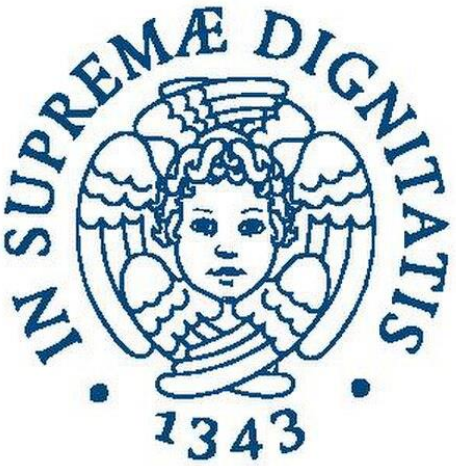


UNIVERSITÀ DI PISA

Final Project Report

Battle Card Game

Advanced Software Engineering 2025

Group North+Serbia

Members:

Alexander MITTET 708819

Estelle KEYMEULEN 712904

Katrine MIKALSEN 710649

Tara POPOV 722164

Table of Contents

Table of Contents 2

Cards Overview 4

- Card Structure 4
- Card Attributes 4
- Card Hierarchy 4
- Visual Representation 4
- Deck Management 5

Architecture Overview 6

- Architecture Diagram 6
- Microservices Summary 6
- Microservice-to-Microservice Connections 7
- Architectural choices 7

User Stories 9

Game rules 11

- Rules of the Game 11

Game flow 12

- Phase 1: Setup (Both Players) 12
- Phase 2: Game Creation (Player 1) 12
- Phase 3: Deck Selection (Both Players) 12
- Phase 4: Gameplay (7 Rounds) 13
- Phase 5: Game Completion 13

Testing 14

- Testing Implementation Details 14

Security - Data 16

- Sanitise input: Username 16
- Encrypted Data at Rest 17

Security – Authorization and Authentication 18

- Authentication Architecture: Centralized Issuance, Distributed Validation 18
- Handling Expired Access Tokens 19

Security - Analyses 21

Static Analysis Tool Results 21
Docker Scout Vulnerability Analysis 22
Threat Model: Battle Card Game Microservices Architecture 23
Use of Generative AI 29
Overview 29
AI Tools Used 29
Specific Use Cases 29
Honest Assessment 31
Conclusion 31
Additional Features 33
Feature Category 1: Security Enhancements (Beyond Requirements) 33
Feature Category 2: Frontend and User Experience 40
Summary of Additional Features 47

Cards Overview

Card Structure

The Battle Card Game uses a standard deck of **39 cards** divided into three types based on the classic Rock-Paper-Scissors mechanic:

Card Type	Count	Number Range	Special Properties
Rock	13	1-13	Beats Scissors
Paper	13	1-13	Beats Rock
Scissors	13	1-13	Beats Paper

Card Attributes

Each card has the following properties:

- **Type**: Rock, Paper, or Scissors (determines type advantage)
- **Power**: Number from 1 to 13 (determines strength within same type)
- **Card ID**: Unique identifier in database

Card Hierarchy

Type Advantage (primary):

- Rock > Scissors
- Scissors > Paper
- Paper > Rock

Power Hierarchy (secondary, when types match):

- 13 > 12 > 11 > ... > 3 > 2 > 1

Visual Representation

Deck Management

Players interact with cards through:

1. **View Collection**: Browse all 39 available cards before game starts
2. **Deck Selection**: Choose 22 cards to form their game deck
3. **Hand Management**: Draw 3 cards per round from their deck
4. **Card Play**: Select and play 1 card per round from their hand

Cards are stored in the PostgreSQL database and managed by the dedicated **Cards Service** microservice, which handles all card-related queries and validation.

