADD(NOW(), INTERVAL 30 DAY))
   - end_time = start_time + 86399 (23h 59m 59s)
3. Check if WEEKLY leaderboard exists for this week + 4 weeks
4. If not, INSERT new WEEKLY leaderboard_rule
5. Cleanup: DELETE expired leaderboards older than 1 year

3.6.2.9 ranking

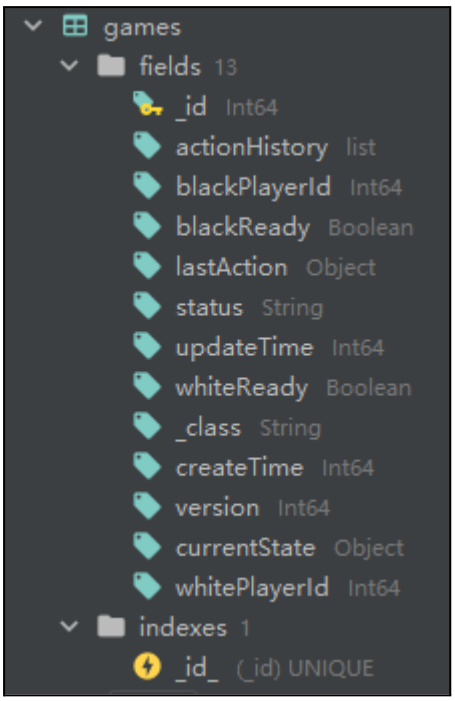
Purpose:

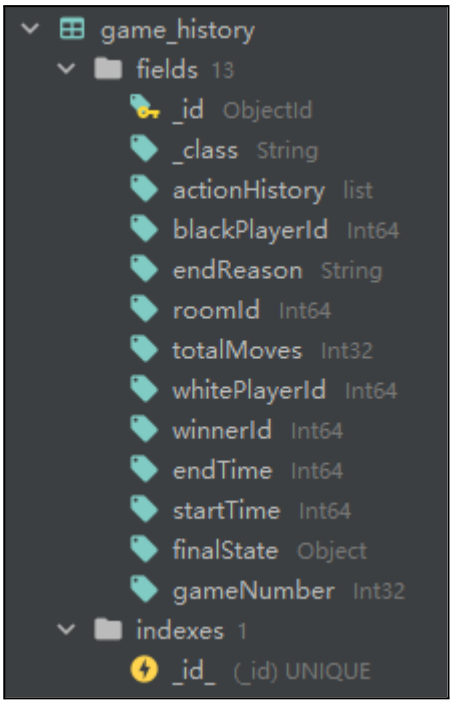
Denormalized snapshot of each player's ranking across all leaderboard time periods (optimized for fast leaderboard display).

Key Logic:

- **Denormalized Design:**
  - Instead of calculating rankings on-the-fly by aggregating score records (slow), this table caches the current state. One row per player per leaderboard = fast SELECT * WHERE leaderboard_rule_id=1 ORDER BY current_total_score DESC LIMIT 100.
- **Dual Tracking:**
  - total_exp + level_id: Experience-based progression (levels 1-10), gained from ALL modes - current_total_score + rank_position: ELO-based ranking, ONLY from RANKED mode
- **Incremental Updates:**
  - After each match, UPDATE the existing ranking row (if exists) or INSERT new row. Add score_change to current_total_score, add exp_value to total_exp.
- **Rank Position Caching:**
  - The rank_position field (1st, 2nd, 3rd…) is recalculated periodically (every 5 minutes by cronjob) because real-time ranking is too expensive for 10,000+ players. This trades freshness for performance.

3.6.3 MongoDB Table Structure





3.6.3.1 games

Purpose:

Stores the live state of all active games (waiting, playing, or just finished, not yet archived).

Key Logic:

- **Room ID as Primary Key:**
  - Uses MySQL's game_room.id as MongoDB's _id. This creates a 1:1 relationship where one room has exactly one active game document.
- **Optimistic Locking:**
  - The version field prevents race conditions when two players move simultaneously.
  - Each update increments version and checks the old version: java
    - // Player 1 and Player 2 both try to move at same time
    - // Only one succeeds; the other gets "version mismatch" error and retries
    - updateQuery = { _id: roomId, version: 41 }
    - update = { $set: { currentState: newState }, $inc: { version: 1 } }
- **Proposal-Response Pattern:**
  - Draw, undo, and restart requests use a two-phase commit:
    - 1. Player A proposes → Set drawProposerColor = 'BLACK'
    - 2. Player B sees proposal in UI (polling detects non-null proposer)
    - 3. Player B agrees → Execute action, clear drawProposerColor
    - 4. Player B declines → Just clear drawProposerColor
- **Action History:**
  - Every move, undo, draw, surrender is appended to actionHistory[]. \
  - This enables:
    - **Undo**: Pop last N actions, recalculate board
    - **Replay**: Step through actions chronologically
    - **Cheat** detection: Verify move sequence validity
    - **Game** Reset: On restart, the current game is archived to game_history, then resetForNewGame():
      - Swaps player colors (fairness: loser gets black piece advantage next game) - Clears board and action history
      - Increments gameCount (1st game, 2nd game, 3rd game…)
      - Keeps same roomId and player IDs

3.6.3.2 game_history

Purpose:

Permanent archive of all completed games for statistics, replays, and record-keeping.

Key Logic:

- **Archival Trigger:**
  - When players click “Restart”, the current GameDocument is copied to GameHistoryDocument, then the original is reset.
  - Similarly, when both players leave a room, the game is archived before deletion.
- **Compound Index:**
  - (roomId, gameNumber) enables queries like “show me the 3rd game from room 12345” in O(log N) time.
  - The gameNumber comes from GameDocument.gameCount (1, 2, 3…).
- **Winner Determination:**
  - If finalState.winner = 1 (black won) → winnerId = blackPlayerId
  - If finalState.winner = 2 (white won) → winnerId = whitePlayerId
  - If finalState.winner = 0 (draw) → winnerId = null
  - If finalState.winner = -1 (ongoing/timeout) → winnerId = null
- **Replay Functionality:**
  - Frontend can fetch a historical game and “replay” it by iterating through actionHistory[], applying each move sequentially with animation delays.

Why Not Store in MySQL?

MongoDB Advantages:

- Flexible schema: Action history is variable-length array (10 moves vs 200 moves)
- Nested objects: GameStateSnapshot has nested board arrays
- Write performance: High-throughput inserts (1000+ games/day)
- Document size: No row size limits (MySQL has 65KB row limit)

MySQL would require:

- Separate game_history table + game_actions table (1:N join)
- JSON columns (less queryable)
- Row size limits (need chunking for long games)

4. DevSecOps and Development Lifecycle

4.1 Source Control Strategy

4.1.1 Repository Structure

The project maintains **two primary repositories** on GitLab:

1. **gomoku-frontend** (React PWA)
2. **gomuku-backend** (Spring Boot multi-module)

4.1.2 Branching Model

We adopted a three-tier Git Flow strategy:

main (production)

↑ merge from test (manual approval required)

test (staging)

↑ merge from development (manual approval required)

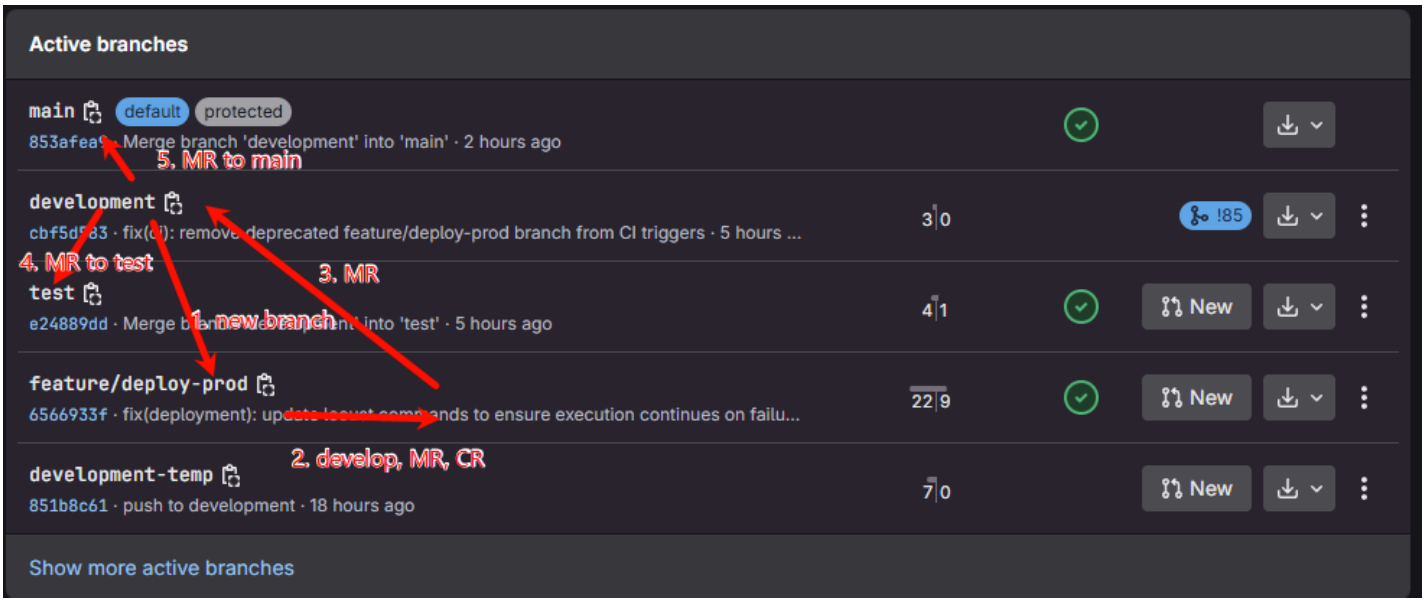
development

↑ merge from feature branches (code review required

feature/<module>/<feature>

↑ branch from development

4.1.3 Workflow



1. **Feature Development:**
    - Developer creates feature/<module>/<feature> from development
    - Implements feature, commits locally
    - Pushes to remote

2. **Code Review:**
    - Developer opens Merge Request (MR) to development
    - Team members review code changes
    - At least 1 approval required (configured in GitLab)

3. **Merge to Development:**
    - After approval, merge to development branch
    - Triggers full CI/CD pipeline:
    - Quality gates (lint, SAST, SCA, DAST)
    - Build & push Docker image (tag: test)
    - Deploy to Docker Swarm test environment
    - Performance testing (backend only)

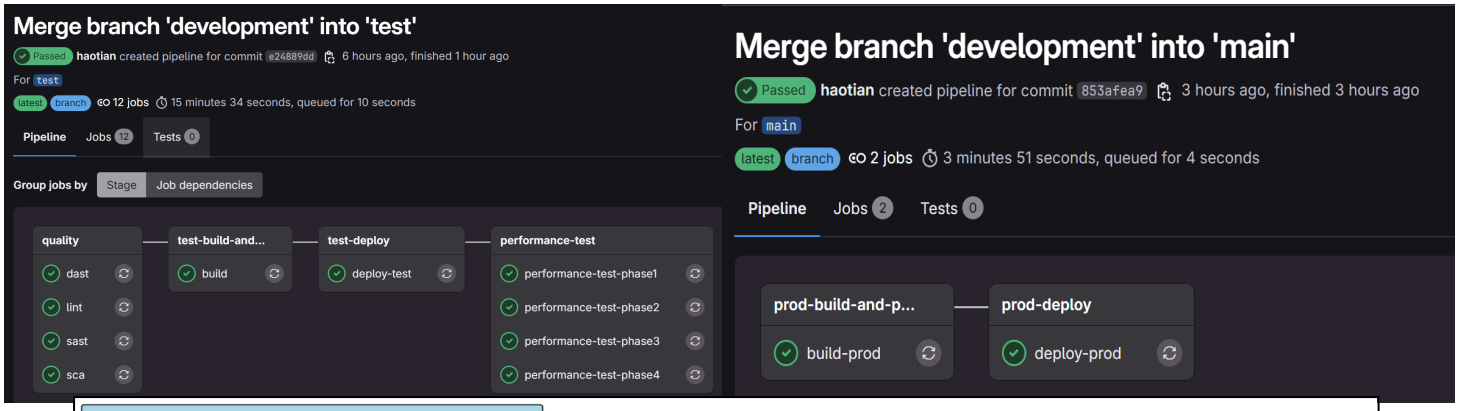
4. **Merge to Test:**
    - Merge to test branch manual
    - Triggers full CI/CD pipeline:
    - Quality gates (lint, SAST, SCA, DAST)
    - Build & push Docker image (tag: test)
    - Deploy to Docker Swarm test environment
    - Performance testing (backend only)