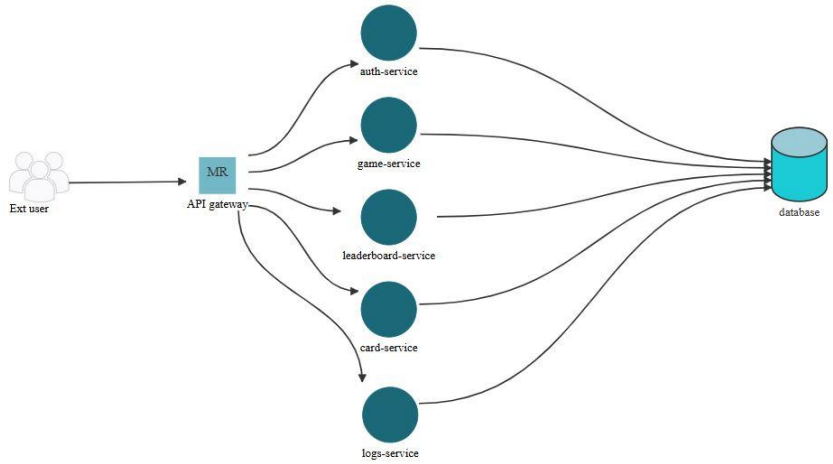
Architecture Overview

The Battle Card Game application implements a **microservice-based architecture** where five independent Python/Flask services (Auth, Cards, Game, Leaderboard, and Logs) communicate through REST APIs via an **Nginx API Gateway**, all persisting data to a shared **PostgreSQL database**.

The system implements JWT-based authentication, HTTPS/TLS encryption, and comprehensive security measures including input sanitization, account lockout mechanisms, and encrypted game history logging.

The application is distributed using **Docker containers**, where each microservice is packaged into a separate container with its own environment. This approach enables service isolation, easier dependency management, and simpler application startup and deployment.

Architecture Diagram



MicroFreshener was used as a tool for detecting microservice smells and suggesting refactoring solutions, as well as for generating this graph.

Microservices Summary

The Battle Card Game implements a microservice-based architecture with **five business microservices** (Auth, Cards, Game, Leaderboard, Logs) behind an **Nginx API Gateway** that handles routing and TLS termination.

Microservice	Responsibility	Language used
Cards Service	Card database queries, deck generation (random/manual selection), card statistics, filtering by type/power	Python (Flask)

Game Service	Game state management, battle logic resolution, game invitations, deck selection, round outcomes, game history archiving	Python (Flask)
Leaderboard Service	Global rankings, player statistics aggregation, win/loss calculations, performance metrics, recent game history	Python (Flask)
Auth Service	User registration, login, profile management, JWT token generation/validation, concurrent session control, account lockout after failed attempts	Python (Flask)
Logs Service	User action audit trails, security event logging, admin-only log retrieval, system monitoring	Python (Flask)
Api Gateway	Request routing, HTTPS/TLS encryption, load balancing, endpoint consolidation	Nginx + Python (Flask)

Microservice-to-Microservice Connections

Connected Services	Reason for Connection
Game Service and Card Service	Game Service calls /api/cards/random-deck to generate random card selections for both players at game start

Design choices

Functional separation: Each microservice encapsulates a specific domain:

- **Cards** → card logic & storage
- **Game** → gameplay & state
- **Leaderboard** → player statistics & rankings
- **Auth** → user security & session control
- **Logs** → monitoring & audit trails

Service interactions rationale:

- Game Service needs Cards Service to provide decks dynamically.

Deployment & isolation:

- Each service runs in a separate Docker container.
- Enables independent scaling, easy dependency management, and simplified CI/CD.

Architectural Trade-offs & Limitations

We identified the **Shared Persistence** architectural smell in our application and explored the **Database Manager** pattern as a potential solution. This pattern addresses the issue by centralizing all database access within a single, dedicated component.

Given the scale of the application and the available development timeframe, we initially selected this approach as the most appropriate architectural direction. The intended solution introduced a **DB Manager** as a **new internal microservice**, responsible exclusively for persistence concerns. This service was designed to interact only with the database and the other backend microservices, without being exposed through the API Gateway, thereby remaining an internal infrastructural component, for handling all persistence logic and mediating all interactions with the database.

The implementation reached an advanced stage and was **largely functional**. However, integrating the DB Manager required substantial changes across multiple services, and a number of unresolved issues remained. Due to time constraints, we were unable to complete the remaining refactoring and stabilization efforts necessary to guarantee full build stability and complete test coverage.