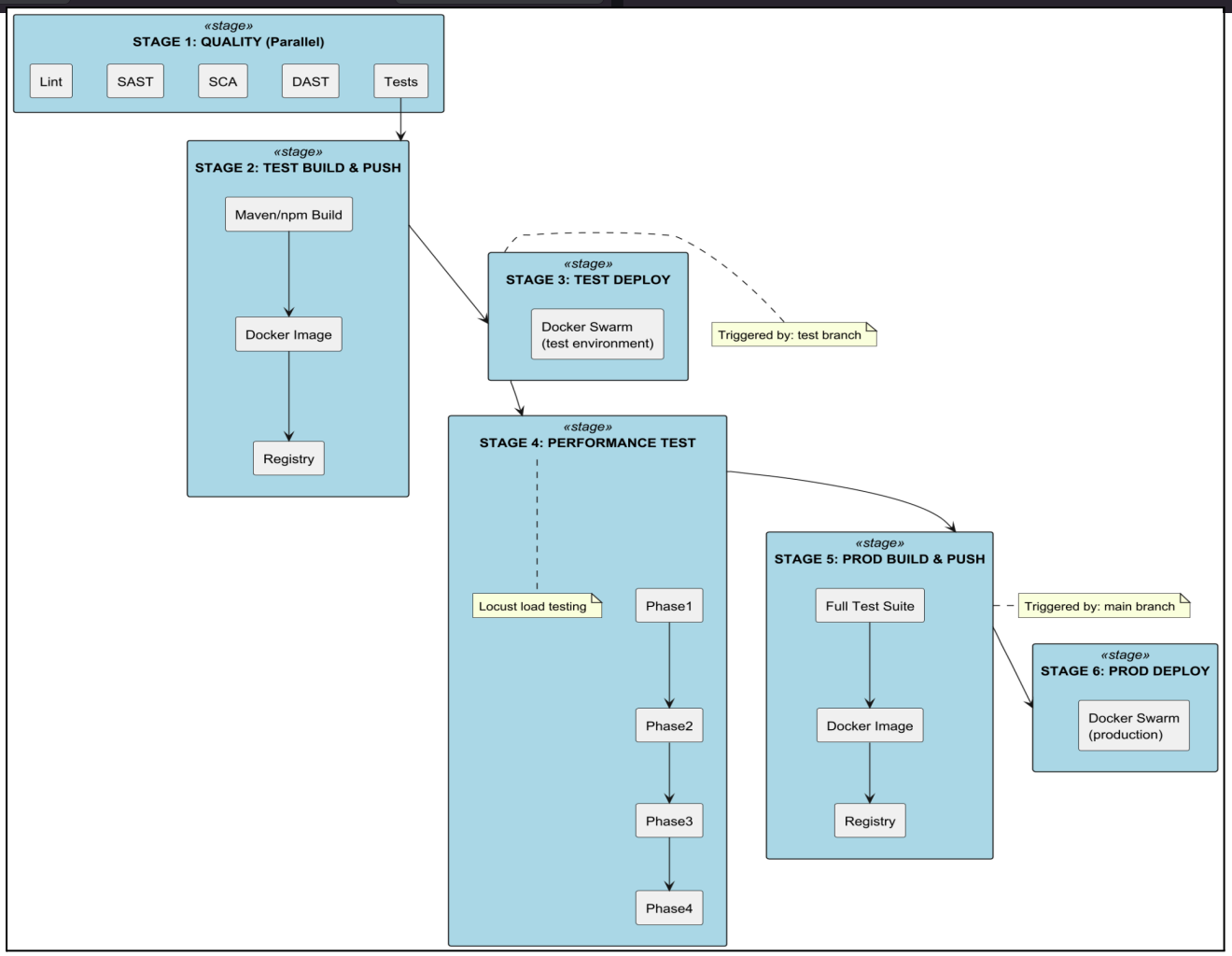
5. **Promotion to Production:**
    - After successful test deployment + manual UAT
    - Create MR from development to main
    - Triggers production pipeline:
    - Quality gates + build (tag: prod)
    - Deploy to production Docker Swarm
    - No performance testing (already validated in test)

4.2 Continuous Integration

4.2.1 Pipeline Architecture

Our project implements a **6-stage GitLab CI/CD** pipeline that automates the complete delivery lifecycle from code commit to production deployment.





Branch Triggers: - test branch → Stages 1-4 (continuous testing) - main branch → Stages 5-6 (production release)

4.2.2 Stage 1: Quality Gates (Parallel Execution)

Branch Triggers: - test branch → Stages 1-4 (continuous testing) - main branch → Stages 5-6 (production release)

Check	Tool	Purpose
Lint	ESLint, PMD	Code style and best practices
Unit Tests	Jest, JUnit	Functional correctness
SAST	ESLint Security, SpotBugs	Static security analysis
SCA	audit-ci, OWASP Dependency-Check	Dependency vulnerabilities
DAST	OWASP ZAP	Dynamic security testing

4.2.3 Stage 2: Build & Push

Frontend: React application compiled into optimized production bundle, packaged in nginx-based Docker image

Backend: Maven multi-module build producing 6 microservice JARs, each containerized with JRE-only base images

All images pushed to private registry at 152.42.207.145:5000

4.2.4 Stage 3: Test Deploy

Deployment to test environment using Docker Swarm orchestration:

- Zero-downtime rolling updates
- Health check polling (60s max)
- Service verification and status checks

Test Environment:

- API: https://test-api-gomoku.goodyhao.me
- Frontend: https://test-gomoku.goodyhao.me

4.2.5 Stage 4: Performance Testing (Backend Only)

Sequential Locust load tests validate system performance under realistic load:

Phase	Scenario	Load	Duration	Key Metric
Phase 1	User Module	100 users	20s	Registration/login latency
Phase 2	Matching & Room	200 users	1m	Room creation time
Phase 3	Full Game	100 users	1m	Move latency <200ms
Phase 4	Leaderboard	500 users	20s	Query response time

4.2.6 Stages 5-6: Production Deployment

Stage 5 - Production Build:

- Full test suite execution (vs skipped in test build)
- Production-optimized compilation
- Tagged as prod images

Stage 6 - Production Deploy:

- Deploy to production Swarm cluster
- Zero-downtime rolling updates
- Health check validation before routing traffic

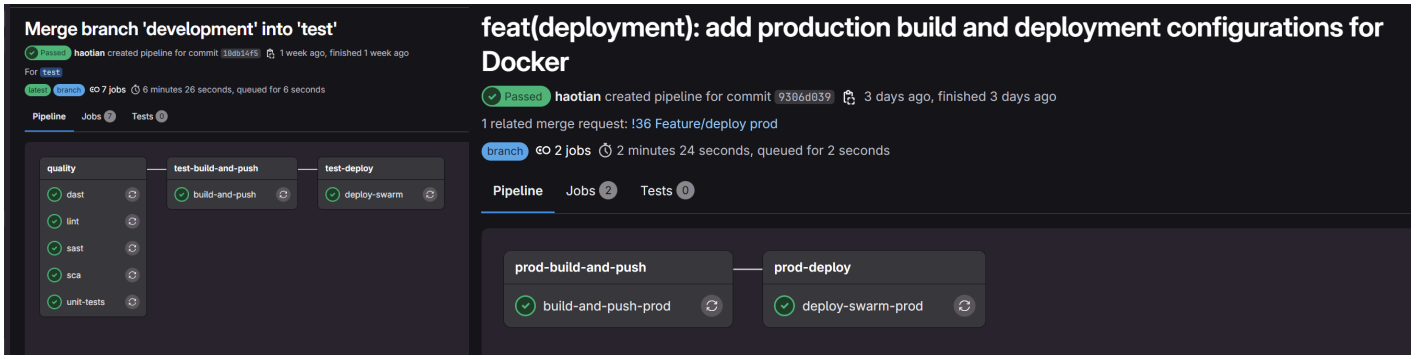
Production Environment:

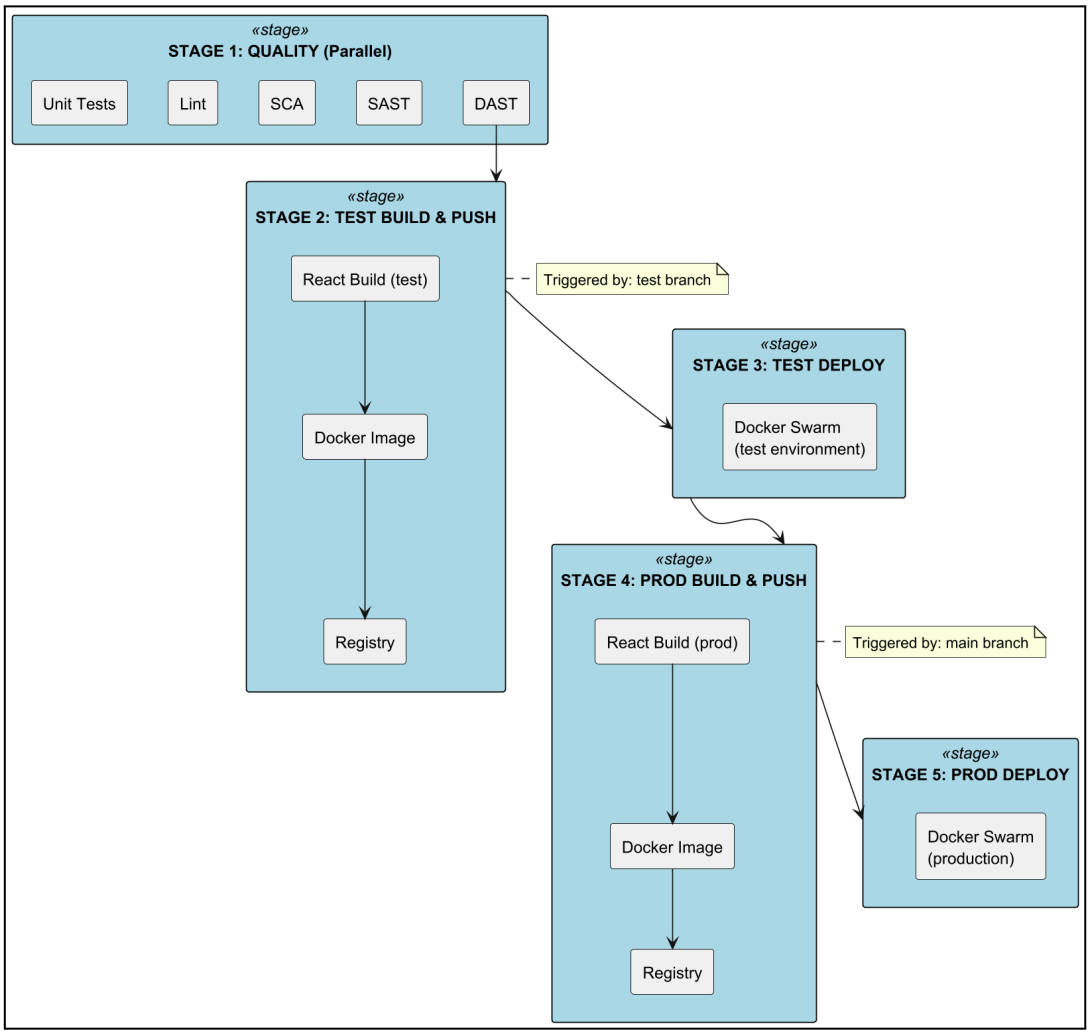
- API: https://api-gomoku.goodyhao.me
- Frontend: https://gomoku.goodyhao.me

4.3 Continuous Delivery

4.3.1 Pipeline Architecture

The frontend implements a **5-stage GitLab CI/CD pipeline** for the React Progressive Web App, automating quality checks, Docker image builds, and deployments to test and production environments.