To avoid introducing potential regressions or compromising the stability of the existing system, the DB Manager implementation was maintained in a separate branch (*manager*) and intentionally **not merged into the main branch** for the final delivery. This decision reflects a conscious trade-off prioritizing system **reliability over partially stabilized architectural changes.**

We are confident that a fully completed implementation of this approach would provide several significant benefits:

- **Reduced complexity:** All persistence logic would be centralized in a single, well-defined component.
- **Improved maintainability:** Database schema changes and query modifications would be handled in one place, reducing the impact on individual services.
- **Enhanced security:** Sensitive database operations would be isolated within the DB Manager, simplifying access control, auditing, and monitoring.

In the intended architecture, services such as **Game**, **Leaderboard**, **Logs**, **Authentication**, and **Cards**—all of which require database access—would no longer interact with the database directly. Instead, they would communicate exclusively with the **DB Manager**, which would be solely responsible for executing all database queries and managing persistence concerns.

User Stories

#	User Story	Endpoints	Microservices Involved
1	Create an account SO THAT I can participate in the game	• POST /api/auth/register	Gateway → Auth Service → Database → Auth Service → Gateway
2	Login into the game SO THAT I can play a match	• POST /api/auth/login	Gateway → Auth Service → Database → Auth Service → Gateway
3	Check/modify my profile SO THAT I can update my information	• GET /api/auth/profile • PUT /api/auth/profile	Gateway → Auth Service → Database → Auth Service → Gateway
4	Be safe about my account data SO THAT nobody can steal/modify them	• All authenticated endpoints (JWT validation) • Password hashing (bcrypt) • Input sanitization on all services	Auth Service (JWT tokens, password hashing)All Services (input validation, SecurityMiddleware)
5	See the overall card collection SO THAT I can think of a strategy	• GET /api/cards	Gateway → Card Service → Database → Card Service → Gateway
6	View the details of a card SO THAT I can think of a strategy	• GET /api/cards/{card_id}	Gateway → Card Service → Database → Card Service → Gateway
7	Start a new game SO THAT I can play	• POST /api/games	Gateway → Game Service → Database → Game Service → Gateway
8	Select the subset of cards SO THAT I can start a match	• POST /api/games/{game_id}/select-deck	Gateway → Game Service → Card Service (for power assignment) → Database → Game Service → Gateway
9	Select a card SO THAT I can play my turn	• POST /api/games/{game_id}/play-card	Gateway → Game Service → Database → Game Service → Gateway
10	Know the score SO THAT I know if I am winning or losing	• GET /api/games/{game_id}	Gateway → Game Service → Database → Game Service → Gateway

11	Know the turns SO THAT I can know how many rounds are left	• GET /api/games/{game_id} • GET /api/games/{game_id}/turn-info	Gateway → Game Service → Database → Game Service → Gateway
12	See the score SO THAT I know who is winning	• GET /api/games/{game_id}	Gateway → Game Service → Database → Game Service → Gateway
13	Know who won the turn SO THAT I know the updated score	• POST /api/games/{game_id}/resolve-round	Gateway → Game Service → Database → Game Service → Gateway
14	Know who won a match SO THAT I know the result	• GET /api/games/{game_id} (winner field) • GET /api/games/{game_id}/details	Gateway → Game Service → Database → Game Service → Gateway
15	That the rules are not violated SO THAT I can play a fair match	• All game logic endpoints enforce rules • Validation in POST /api/games/{game_id}/play-card • Validation in POST /api/games/{game_id}/resolve-round	Game Service (server-side validation, game state checks, turn management)
16	View the list of my old matches SO THAT I can see how I played	• GET /api/games/user/{username} • GET /api/leaderboard/my-matches	Gateway → Game Service / Leaderboard Service → Database → Service → Gateway
17	View the details of one of my matches SO THAT I can see how I played	• GET /api/games/{game_id}/details • GET /api/games/{game_id}/history	Gateway → Game Service → Database (game_history table with encryption) → Game Service → Gateway
18	View the leaderboards SO THAT I know who the best players are	• GET /api/leaderboard • GET /api/leaderboard/top-players	Gateway → Leaderboard Service → Database → Leaderboard Service → Gateway
19	Prevent people from tampering my old matches SO THAT I have them available	• Game history encryption/signing in database • GET /api/games/{game_id}/history (secured with HMAC)	Game Service (game_history table with encrypted snapshots, HMAC validation via security.py)

Game rules

For our game, we wanted to create an experience that is easy to understand yet slightly more strategic. We chose Rock-Paper-Scissors because its rules are simple and familiar, making it easy for players to remember which choice beats another. To add depth and reduce reliance on luck, we incorporated numbers on the cards, so that higher numbers generally beat lower ones.