Branch Triggers: - test branch → Stages 1-3 (continuous testing) - main branch → Stages 4-5 (production release)

4.3.2 Stage 1: Quality Gates

All checks run in parallel on Node.js 18 containers:

Check	Tool	Purpose
Unit Tests	Jest + React Testing Library	Functional testing, >75% coverage
Lint	ESLint (Airbnb config)	Code style, React best practices
SCA	audit-ci	Dependency vulnerability scanning
SAST	ESLint Security Plugin	Static security analysis
DAST	OWASP ZAP	Dynamic testing against live deployment

4.3.3 Stage 2: Test Build & Push

React application built with test environment configuration:

- Webpack compilation with code splitting and tree shaking
- Optimized production bundle
- Docker image with nginx-alpine base
- Pushed to private registry with test tag

4.3.4 Stage 3: Test Deploy

Deployment to test environment:

- Docker Swarm stack deployment
- Health check polling with 60-second timeout
- Service status verification

Test Environment: https://test-gomoku.goodyhao.me

4.3.5 Stage 4: Production Build & Push

Same build process as Stage 2, with production environment configuration:

- Production API endpoints
- Image tagged as prod
- Only triggered on main branch

4.3.6 Stage 5: Production Deploy

Deployment to production Swarm cluster:

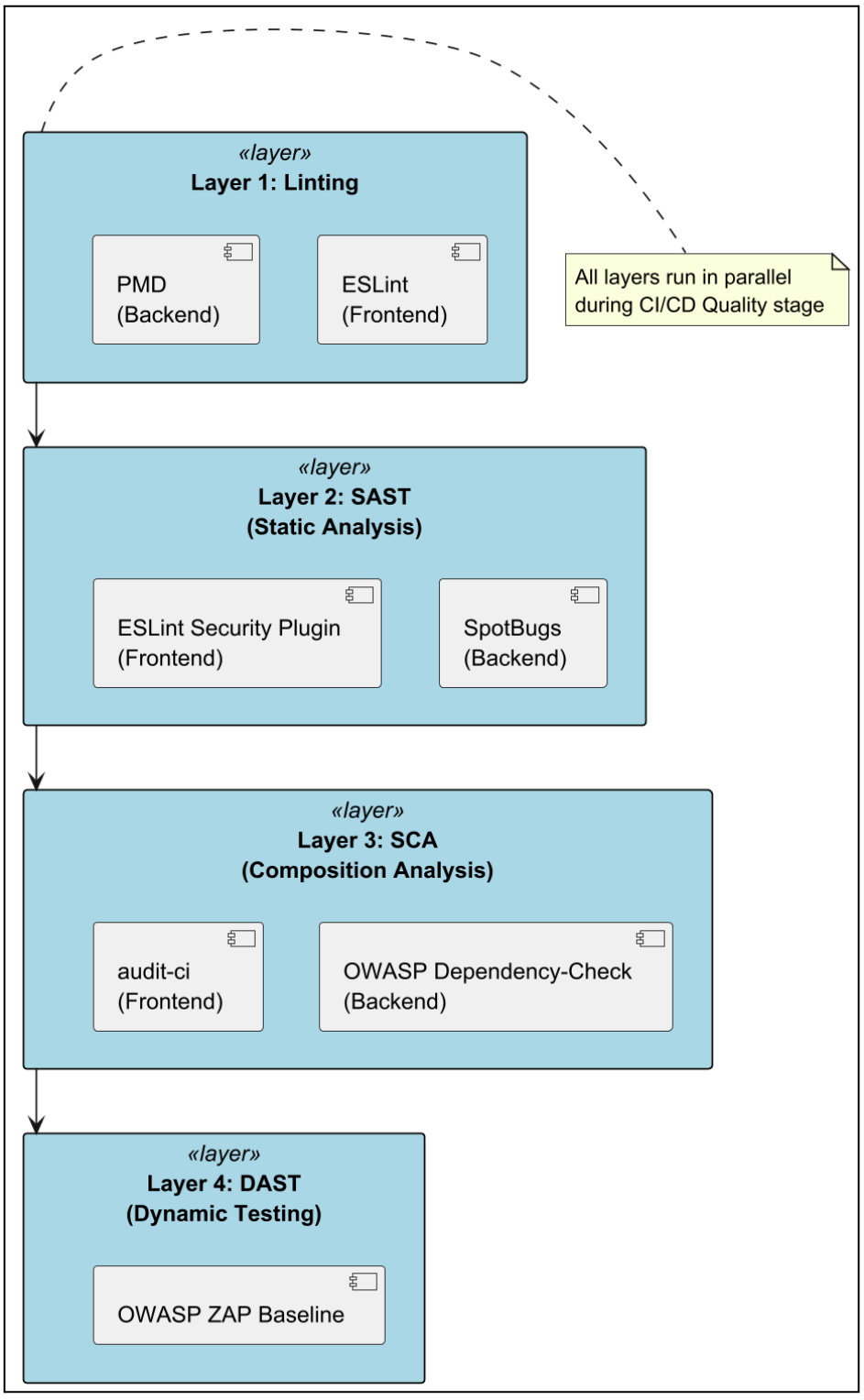
- Stack name: gomoku-frontend-stack-prod
- Zero-downtime rolling updates
- Service verification

Production Environment: https://gomoku.goodyhao.me

4.4 Security and Compliance within DevSecOps

4.4.1 Overview

The Gomoku platform embeds **security scanning at every stage** of the CI/CD pipeline, following the **shift-left security** principle to detect and remediate vulnerabilities early in the development lifecycle.



4.4.2 Layer 1: Code Quality & Linting

Purpose: Enforce code style, best practices, and catch common bugs

Tools:

- Frontend: ESLint with Airbnb configuration
- Backend: PMD for Java code analysis

Detection: Code style violations, complexity issues, anti-patterns

4.4.3 Layer 2: SAST (Static Application Security Testing)

Purpose: Analyze source code for security vulnerabilities without execution

Tools:

- Backend: SpotBugs
- Frontend: ESLint Security Plugin

Detection: SQL injection, XSS, insecure crypto, null pointer errors, resource leaks

4.4.4 Layer 3: SCA (Software Composition Analysis)

Purpose: Identify known vulnerabilities in third-party dependencies

Tools:

- Backend: OWASP Dependency-Check with NVD integration
- Frontend: audit-ci

Detection: Known CVEs with CVSS scoring, outdated dependencies

4.4.5 Layer 4: DAST (Dynamic Application Security Testing)

Purpose: Test running application for runtime vulnerabilities

Tool: OWASP ZAP 2.14.0 Baseline Scan

Detection: OWASP Top 10, XSS, SQL injection, CSRF, insecure headers

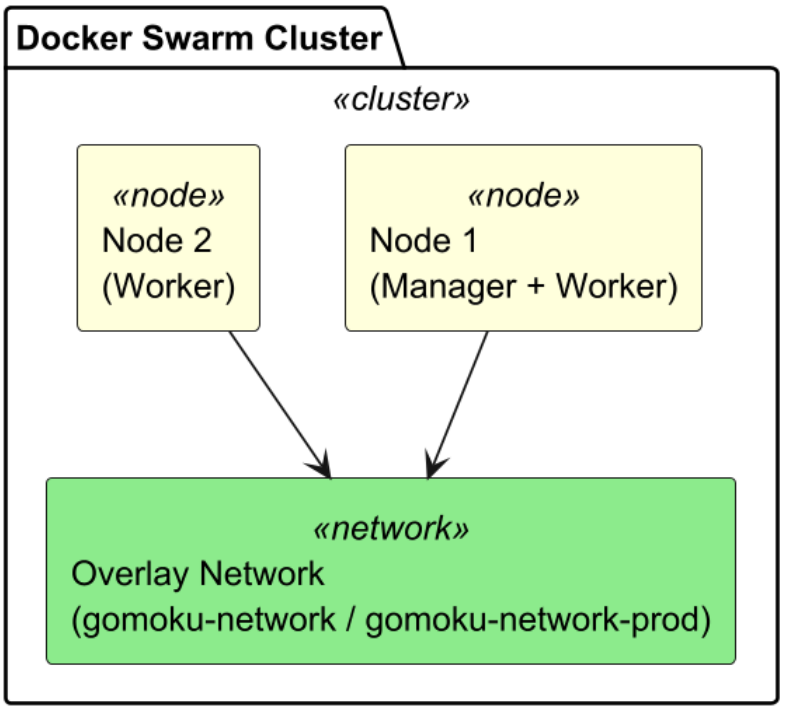
Approach: Scans run in isolated Docker networks against live test deployments

4.5 Security and Compliance within DevSecOps

4.5.1 Overview

The Gomoku platform is deployed using **Docker Swarm** orchestration across **two physical nodes**, with separate clusters for test and production environments. Each environment uses Swarm's **overlay networking** for service discovery and load balancing.

4.5.2 Physical Infrastructure



Node Roles:

- **Node 1**: Manager + Worker (orchestrates and runs services)
- **Node 2**: Worker (runs services)

4.5.3 Environment Separation

4.5.3.1 Test Environment

Network: gomoku-network

Stack Name: gomoku-stack-test

Deployment Mode: Global (one instance per node)

Service	Port	Instances	Distribution
Gomoku Service	8080	2	Node 1 + Node 2
User Service	8081	2	Node 1 + Node 2
Ranking Service	8082	2	Node 1 + Node 2
Gateway	8083	2	Node 1 + Node 2
Match Service	8084	2	Node 1 + Node 2
Room Service	8085	2	Node 1 + Node 2
TURN Server	3478	2	Node 1 + Node 2

URL: https://test-api-gomoku.goodyhao.me → Gateway:8083

4.5.3.2 Prod Environment

Network: gomoku-network-prod

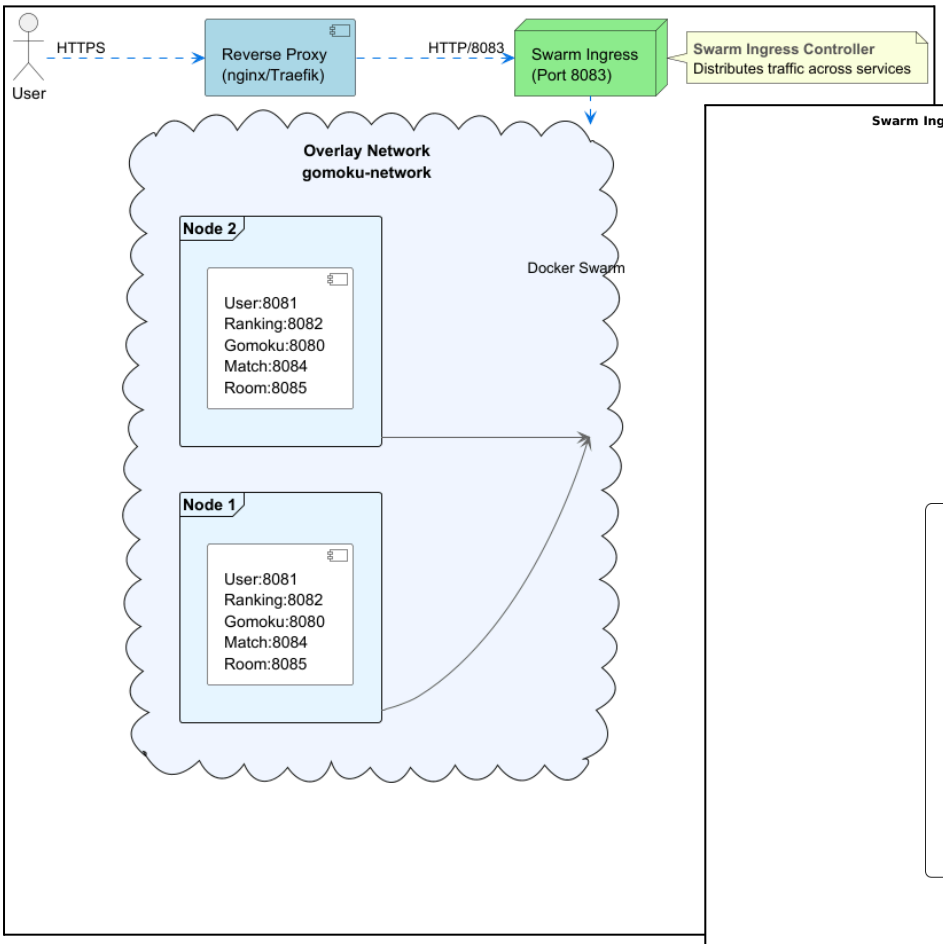
Stack Name: gomoku-stack-prod

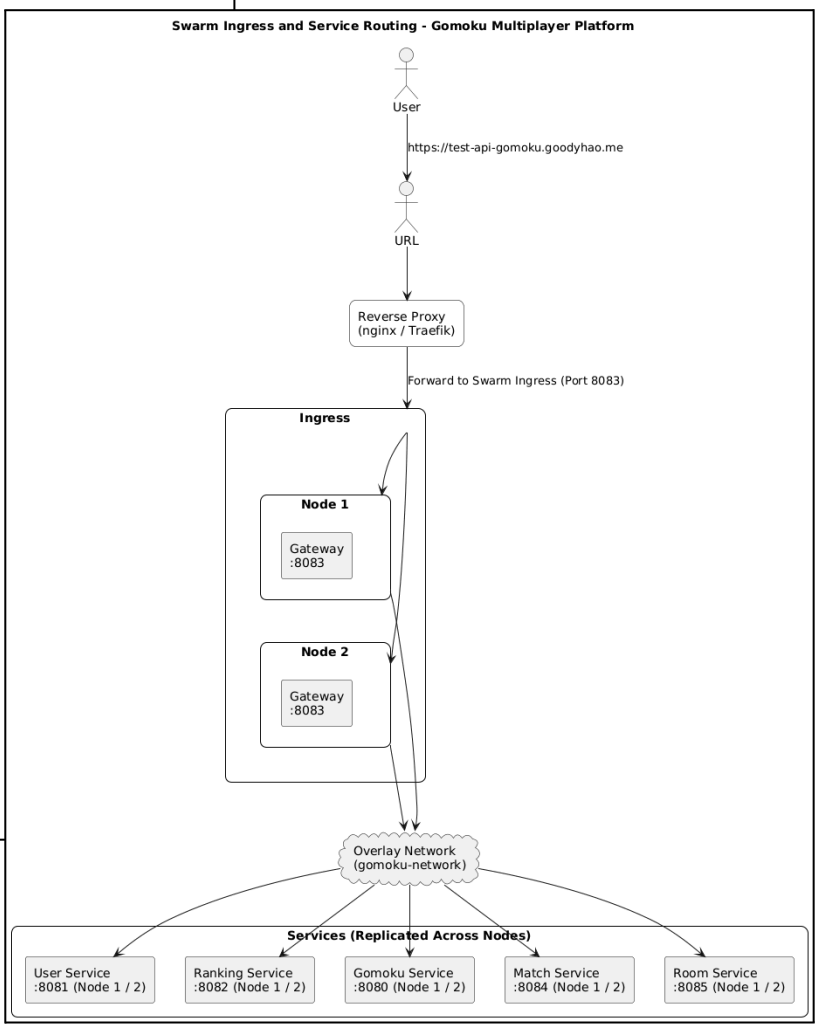
Deployment Mode: Global (one instance per node)