Rules of the Game

The overall card deck has 39 cards: Rock, Paper, and Scissors suits, each numbered 1 to 13. The game starts with the player viewing all possible cards.

Overall Rules

1. Each player selects 22 cards from their own overall card deck. This becomes the player's deck for the game.
2. At the start of each round, each player chooses a hand of three cards for that round.
3. Both players play one card simultaneously.
4. The winning card gives one point to the winning player. If it's a tie, no one gets a point.

Repeat steps 2–4 for all 7 rounds.

Ordering Rules

1. Rock beats Scissors, Scissors beats Paper, Paper beats Rock (and the cycle repeats).
2. Among the numbers 1–13, the larger number always beats the smaller one, making the relation transitive: 13 > 12 > 11 > 10 > ... > 2 > 1.

This means that among the numbers 1–13, the larger number always beats the smaller one.

Game flow

Step-by-step API call sequence for two logged-in players to complete a match.

Prerequisites:

- Base URL: https://localhost:8443

Phase 1: Setup (Both Players)

1.1: Register/Login Both Players

```
POST /api/auth/register
Content-Type: application/json
{"username": "player1", "password": "password123!"}
POST /api/auth/register
Content-Type: application/json
{"username": "player2", "password": "password123!"}
```

Phase 2: Game Creation (Player 1)

2.1: Player 1 Creates Game

```
POST /api/games
Authorization: Bearer {P1_TOKEN}
Content-Type: application/json
{"player2_name": "player2"}
```

Phase 3: Deck Selection (Both Players)

3.1: Player 1 Selects Deck (22 cards)

```
POST /api/games/{GAME_ID}/select-deck
Authorization: Bearer {P1_TOKEN}
Content-Type: application/json
{
  "deck": [
    {"type": "Rock"}, {"type": "Rock"}, {"type": "Paper"},
    {"type": "Paper"}, {"type": "Scissors"}, {"type": "Scissors"},
    ... (22 cards total - mix of Rock, Paper, Scissors)
  ]
}
```

3.2: Player 2 Selects Deck (22 cards)

Repeat for player 2

Phase 4: Gameplay (7 Rounds)

Repeat Steps 4.1-4.5 for each round:4.1: Both Players Draw Hand

POST /api/games/{GAME_ID}/draw-hand
Authorization: Bearer {P1_TOKEN}
POST /api/games/{GAME_ID}/draw-hand
Authorization: Bearer {P2_TOKEN}

Response**:** Returns 3 cards from deck.

4.2: Both Players Play Card

POST /api/games/{GAME_ID}/play-card
Authorization: Bearer {P1_TOKEN}
Content-Type: application/json
{"card_index": 0}

POST /api/games/{GAME_ID}/play-card
Authorization: Bearer {P2_TOKEN}
Content-Type: application/json
{"card_index": 1}

Note**:** card_index is position in hand (0, 1, or 2).

Phase 5: Game Completion

5.1: Check Final State

GET /api/games/{GAME_ID}
Authorization: Bearer {P1_TOKEN}

Testing

Testing Implementation Details

Integration Testing Without Mocking

- All microservices tests (auth, game, card, leaderboard) are integration tests that communicate directly with the actual running services through the Nginx gateway at https://localhost:8443 - no service communication is mocked, ensuring real inter-service behaviour is tested.
- The Game Service tests make real API calls to the Card Service to get decks (via internal service URLs like http://localhost:5002), testing actual microservice-to-microservice communication rather than mocking these dependencies.

SSL Certificate Handling in Testing

- All test files disable SSL certificate verification (session.verify = False) and suppress urllib3 warnings because the application uses self-signed SSL certificates in development - this allows tests to run against HTTPS endpoints without certificate validation errors.