Service	Port	Instances	Distribution
Gomoku Service	8090	2	Node 1 + Node 2
User Service	8091	2	Node 1 + Node 2
Ranking Service	8092	2	Node 1 + Node 2
Gateway	8093	2	Node 1 + Node 2
Match Service	8094	2	Node 1 + Node 2
Room Service	8095	2	Node 1 + Node 2

URL: https://api-gomoku.goodyhao.me → Gateway:8093

4.5.3.3 Request Flow with Load Balancing





Load Balancing:

- Swarm's built-in ingress routing mesh distributes requests
- Each service has 2 instances (one per node)
- Round-robin distribution across healthy instances

4.5.3.4 Deployment Mode: Global

Key Characteristic: deploy.mode: global

Behavior:

- Exactly one instance per node in the Swarm cluster
- New nodes automatically get all global services
- Ensures every node has local access to services

Benefits:

- High Availability: If one node fails, other node continues serving
- Load Distribution: Traffic automatically spreads across nodes
- Low Latency: Local service-to-service communication within same node

Trade-offs:

- Fixed scaling (can’t exceed node count)
- All nodes must have sufficient resources

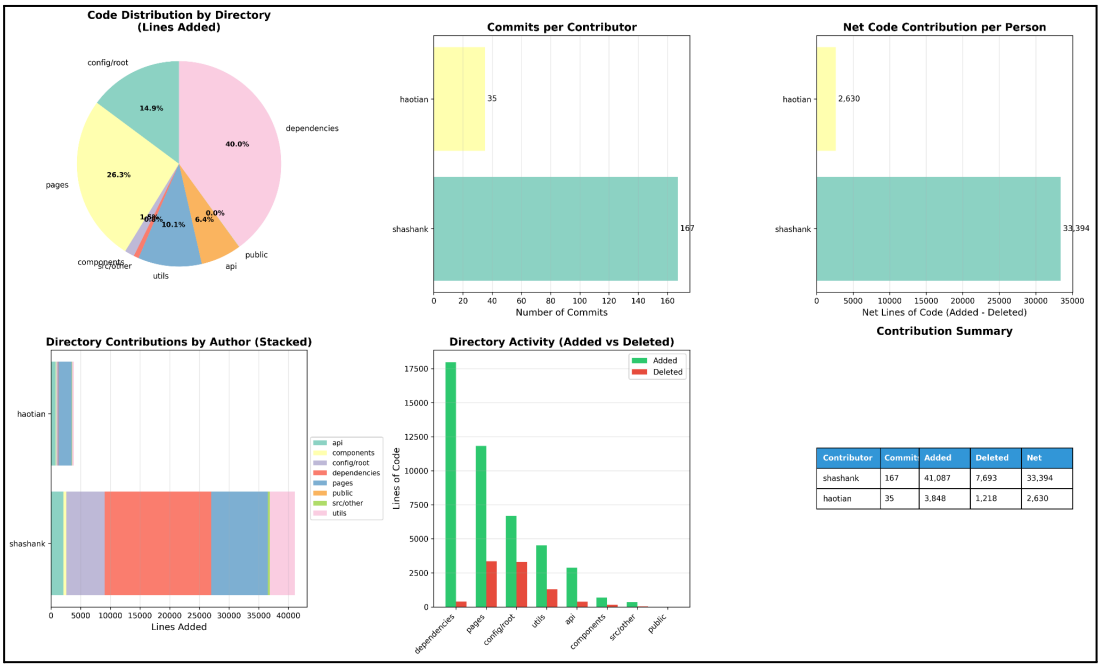
5. INDIVIDUAL MEMBERS ACTIVITY CONTRIBUTION SUMMARY

GitLab Contributions

Backend:

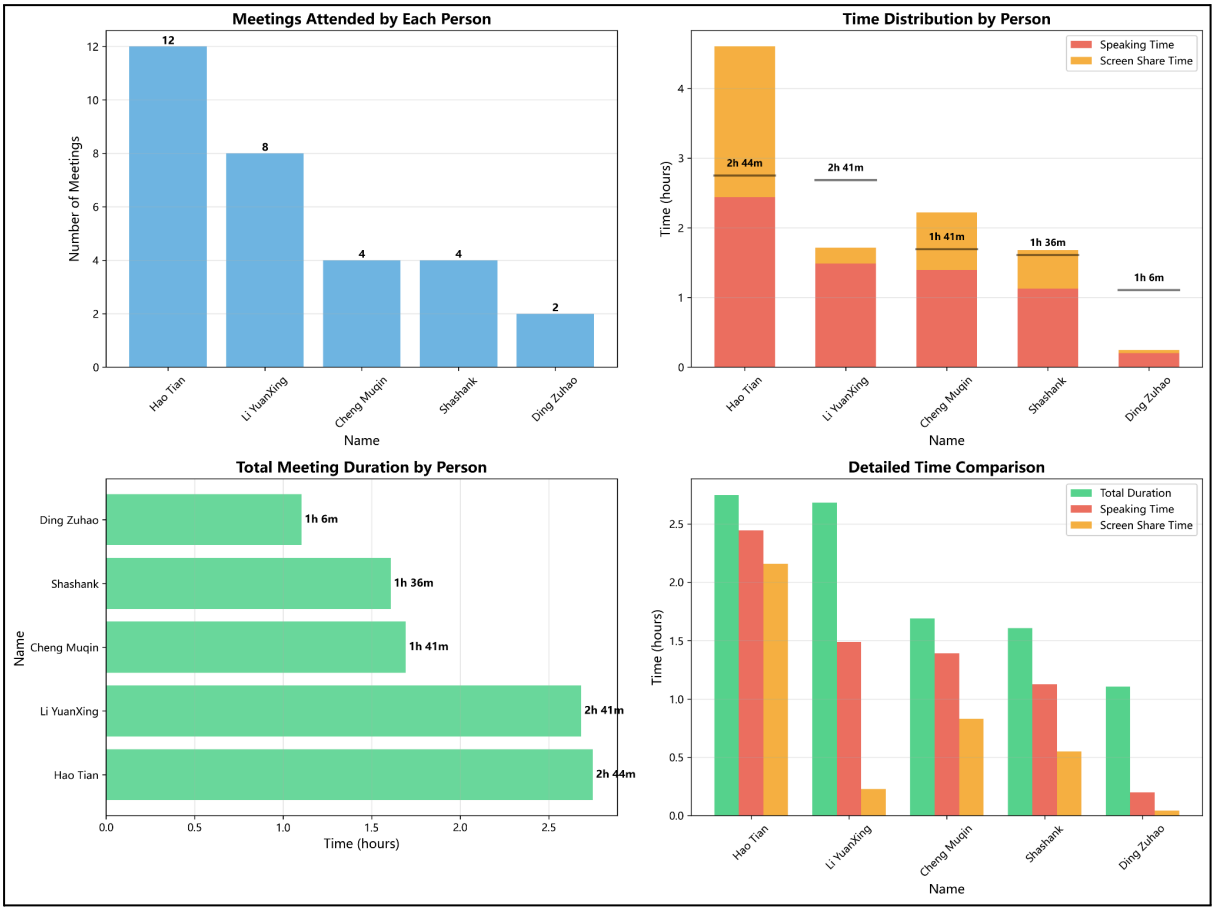
Contributor	Commits	Added	Deleted	Net
haotian	205	65,666	13,102	52,564
shashank	70	1,656	568	1,088
liyuanxing	50	3,004	366	2,638
dingzuhao	9	1,407	4	1,403
chengmuqin	7	11,100	872	10,228

Frontend:

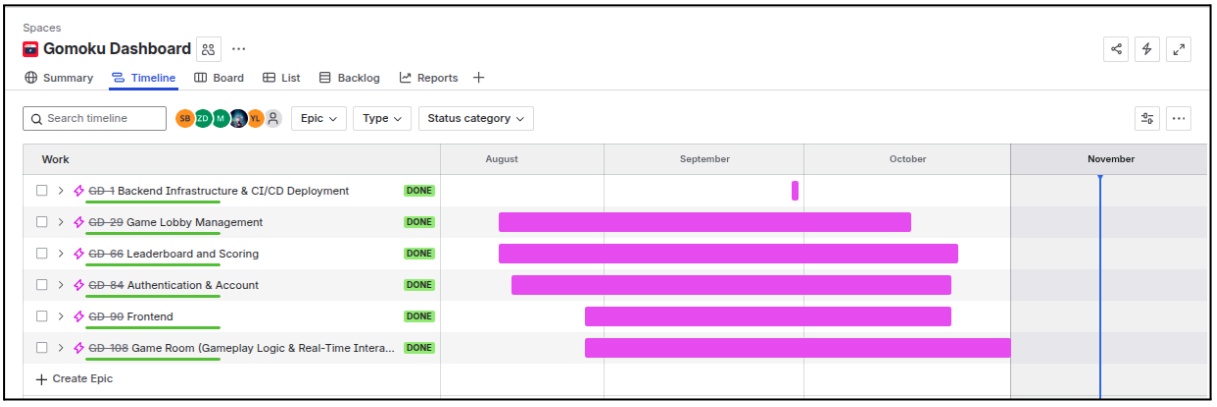


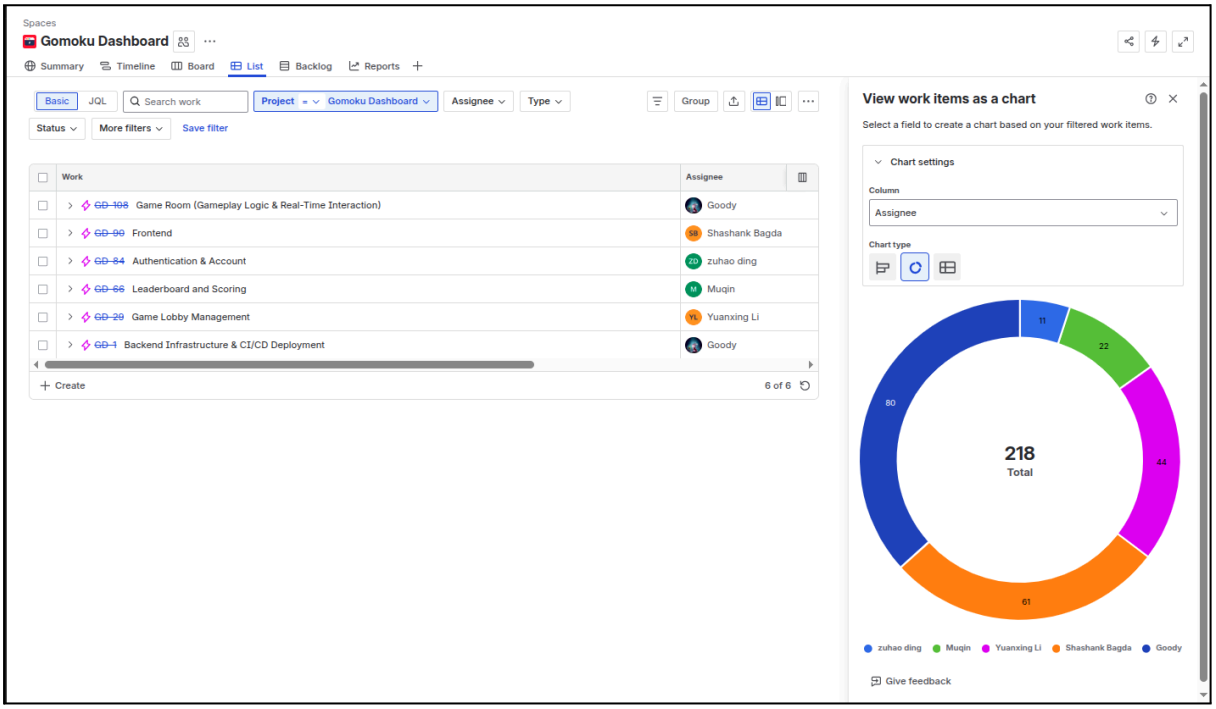
Contributor	Commits	Added	Deleted	Net
shashank	167	41,087	7,693	33,394
haotian	35	3,848	1,218	2,630

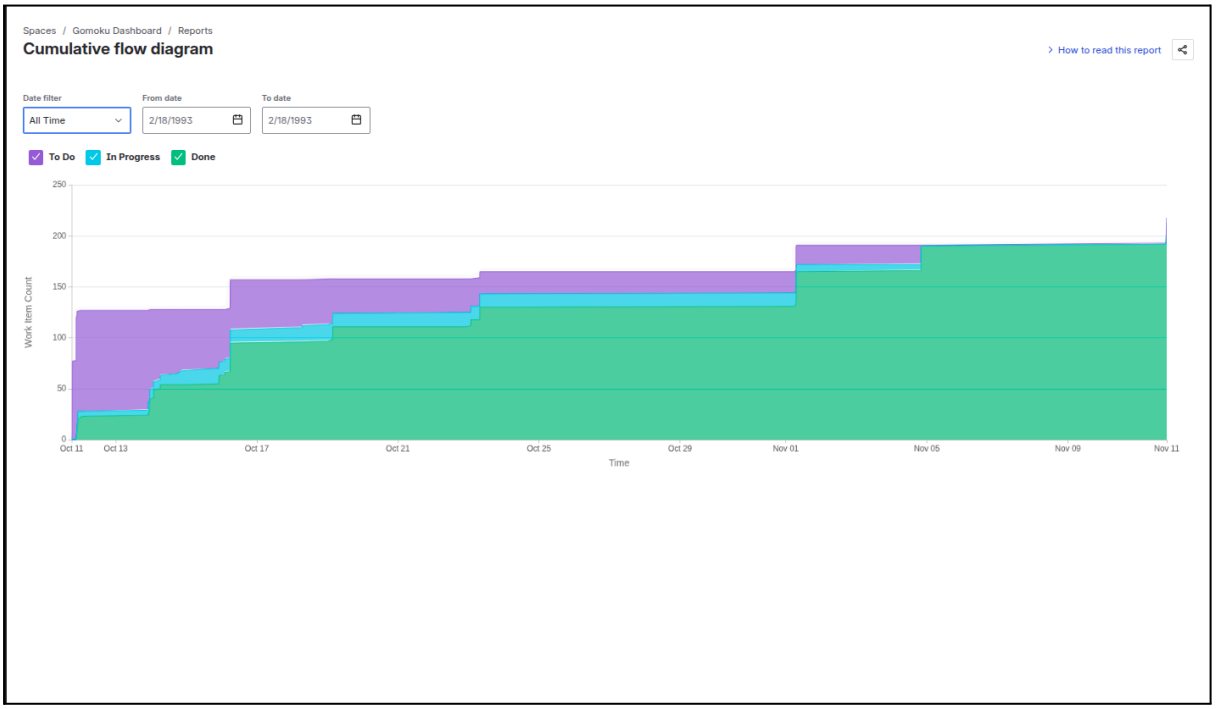
Discord:



Jira:




Spaces
Gomoku Dashboard
Summary
Timeline
Board
List
Backlog
Reports
Basic
JQL
Search work
Project = Gomoku Dashboard
Assignee
Type
Group
Status
More filters
Save filter
Work
Assignee
GD-108 Game Room (Gameplay Logic & Real-Time Interaction) Goody
GD-99 Frontend Shashank Bagda
GD-84 Authentication & Account zuhao ding
GD-66 Leaderboard and Scoring Muqin
GD-29 Game Lobby Management Yuanxing Li
GD-1 Backend Infrastructure & CI/CD Deployment Goody
Create
6 of 6
View work items as a chart
Select a field to create a chart based on your filtered work items.
Chart settings
Column
Assignee
Chart type
11
22
44
61
80
218
Total
zuhao ding
Muqin
Yuanxing Li
Shashank Bagda
Goody
Give feedback


Spaces / Gomoku Dashboard / Reports
Cumulative flow diagram
How to read this report
Date filter
All Time
From date
2/18/1993
To date
2/18/1993
To Do
In Progress
Done
Work Item Count
250
200
150
100
50
0
Oct 11
Oct 13
Oct 17
Oct 21
Oct 25
Oct 29
Nov 01
Nov 05
Nov 09
Nov 11
Time

5.1 Shashank Bagda — Frontend Lead & AI Developer & Test Engineer

Scope & Impact

Shashank owned the end-to-end player experience (React PWA + Tailwind) and the AI practice module. He also set up front-end tests, participated in game room logic on the backend, and partnered on quality (SAST/SCA, DAST, and UX performance).

Key Contributions (technical)

1. React PWA & UI System
    a. Built responsive board and room layouts; stabilized mobile interactions; implemented PWA install flows and cache-busting.
    b. State handling for turn indicators, move animation, and error toasts; guarded UI against invalid board clicks and stale versions.
2. Game Room UX & Real-time
    a. Implemented ready/undo/draw/restart proposal UX; history timeline; last-move highlight with animation for readability.
    b. Integrated LiveKit client for optional voice in rooms; UI affordances for push-to-talk and connection state.
3. AI Practice (Hybrid Strategy)
    a. Designed Strategy + Factory selection by K-value (Easy/Medium/Hard) and wrote heuristic evaluator for candidate scoring (open-three, four-threat, center bias, defense multipliers).
    b. Added OpenAI adapter with bounded tokens and circuit-breaker fallback to heuristics; exposed confidence surface in logs for tuning.
4. Testing & Quality
    a. Jest + RTL tests for reducers/components; integration shims for Practice page.
    b. Drove ESLint (Airbnb) + security plugin adoption; resolved SCA advisories on FE dependencies.
    c. Helped craft DAST target endpoints and verify CORS/security headers post-gateway.
5. Performance & Observability
    a. Instrumented paint/interaction timing via web vitals logging to console (and optionally to a metrics endpoint).
    b. Worked with backend to expose AI latency and game move latency panels in Grafana; used data to tweak polling interval and batch UI updates.

Representative Artifacts

- src/pages/Practice/* — AI practice flows, K-value selector, board binding.
- src/components/Board/* — board grid, event handling, animations.
- src/services/api.ts — typed API client, JWT header injection, exponential backoff.
- Backend (co-authored): AI controller signatures, DTOs for board serialization, and adapter wrapper.

Measurable Outcomes

- p50 move latency on FE path stayed within <200 ms budget end-to-end under load simulations.
- Accessibility: improved contrast & keyboard focus on board cells; reduced layout shift after first render.
- Reliability: AI fallback path handled OpenAI slowness without visible UX stalls.

5.2 Hao Tian — Backend Lead & DevOps Engineer & Test Engineer

Area	Contribution	Key Artifacts
Backend Microservice Architecture	Designed the overall Spring Boot service structure with modular Maven projects (user-service, game-service, etc.).	Design, User-service,
Gomoku-service	Implemented real-time move updates using Strategy Pattern.	ExecuteChain.java, ExecuteChainHandler.java, ValidateChain.java, ValidateChainHandler.java
User-Service	Implemented User Register/Login/Logout/Verify/Email(By Liyuanxing)/Reset/User-Profile	
CI/CD Automation	Authored complete GitLab CI/CD pipeline, integrating build, test, SonarQube, and deploy jobs.	.gitlab-ci.yml
Deployment Infrastructure	Deployed production environment on DigitalOcean Droplet using Docker Warm + NGINX.	docker-compose-{}.yml docker-swarm-deploy-{}.yml
Performance Test	Use Locust, write testing scripts and deploy to CI/CD	phase1_user_module.py, phase2_matching_room.py, phase3_full_game.py, phase4_leaderboard.py

Impact:

Led backend architecture and DevOps for a real-time Gomoku gaming platform. Designed scalable Spring Boot microservices with modular Maven structure, enabling clean separation of concerns. Engineered real-time move updates using Strategy Pattern for maintainable game logic. Automated CI/CD pipeline with GitLab, integrating SonarQube and Locust-based performance testing. Deployed production-ready infrastructure on Docker Swarm with NGINX, ensuring high availability and streamlined deployment.

5.3 Cheng Muqin — Leaderboard Service Developer

Area	Contribution	Key Artifacts
Match Settlement System	Designed and implemented idempotent match settlement mechanism with multi-leaderboard reward distribution across RANKED/CASUAL/PRIVATE modes	• settleMatch() with idempotency check • Score & experience calculation logic • Transaction-based multi-leaderboard updates
Multi-Period Leaderboard	Designed unified data model supporting 5 concurrent leaderboard types (DAILY/WEEKLY/MONTHLY/SEASONAL/TOTAL) with timestamp-based activation	• leaderboard_rule table with start/end time • Active rule query by current timestamp • On-demand ranking calculation
Progressive Level System	Implemented 6-tier level progression (Bronze to Master) with experience-based automatic advancement	• level table with exp thresholds • calculateLevelByExp() algorithm • level_rule table for mode-specific exp rewards
Flattened Score Rules	Designed flattened schema storing all 9 score configurations (3 modes × 3 results) in single table to avoid excessive joins	• score_rule table with 9 columns • getScoreValue() method with mode/result mapping • Simplified query performance

Impact:

Delivered a transaction-safe ranking system handling 1500+ players across 5 leaderboard types with zero race conditions. The unified period model eliminates duplicate code, while flattened score rules reduce query complexity. Match settlement achieves full idempotency through database-level duplicate checks.

5.4 Li YuanXing — Backend Developer

Area	Contribution	Key Artifacts
Room-Service	Implemented the services of the game room using redis, such as leaving, creating, joining and so on.	RoomCodeServiceImpl.java
Match-service	Implemented the logic of matching using redis.	MatchServiceImpl.java
Prometheus and Grafana	Develop and deploy the prometheus and grafana.	The screenshot of the monitoring metrics

Impact:

By implementing the Room-Service, Gomoku project can have different rooms of use while managing the room well. By implementing the Matching Service, Gomoku project can make users in the ranking queue join the ranking games by good matching logic. After developing and deploying the prometheus and grafana, we can monitor the Goomku in lots of aspects, such as memory use, CPU and so on.

5.5 Ding Zuhao — Performance Engineer & Quality Analyst

Area	Contribution	Key Artifacts
JMeter Load Testing(not adopted)	JMeter Load Testing, Developed test plan simulating 50 concurrent matches with 1000 requests/min.	/testing/jmeter/auto_login_match_cancel_logout.jmx (branchdingzuhao-learning)

Impact: Learning JMeter

6. Appendix

6.1 Appendix A1 – Project Structure Trees

6.1.1 A1.1 Frontend Project Structure (React + TailwindCSS)

```
gomoku-frontend/
├── public/                 # Static assets
│   ├── manifest.json               # PWA manifest
│   └── service-worker.js           # Offline support
├── src/
│   ├── components/                 # React components
│   │   ├── Board/          # Game board UI
│   │   ├── Room/           # Room management
│   │   ├── Leaderboard/            # Rankings display
│   │   └── Auth/           # Login/Register
│   ├── pages/              # Page-level components
│   │   ├── GamePage.jsx
│   │   ├── LobbyPage.jsx
│   │   └── ProfilePage.jsx
│   ├── services/           # API integration
│   │   ├── api.js          # REST API client
│   │   ├── websocket.js            # WebSocket client
│   │   └── livekit.js      # Voice chat
│   ├── store/              # State management
│   │   ├── gameSlice.js
│   │   └── userSlice.js
│   └── utils/              # Utilities
├── tailwind.config.js      # TailwindCSS config
└── package.json                    # Dependencies
```

Key Technologies:

- React 19.2.0 with Hooks
- TailwindCSS 3.4.10 for styling
- Redux Toolkit for state management
- Axios for HTTP requests
- STOMP.js for WebSocket
- LiveKit Client for WebRTC voice

6.1.2 A1.2 Backend Project Structure (Spring Boot + Microservices)

```
gomuku-backend/
├── common/                    # Shared utilities
│   ├── validation/            # BasicCheck, ValueChecker frameworks
│   └── exception/             # Global exception handling
├── dao/                       # Data access layer
│   ├── entity/                # MyBatis entities
│   └── mapper/                # MyBatis mappers
├── gomoku/                    # Game module
│   ├── gomoku-biz/                  # Game logic
│   │   └── chain/             # Chain of Responsibility
│   └── gomoku-controller/           # REST controllers
├── user/                      # User module
│   ├── user-biz/
│   └── user-controller/
├── matching/                  # Matchmaking module
├── room/                      # Room management module
├── ranking/                   # Leaderboard module
├── gateway/                   # API Gateway
└── pom.xml                    # Maven dependencies
```

Key Technologies:

- Spring Boot 3.5.5
- MyBatis 3.0.5 for ORM
- Spring WebSocket (STOMP)
- Spring Cloud Gateway
- Spring AOP for validation

6.1.3 A1.3 Support and Infrastructure

```
Infrastructure/
├── docker/
│   ├── docker-compose-test.yml
│   ├── docker-compose-prod.yml
│   ├── docker-swarm-deploy-test.yml
│   └── docker-swarm-deploy-prod.yml
├── gitlab-ci/
│   ├── .gitlab-ci.yml (frontend)
│   └── .gitlab-ci.yml (backend)
├── nginx/
│   └── nginx.conf          # Reverse proxy config
└── databases/
    ├── mysql/              # Schema definitions
    ├── mongodb/            # Game state collections
    └── redis/              # Cache & session config
```

6.2 Appendix A2 – API Reference (REST, WebSocket, AI)

The Gomoku platform exposes both **synchronous REST APIs and asynchronous WebSocket events**, complemented by **AI computation endpoints** integrated within the AIEngine module.