Database State Management

- Tests use a fresh database reset strategy (docker-compose down -v) to match GitHub Actions CI environment, ensuring no pre-existing test data affects results - this prevents tests from passing locally due to leftover data but failing in CI.
- Direct PostgreSQL database connections are used in some tests (e.g., test_game_service.py) to set up or verify complex game states that would be tedious to create through API calls alone.

Unit Testing with Mocking

- The test_security.py file uses unittest.mock.patch and MagicMock to test the InputSanitizer class in isolation, validating SQL injection, XSS, and command injection detection without hitting actual services or databases.

Performance Testing Configuration

- Locust tests (locustfile.py) generate unique usernames with 16-character random strings (e.g., testuser_a7k2m9x4p1q8) for each simulated user to avoid concurrent session conflicts during high-load scenarios, since the system blocks multiple simultaneous logins.

Temporal Isolation

- All tests use timestamp-based unique identifiers (int(time.time() * 1000)) for usernames to prevent collisions between test runs and parallel test executions, ensuring complete test isolation.

Real-World Scenario Testing

- test_complete_game_flow.py plays an entire game from start to finish (registration → deck selection → gameplay until deck depletion (typically 7 rounds with 22-card decks) → completion → leaderboard verification) to validate the complete user journey rather than testing endpoints in isolation.
- test_concurrent_sessions.py simulates multiple browser sessions by using different User-Agent headers in requests to test the strict concurrent session detection mechanism.

Security Testing Strategy

- Account lockout tests (test_account_lockout.py) make exactly 3 failed login attempts to trigger the lockout mechanism, then verify both the lockout state and the password reset flow, testing the complete security feature rather than mocking the lockout logic.

Security - Data

Sanitise input: Username

We had to sanitise the input “username”. The username input represents a user identifier that players use to register, login, and participate in games. It's a critical input as it's used throughout the system for authentication, game creation, leaderboard tracking, and user identification.
The username input is sanitized across three microservices:

- Auth-service - During user registration and login
- Game-service - When creating games and identifying players
- Leaderboard-service - When retrieving player statistics and game history

How it's sanitized:

The sanitization is implemented through the *validate_username()* method in the InputSanitizer class, which implements multiple layers of security:

1. **Length Validation**: Minimum 3 characters and maximum 50 characters
    - sanitize_string() with max_length=50
    - validate_username() with additional minimum 3 check
2. **Pattern Matching**:
    - Only allows alphanumeric characters plus dots (.), underscores (_), and hyphens (-).
    - Uses regex pattern: ^[a-zA-Z0-9_.-]+$
3. **Security Checks**:
    - Scans for SQL injection patterns (e.g., SELECT, DROP, UNION, SQL comments)
    - Checks for XSS patterns (e.g., <script>, javascript:, onclick=)
    - Detects command injection attempts (e.g., semicolons, pipes, path traversal)
4. **Character Filtering**:
    - HTML-encodes dangerous characters using html.escape()
    - Removes non-printable characters
    - Strips additional special characters like <, >, &, quotes
5. **Error Handling**:
    - Raises ValueError with descriptive messages if validation fails
    - Prevents malicious input from reaching the database

This centralized approach ensures consistent security across all microservices and protects against common web vulnerabilities like SQL injection, XSS attacks, and command injection.

Encrypted Data at Rest

Key Security Features:

Passwords: Use industry-standard bcrypt with automatic salting, making them resistant to rainbow table attacks

Game History: Uses Fernet encryption with separate HMAC keys for encryption and integrity verification, ensuring both confidentiality and tamper-detection

Key Management: The GAME_HISTORY_KEY is automatically generated during deployment and stored in environment variables (gitignored for security)

1. User Passwords

What they represent: Authentication credentials for user accounts

Database: PostgreSQL database battlecards, table: users

Encryption method: bcrypt hashing (one-way cryptographic hash)

Where encrypted: Encryption location: Auth Service (app.py, function hash_password() at line ~102)

Where decrypted/verified: Auth Service (app.py, function verify_password() at line ~109)

Storage format: bcrypt-hashed strings in the users.password (VARCHAR(255))

2. Game History Snapshots

What they represent: Immutable records of completed games (game state, player info, round history, timestamps)

Encryption method: Fernet symmetric encryption (AES-128-CBC with PKCS7 padding), HMAC-SHA256 for integrity

Key: 32-byte base64 URL-safe key in `GAME_HISTORY_KEY` environment variable

Where encrypted: Game Service, `archive_game_history()` in `app.py` (~line 253), logic in `HistorySecurity` class in `security.py`

Where decrypted: Game Service (app.py, function decrypt_history_row() at line ~302), when retrieving completed games or displaying user history.