All REST APIs conform to:

Base URL: https://api.gomoku.goodyhao.me/api/v1

Format: JSON

Auth: JWT token via Authorization: Bearer <token>

HTTP Codes:

200 OK

400 Bad Request

401 Unauthorized

404 Not Found

500 Internal Server Error

6.2.1 A2.1 REST API Endpoints

Base URL: https://api-gomoku.goodyhao.me/api/v1

Authentication: JWT Bearer token in Authorization header

6.2.1.1 User Service

Method	Endpoint	Description	Request Body	Response
POST	/auth/register	User registration	{username, password, email}	{token, userId}
POST	/auth/login	User login	{username, password}	{token, userId}
GET	/users/{id}	Get user profile	-	{id, username, wins, losses}
PUT	/users/{id}	Update profile	{nickname, avatar}	{success: true}

6.2.1.2 Room Service

Method	Endpoint	Description	Request Body	Response
POST	/rooms	Create room	{name, isPrivate}	{roomId, code}
GET	/rooms	List rooms	-	[{roomId, name, players}]
POST	/rooms/{id}/join	Join room	-	{success: true}
DELETE	/rooms/{id}	Delete room	-	{success: true}

6.2.1.3 Matching Service

Method	Endpoint	Description	Request Body	Response
POST	/match/queue	Join matchmaking	-	{queueId}
DELETE	/match/queue	Leave queue	-	{success: true}
GET	/match/status	Check match status	-	{matched: false, estimated: 30}

6.2.1.4 Game Service

Method	Endpoint	Description	Request Body	Response
GET	/games/{id}	Get game state	-	{board, currentPlayer, moves}
POST	/games/{id}/move	Make move	{x, y}	{valid: true, winner: null}
POST	/games/{id}/resign	Resign game	-	{success: true}

6.2.1.5 Ranking Service

Method	Endpoint	Description	Request Body	Response
GET	/rankings	Get leaderboard	?page=1&size=20	[{rank, username, score, wins}]
GET	/rankings/user/{id}	Get user rank	-	{rank, score, percentile}

6.2.2 A2.2 WebSocket Events

Connection: wss://api-gomoku.goodyhao.me/ws

Protocol: STOMP over WebSocket

Client → Server

Destination	Payload	Description
/app/game/{id}/move	{x, y}	Send move
/app/room/{id}/chat	{message}	Send chat message
/app/match/cancel	-	Cancel matchmaking

Server → Client

Topic	Payload	Description
/topic/game/{id}/state	{board, currentPlayer}	Game state update
/topic/game/{id}/move	{x, y, player}	Opponent move
/topic/game/{id}/result	{winner, reason}	Game ended
/topic/room/{id}/chat	{user, message, timestamp}	Chat message
/topic/match/found	{gameId, opponent}	Match found

6.3 Appendix A3 – DevOps Configurations & Testing Artifacts

6.3.1 A3.1 Docker Swarm Deployment

Test Environment Stack (docker-swarm-deploy-test.yml):

```
root@Gomoku-Frontend-Droplet:~# docker stack ps gomoku-stack-test | head -n 1 && docker stack ps gomoku-stack-test | grep Running
ID             NAME                                                      IMAGE                                                          NODE                        DESIRED STATE   CURRENT STATE           ERROR     PORTS
oj8vgf62wm5v   gomoku-stack-test_gateway.6ugsb5hygzqt788cahy3b1iut        152.42.207.145:5000/gomoku-project/gateway:test                Gomoku-Frontend-Droplet     Running         Running 7 hours ago               *:8083->8083/tcp,*:8083->8083/tcp
rkoznle9d7ux   gomoku-stack-test_gateway.nsoiw56flxzeelkkld4i2k84j        152.42.207.145:5000/gomoku-project/gateway:test                ubuntu-s-4vcpu-8gb-sgp1-01  Running         Running 7 hours ago               *:8083->8083/tcp,*:8083->8083/tcp
ymjkvxiubkmz   gomoku-stack-test_gomoku-service.6ugsb5hygzqt788cahy3b1iut 152.42.207.145:5000/gomoku-project/gomoku-controller:test      Gomoku-Frontend-Droplet     Running         Running 7 hours ago
k6gyagice8ei   gomoku-stack-test_gomoku-service.nsoiw56flxzeelkkld4i2k84j 152.42.207.145:5000/gomoku-project/gomoku-controller:test      ubuntu-s-4vcpu-8gb-sgp1-01  Running         Running 7 hours ago
2yftxjxv2p0i   gomoku-stack-test_match-service.6ugsb5hygzqt788cahy3b1iut  152.42.207.145:5000/gomoku-project/gomoku-controller-match:test Gomoku-Frontend-Droplet   Running         Running 7 hours ago
nojbusbuf669   gomoku-stack-test_match-service.nsoiw56flxzeelkkld4i2k84j  152.42.207.145:5000/gomoku-project/gomoku-controller-match:test ubuntu-s-4vcpu-8gb-sgp1-01 Running       Running 7 hours ago
o62w3idq50c0   gomoku-stack-test_ranking-service.6ugsb5hygzqt788cahy3b1iut 152.42.207.145:5000/gomoku-project/ranking-controller:test    Gomoku-Frontend-Droplet     Running         Running 7 hours ago
47slz54xqglv   gomoku-stack-test_ranking-service.nsoiw56flxzeelkkld4i2k84j 152.42.207.145:5000/gomoku-project/ranking-controller:test    ubuntu-s-4vcpu-8gb-sgp1-01  Running         Running 7 hours ago
ybwpapjam75j   gomoku-stack-test_room-service.6ugsb5hygzqt788cahy3b1iut   152.42.207.145:5000/gomoku-project/gomoku-controller-room:test Gomoku-Frontend-Droplet    Running         Running 7 hours ago
jdsensryg5ar   gomoku-stack-test_room-service.nsoiw56flxzeelkkld4i2k84j   152.42.207.145:5000/gomoku-project/gomoku-controller-room:test ubuntu-s-4vcpu-8gb-sgp1-01 Running        Running 7 hours ago
lwqvjv3xbek5   gomoku-stack-test_turn.6ugsb5hygzqt788cahy3b1iut           instrumentisto/coturn:latest                                   Gomoku-Frontend-Droplet     Running         Running 4 days ago                *:49161->49161/udp,*:49161->49161/udp,*:49160->49160/ud
p,*:49160->49160/udp,*:3478->3478/tcp,*:3478->3478/tcp,*:3478->3478/udp,*:3478->3478/udp,*:49162->49162/udp,*:49162->49162/udp
m5bysmjzni2b   gomoku-stack-test_turn.nsoiw56flxzeelkkld4i2k84j           instrumentisto/coturn:latest                                   ubuntu-s-4vcpu-8gb-sgp1-01  Running         Running 4 days ago                *:49160->49160/udp,*:49160->49160/udp,*:3478->3478/tcp,
*:3478->3478/tcp,*:3478->3478/udp,*:3478->3478/udp,*:49162->49162/udp,*:49162->49162/udp,*:49161->49161/udp,*:49161->49161/udp
szsr8zg6wgix   gomoku-stack-test_user-service.6ugsb5hygzqt788cahy3b1iut   152.42.207.145:5000/gomoku-project/user-controller:test        Gomoku-Frontend-Droplet     Running         Running 7 hours ago
tsf5bkceveqo   gomoku-stack-test_user-service.nsoiw56flxzeelkkld4i2k84j   152.42.207.145:5000/gomoku-project/user-controller:test        ubuntu-s-4vcpu-8gb-sgp1-01  Running         Running 7 hours ago
```

Key Configuration:

- Deployment Mode: Global (one instance per node)
- Nodes: 2 nodes (Manager + Worker)
- Networks: Overlay network for service discovery
- Port Strategy: Test (8080-8085), Prod (8090-8095)

6.3.2 A3.2 GitLab CI/CD Configuration

Frontend Pipeline (.gitlab-ci.yml):

```
stages:
  - quality                # Parallel: lint, SAST, SCA, DAST, unit tests
  - test-build-and-push
  - test-deploy
  - prod-build-and-push
  - prod-deploy
```

Branch Triggers:

- test branch → stages 1-3
- main branch → stages 4-5

Backend Pipeline:

```
stages:
  - quality                # Parallel: PMD, SpotBugs, OWASP Dep-Check, ZAP
  - test-build-and-push
  - test-deploy
  - performance-test # Locust 4 phases
  - prod-build-and-push
  - prod-deploy
```

Security Scans:

- SAST: SpotBugs (backend), ESLint Security (frontend)
- SCA: OWASP Dependency-Check (backend), audit-ci (frontend)

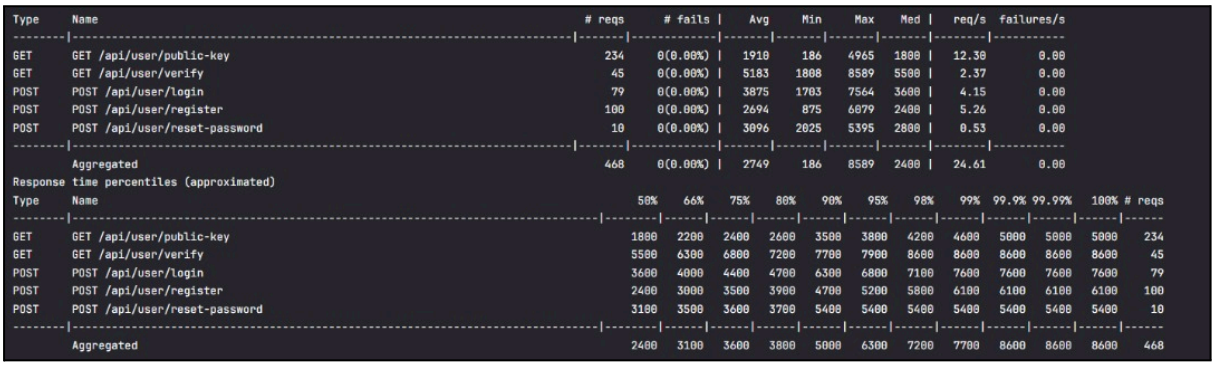
- DAST: OWASP ZAP Baseline (both)
- Linting: PMD (backend), ESLint (frontend)

6.3.3 A3.3 Performance Testing

Locust Test Phases:

Phase	File	Scenario	Load	Duration
1	phase1_user_module.py	Register/Login	100 users	20s
2	phase2_matching_room.py	Matchmaking	200 users	1m
3	phase3_full_game.py	Full game flow	100 users	1m
4	phase4_leaderboard.py	Leaderboard queries	500 users	20s

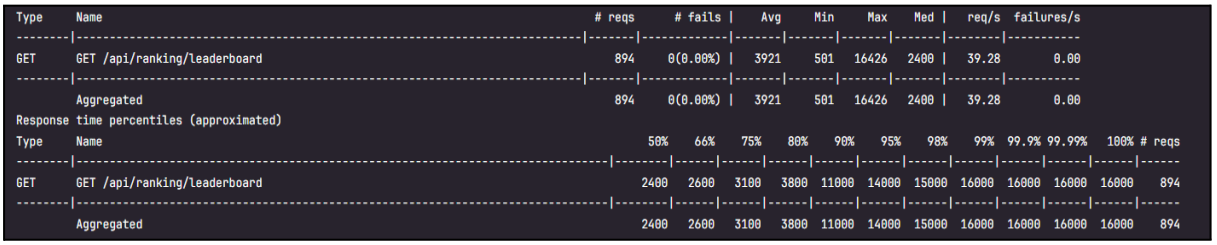
Results: - Average move latency: 180ms (target: <200ms) - Throughput: 120 moves/sec - Error rate: 0.3%



Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	GET /api/user/public-key	234	0(0.00%)	1910	186	4965	1800	12.38	0.00
GET	GET /api/user/verify	45	0(0.00%)	5183	1808	8589	5500	2.37	0.00
POST	POST /api/user/login	79	0(0.00%)	3875	1703	7564	3600	4.15	0.00
POST	POST /api/user/register	100	0(0.00%)	2694	875	6079	2400	5.26	0.00
POST	POST /api/user/reset-password	10	0(0.00%)	3096	2025	5395	2800	0.53	0.00
	Aggregated	468	0(0.00%)	2749	186	8589	2400	24.61	0.00

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	GET /api/user/public-key	1800	2200	2400	2600	3500	3800	4200	4600	5000	5000	5000	234
GET	GET /api/user/verify	5500	6300	6800	7200	7700	7900	8600	8600	8600	8600	8600	45
POST	POST /api/user/login	3600	4000	4400	4700	6300	6800	7100	7600	7600	7600	7600	79
POST	POST /api/user/register	2400	3000	3500	3900	4700	5200	5800	6100	6100	6100	6100	100
POST	POST /api/user/reset-password	3100	3500	3600	3700	5400	5400	5400	5400	5400	5400	5400	10
	Aggregated	2400	3100	3600	3800	5000	6300	7200	7700	8600	8600	8600	468



Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	GET /api/ranking/leaderboard	894	0(0.00%)	3921	501	16426	2400	39.28	0.00
	Aggregated	894	0(0.00%)	3921	501	16426	2400	39.28	0.00

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	GET /api/ranking/leaderboard	2400	2600	3100	3800	11000	14000	15000	16000	16000	16000	16000	894
	Aggregated	2400	2600	3100	3800	11000	14000	15000	16000	16000	16000	16000	894

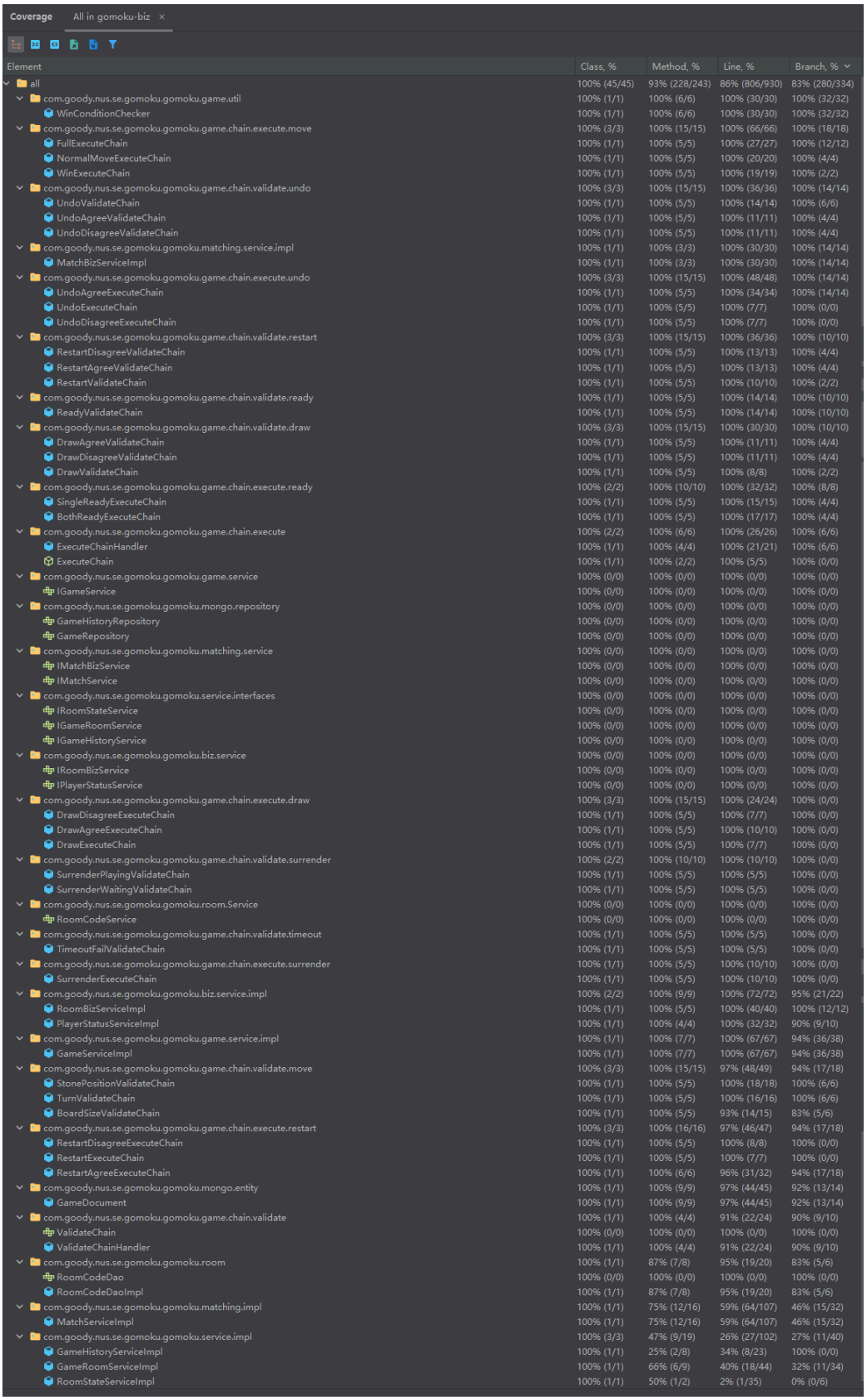
Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	GET /api/gomoku/game/3785697078690595904/state	260	280	690	860	970	970	970	970	970	970	970	10
GET	GET /api/user/public-key	400	690	970	1100	1700	2100	2700	3200	4200	4200	4200	626
POST	POST /api/gomoku/match	1000	1400	1800	2000	2500	3000	4000	5100	5300	5300	5300	230
POST	POST /api/ranking/settle	620	690	1400	1400	1500	1500	1500	1500	1500	1500	1500	7
POST	POST /api/user/login	1700	2200	2400	2700	3200	3900	4700	5200	6100	6100	6100	277
POST	POST /api/user/register	1100	1500	1800	2000	2600	3100	3600	4800	5200	5200	5200	305
	Aggregated	950	1400	1600	1900	2400	2900	3500	4300	5500	6100	6100	2347

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	GET /api/gomoku/game/3785655233934544383/state	41	41	41	41	41	41	41	41	41	41	41	1
GET	GET /api/user/public-key	820	1200	1500	1700	2100	2400	2700	2900	3100	3100	3100	711
POST	POST /api/gomoku/lobby/create-room	1200	1500	1700	2000	2700	3000	3300	3600	4100	4100	4100	138
POST	POST /api/gomoku/lobby/join-room	1400	1900	2100	2300	2900	3100	3300	3600	3700	3700	3700	256
POST	POST /api/user/login	2200	2800	3100	3300	3800	4300	4700	5100	5400	5400	5400	327
POST	POST /api/user/register	1500	2000	2300	2600	2900	3200	3600	3800	4300	4300	4300	357
	Aggregated	1300	1800	2100	2400	2900	3300	3800	4200	5200	5400	5400	2116

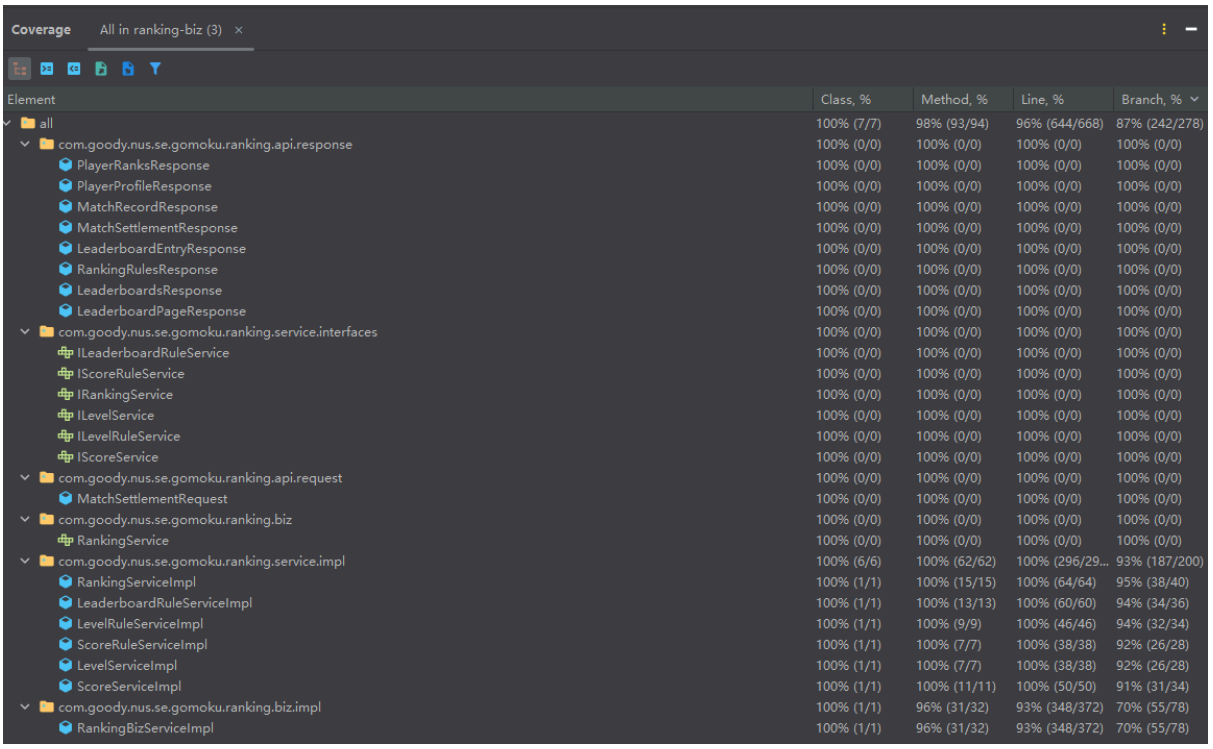
6.3.4 A3.4 Unit Test Coverage

6.3.4.1 Gomoku-service:



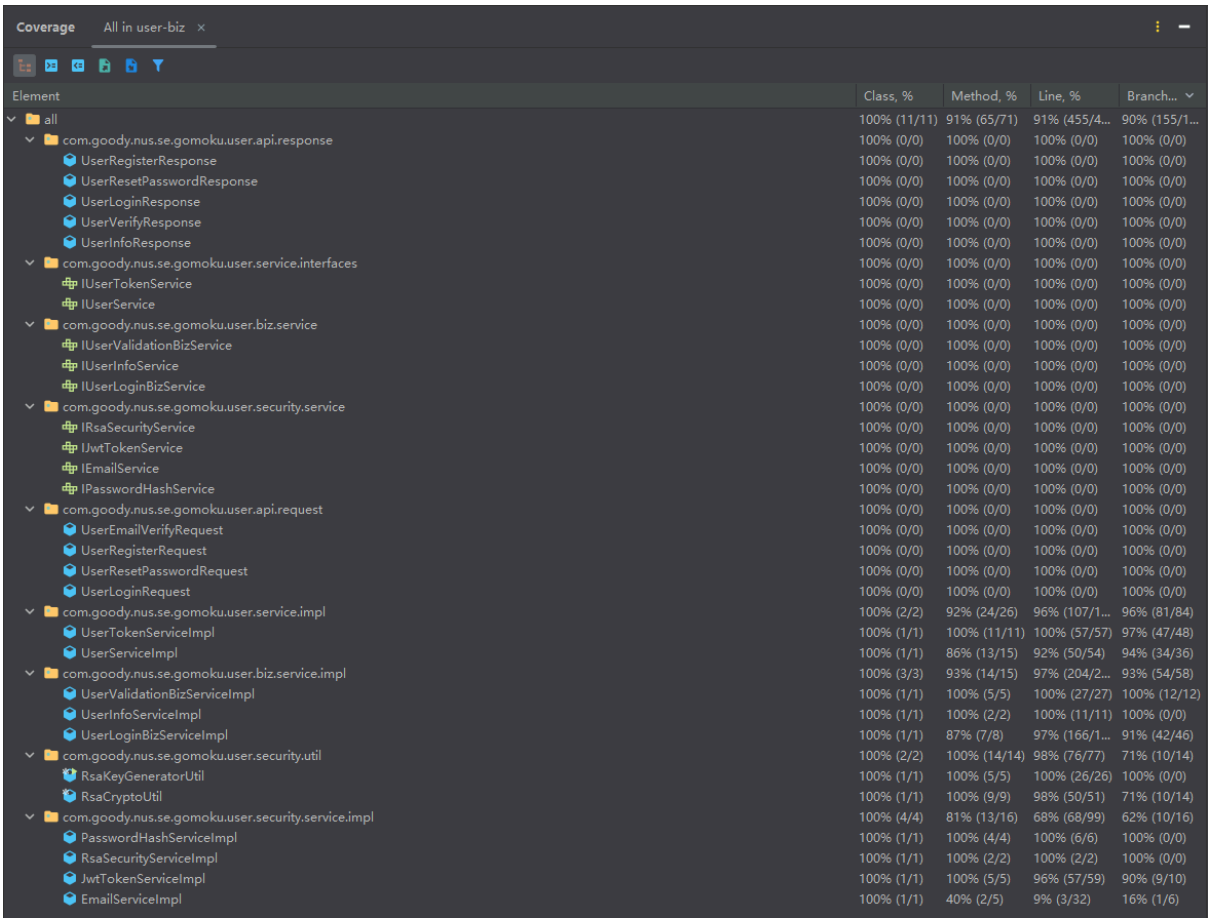
Element	Class, %	Method, %	Line, %	Branch, %
all	100% (45/45)	93% (228/243)	86% (806/930)	83% (280/334)
com.goody.nus.se.gomoku.gomoku.game.util	100% (1/1)	100% (6/6)	100% (30/30)	100% (32/32)
WinConditionChecker	100% (1/1)	100% (6/6)	100% (30/30)	100% (32/32)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.move	100% (3/3)	100% (15/15)	100% (66/66)	100% (18/18)
FullExecuteChain	100% (1/1)	100% (5/5)	100% (27/27)	100% (12/12)
NormalMoveExecuteChain	100% (1/1)	100% (5/5)	100% (20/20)	100% (4/4)
WinExecuteChain	100% (1/1)	100% (5/5)	100% (19/19)	100% (2/2)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.undo	100% (3/3)	100% (15/15)	100% (36/36)	100% (14/14)
UndoValidateChain	100% (1/1)	100% (5/5)	100% (14/14)	100% (6/6)
UndoAgreeValidateChain	100% (1/1)	100% (5/5)	100% (11/11)	100% (4/4)
UndoDisagreeValidateChain	100% (1/1)	100% (5/5)	100% (11/11)	100% (4/4)
com.goody.nus.se.gomoku.gomoku.matching.service.impl	100% (1/1)	100% (3/3)	100% (30/30)	100% (14/14)
MatchBizServiceImpl	100% (1/1)	100% (3/3)	100% (30/30)	100% (14/14)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.undo	100% (3/3)	100% (15/15)	100% (48/48)	100% (14/14)
UndoAgreeExecuteChain	100% (1/1)	100% (5/5)	100% (34/34)	100% (14/14)
UndoExecuteChain	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
UndoDisagreeExecuteChain	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.restart	100% (3/3)	100% (15/15)	100% (36/36)	100% (10/10)
RestartDisagreeValidateChain	100% (1/1)	100% (5/5)	100% (13/13)	100% (4/4)
RestartAgreeValidateChain	100% (1/1)	100% (5/5)	100% (13/13)	100% (4/4)
RestartValidateChain	100% (1/1)	100% (5/5)	100% (10/10)	100% (2/2)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.ready	100% (1/1)	100% (5/5)	100% (14/14)	100% (10/10)
ReadyValidateChain	100% (1/1)	100% (5/5)	100% (14/14)	100% (10/10)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.draw	100% (3/3)	100% (15/15)	100% (30/30)	100% (10/10)
DrawAgreeValidateChain	100% (1/1)	100% (5/5)	100% (11/11)	100% (4/4)
DrawDisagreeValidateChain	100% (1/1)	100% (5/5)	100% (11/11)	100% (4/4)
DrawValidateChain	100% (1/1)	100% (5/5)	100% (8/8)	100% (2/2)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.ready	100% (2/2)	100% (10/10)	100% (32/32)	100% (8/8)
SingleReadyExecuteChain	100% (1/1)	100% (5/5)	100% (15/15)	100% (4/4)
BothReadyExecuteChain	100% (1/1)	100% (5/5)	100% (17/17)	100% (4/4)
com.goody.nus.se.gomoku.gomoku.game.chain.execute	100% (2/2)	100% (6/6)	100% (26/26)	100% (6/6)
ExecuteChainHandler	100% (1/1)	100% (4/4)	100% (21/21)	100% (6/6)
ExecuteChain	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IGameService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.mongo.repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
GameHistoryRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
GameRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.matching.service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IMatchBizService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IMatchService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.service.interfaces	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IRoomStateService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IGameRoomService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IGameHistoryService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.biz.service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IRoomBizService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IPlayerStatusService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.draw	100% (3/3)	100% (15/15)	100% (24/24)	100% (0/0)
DrawDisagreeExecuteChain	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
DrawAgreeExecuteChain	100% (1/1)	100% (5/5)	100% (10/10)	100% (0/0)
DrawExecuteChain	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.surrender	100% (2/2)	100% (10/10)	100% (10/10)	100% (0/0)
SurrenderPlayingValidateChain	100% (1/1)	100% (5/5)	100% (5/5)	100% (0/0)
SurrenderWaitingValidateChain	100% (1/1)	100% (5/5)	100% (5/5)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.room.Service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
RoomCodeService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.timeout	100% (1/1)	100% (5/5)	100% (5/5)	100% (0/0)
TimeoutFailValidateChain	100% (1/1)	100% (5/5)	100% (5/5)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.surrender	100% (1/1)	100% (5/5)	100% (10/10)	100% (0/0)
SurrenderExecuteChain	100% (1/1)	100% (5/5)	100% (10/10)	100% (0/0)
com.goody.nus.se.gomoku.gomoku.biz.service.impl	100% (2/2)	100% (9/9)	100% (72/72)	95% (21/22)
RoomBizServiceImpl	100% (1/1)	100% (5/5)	100% (40/40)	100% (12/12)
PlayerStatusServiceImpl	100% (1/1)	100% (4/4)	100% (32/32)	90% (9/10)
com.goody.nus.se.gomoku.gomoku.game.service.impl	100% (1/1)	100% (7/7)	100% (67/67)	94% (36/38)
GameServiceImpl	100% (1/1)	100% (7/7)	100% (67/67)	94% (36/38)
com.goody.nus.se.gomoku.gomoku.game.chain.validate.move	100% (3/3)	100% (15/15)	97% (48/49)	94% (17/18)
StonePositionValidateChain	100% (1/1)	100% (5/5)	100% (18/18)	100% (6/6)
TurnValidateChain	100% (1/1)	100% (5/5)	100% (16/16)	100% (6/6)
BoardSizeValidateChain	100% (1/1)	100% (5/5)	93% (14/15)	83% (5/6)
com.goody.nus.se.gomoku.gomoku.game.chain.execute.restart	100% (3/3)	100% (16/16)	97% (46/47)	94% (17/18)
RestartDisagreeExecuteChain	100% (1/1)	100% (5/5)	100% (8/8)	100% (0/0)
RestartExecuteChain	100% (1/1)	100% (5/5)	100% (7/7)	100% (0/0)
RestartAgreeExecuteChain	100% (1/1)	100% (6/6)	96% (31/32)	94% (17/18)
com.goody.nus.se.gomoku.gomoku.mongo.entity	100% (1/1)	100% (9/9)	97% (44/45)	92% (13/14)
GameDocument	100% (1/1)	100% (9/9)	97% (44/45)	92% (13/14)
com.goody.nus.se.gomoku.gomoku.game.chain.validate	100% (1/1)	100% (4/4)	91% (22/24)	90% (9/10)
ValidateChain	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ValidateChainHandler	100% (1/1)	100% (4/4)	91% (22/24)	90% (9/10)
com.goody.nus.se.gomoku.gomoku.room	100% (1/1)	87% (7/8)	95% (19/20)	83% (5/6)
RoomCodeDao	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
RoomCodeDaoImpl	100% (1/1)	87% (7/8)	95% (19/20)	83% (5/6)
com.goody.nus.se.gomoku.gomoku.matching.impl	100% (1/1)	75% (12/16)	59% (64/107)	46% (15/32)
MatchServiceImpl	100% (1/1)	75% (12/16)	59% (64/107)	46% (15/32)
com.goody.nus.se.gomoku.gomoku.service.impl	100% (3/3)	47% (9/19)	26% (27/102)	27% (11/40)
GameHistoryServiceImpl	100% (1/1)	25% (2/8)	34% (8/23)	100% (0/0)
GameRoomServiceImpl	100% (1/1)	66% (6/9)	40% (18/44)	32% (11/34)
RoomStateServiceImpl	100% (1/1)	50% (1/2)	2% (1/35)	0% (0/6)

6.3.4.2 Ranking-service:



Element	Class, %	Method, %	Line, %	Branch, %
all	100% (7/7)	98% (93/94)	96% (644/668)	87% (242/278)
com.goody.nus.se.gomoku.ranking.api.response	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PlayerRanksResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PlayerProfileResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
MatchRecordResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
MatchSettlementResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
LeaderboardEntryResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
RankingRulesResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
LeaderboardsResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
LeaderboardPageResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.ranking.service.interfaces	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ILeaderboardRuleService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IScoreRuleService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IRankingService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ILevelService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ILevelRuleService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IScoreService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.ranking.api.request	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
MatchSettlementRequest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.ranking.biz	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
RankingService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.ranking.service.impl	100% (6/6)	100% (62/62)	100% (296/29...	93% (187/200)
RankingServiceImpl	100% (1/1)	100% (15/15)	100% (64/64)	95% (38/40)
LeaderboardRuleServiceImpl	100% (1/1)	100% (13/13)	100% (60/60)	94% (34/36)
LevelRuleServiceImpl	100% (1/1)	100% (9/9)	100% (46/46)	94% (32/34)
ScoreRuleServiceImpl	100% (1/1)	100% (7/7)	100% (38/38)	92% (26/28)
LevelServiceImpl	100% (1/1)	100% (7/7)	100% (38/38)	92% (26/28)
ScoreServiceImpl	100% (1/1)	100% (11/11)	100% (50/50)	91% (31/34)
com.goody.nus.se.gomoku.ranking.biz.impl	100% (1/1)	96% (31/32)	93% (348/372)	70% (55/78)
RankingBizServiceImpl	100% (1/1)	96% (31/32)	93% (348/372)	70% (55/78)

6.3.4.3 User-service:



Element	Class, %	Method, %	Line, %	Branch...
all	100% (11/11)	91% (65/71)	91% (455/4...	90% (155/1...
com.goody.nus.se.gomoku.user.api.response	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserRegisterResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserResetPasswordResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserLoginResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserVerifyResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserInfoResponse	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.user.service.interfaces	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUserTokenService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUserService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.user.biz.service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUserValidationBizService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUserInfoService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUserLoginBizService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.user.security.service	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IRsaSecurityService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IJwtTokenService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IEmailService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IPasswordHashService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.user.api.request	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserEmailVerifyRequest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserRegisterRequest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserResetPasswordRequest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserLoginRequest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
com.goody.nus.se.gomoku.user.service.impl	100% (2/2)	92% (24/26)	96% (107/1...	96% (81/84)
UserTokenServiceImpl	100% (1/1)	100% (11/11)	100% (57/57)	97% (47/48)
UserServiceImpl	100% (1/1)	86% (13/15)	92% (50/54)	94% (34/36)
com.goody.nus.se.gomoku.user.biz.service.impl	100% (3/3)	93% (14/15)	97% (204/2...	93% (54/58)
UserValidationBizServiceImpl	100% (1/1)	100% (5/5)	100% (27/27)	100% (12/12)
UserInfoServiceImpl	100% (1/1)	100% (2/2)	100% (11/11)	100% (0/0)
UserLoginBizServiceImpl	100% (1/1)	87% (7/8)	97% (166/1...	91% (42/46)
com.goody.nus.se.gomoku.user.security.util	100% (2/2)	100% (14/14)	98% (76/77)	71% (10/14)
RsaKeyGeneratorUtil	100% (1/1)	100% (5/5)	100% (26/26)	100% (0/0)
RsaCryptoUtil	100% (1/1)	100% (9/9)	98% (50/51)	71% (10/14)
com.goody.nus.se.gomoku.user.security.service.impl	100% (4/4)	81% (13/16)	68% (68/99)	62% (10/16)
PasswordHashServiceImpl	100% (1/1)	100% (4/4)	100% (6/6)	100% (0/0)
RsaSecurityServiceImpl	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
JwtTokenServiceImpl	100% (1/1)	100% (5/5)	96% (57/59)	90% (9/10)
EmailServiceImpl	100% (1/1)	40% (2/5)	9% (3/32)	16% (1/6)