Storage format: `encrypted_payload` (BYTEA), `integrity_hash` (VARCHAR(128)), plus some plaintext metadata (game_id, player names, scores, winner, round history, archived_at)

Security – Authorization and Authentication

Authentication Architecture: Centralized Issuance, Distributed Validation

The system uses a hybrid approach: centralized token issuance with distributed validation.

Aspect	Implementation
Token Issuance	Centralized in Auth Service
Token Validation	Distributed across all microservices
Secret Key Storage	Shared JWT_SECRET_KEY via environment variables
Key Distribution	Docker Compose injects same key to all services

Token Validation Flow:

1) Client sends request with Authorization: Bearer <token> header
2) Microservice extracts token from header
3) Flask-JWT-Extended validates token signature using JWT_SECRET_KEY
4) Library verifies token expiration (exp claim)
5) Library extracts identity (username) from token payload
6) Service uses get_jwt_identity() to get current user
7) Service performs authorization checks (resource ownership, admin status)
8) Request proceeds if authorized, otherwise returns 403 Forbidden

Secret Key Storage and Usage:

Component	Location	Method
Key Storage	Environment variable JWT_SECRET_KEY	Docker Compose .env file
Key Distribution	All microservices receive same key	Environment variable injection
Key Usage	Token signing (Auth Service)	create_access_token(identity=username)
Key Usage	Token validation (All services)	JWTManager validates signature

Configuration	app.config["JWT_SECRET_KEY"]	Loaded from os.getenv("JWT_SECRET_KEY")

Payload format:

Claim	Type	Description	Example
sub	string	Subject (username)	"testuser"
iat	integer	Issued at (Unix timestamp)	1701878400
exp	integer	Expiration time (Unix timestamp)	1701896400
type	string	Token type	"access"
fresh	boolean	Whether token is fresh	true

Handling Expired Access Tokens

Expiration Detection:

Location	Method	Response
Server-side	Flask-JWT-Extended checks exp claim	HTTP 401 with {"error": "Token has expired"}
Client-side	Frontend checks token_expiry timestamp	Proactive refresh 5 minutes before expiry

Token Refresh Flow:

- Access token expires → Client receives HTTP 401
- Client sends refresh token to POST /api/auth/refresh
- Auth Service validates refresh token (database check + signature)
- Auth Service issues new access token
- Client updates stored tokens
- Client retries original request with new access token

Security - Analyses

Static Analysis Tool Results

Tool Used: Bandit (Python Security Linter)

Command executed:

```
bash
# Root directory scan
bandit -r . -f txt

# Microservices scan
bandit -r auth-service/ game-service/ card-service/ leaderboard-service/ logs-service/ database-manager/ -f txt
```

Results Summary (cfr. Github Workflow):

Metric	Root Directory Scan	Microservices Scan
Total Lines of Code	7,027	6,799
High Severity Issues	0	0
Medium Severity Issues	8	8
Low Severity Issues	6	6
Total Issues	14	14

Issue Breakdown by Confidence:

Confidence Level	Count
High	5
Medium	6
Low	3

Key Findings:

- ✅ **No High-Severity Issues** - Critical security vulnerabilities absent from codebase
- ✅ **No Test Results Flagged** - "No issues identified" in test results section
- ✅ **Zero Lines Skipped** - No `#nosec` bypasses used, indicating we didn't suppress warnings
- ✅ **Clean Security Profile** - 14 total issues across 7000+ lines of code represents strong security posture

Docker Scout Vulnerability Analysis

Analyzed Images:

- `battlecards-auth-service:latest`
- `battlecards-game-service:latest`
- `battlecards-card-service:latest`
- `battlecards-leaderboard-service:latest`
- `battlecards-logs-service:latest`

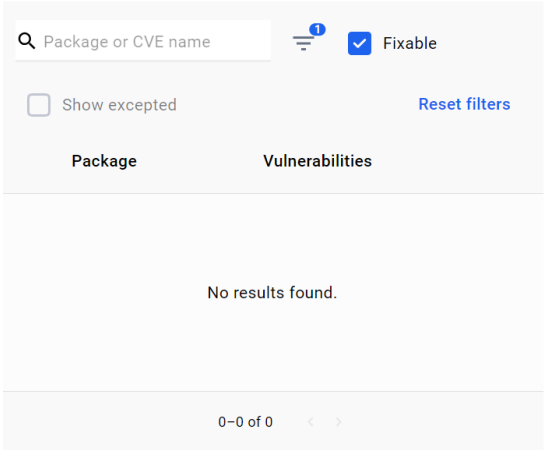
Vulnerability Summary by Image

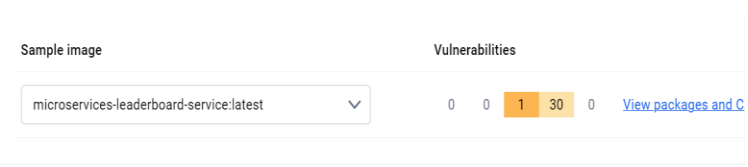
Image	Critical	High	Medium	Low	Total
auth-service	0	0	1	30	31
game-service	0	0	1	30	31
card-service	0	0	1	30	31
leaderboard-service	0	0	1	30	31
logs-service	0	0	1	64	65

Key Findings:

- ✅ **No Critical Vulnerabilities** - Critical security vulnerabilities absent from codebase
- ✅ **No Fixable Vulnerabilities –** All vulnerabilities have been fixed

Example for Leaderboard image





Threat Model: Battle Card Game Microservices Architecture

1) Modeling (System Context & Data Flows)

The Battle Card Game system uses a microservices architecture, consisting of five active microservices and one PostgreSQL database. The client (a browser or testing tool) interacts with the system via an API Gateway.

Component	Purpose	Data Handled
C1: Client/User	Initiates actions (registration, login, gameplay).	User credentials, JWT tokens, game choices.
C2: Nginx API Gateway	Reverse proxy and routing layer on ports 8080 (HTTP) and 8443 (HTTPS)	All incoming/outgoing HTTP requests, SSL/TLS termination
C3: Auth Service	User authentication, profile management, token validation, session management, account lockout	Passwords (bcrypt hashed with salt), JWT access tokens, refresh tokens, session data, failed login attempts
C4: Game Service	Core game logic, state management, battle mechanics, game invitations	Game state, player hands, encrypted archived game history (with HMAC verification)
C5: Card Service	Card collection and deck management (39 total cards).	Card data, card types, deck configurations
C6: Leaderboard Service	Rankings, statistics, and recent games display.	Player statistics and game history data.
C7: Logs service	User action logging and security audit trails	User action logs, authentication events, security events
C8: PostgreSQL Database	Data persistence for all services.	Users table, cards table, games table, game_history table (encrypted), user_logs table, sessions table
C9: Host Filesystem (.env file)	Stores encryption keys and service authentication credentials	GAME_HISTORY_KEY (32-byte Fernet key), service API keys (5 keys for zero-trust), SSL certificates and private keys (when using build-and-start.sh)

2) Identifying Assets (A)

- **A1:** Personally identifiable information (PII) and user profiles.
- **A2:** Credentials, JWT access/refresh tokens, and user sessions.
- **A3:** Secrets and keys (GAME_HISTORY_KEY, JWT signing key, inter-service API keys, database credentials, SSL/TLS private keys (RSA 4096-bit)). Note: When using build-and-start.sh, these are persisted on host filesystem in .env file and nginx/ssl/ directory.).

- **A4:** Game data and state (card inventories, scores, match results).
- **A5:** Game history integrity (archived matches with HMAC verification).
- **A6:** Code integrity, dependencies, and Docker environment.
- **A7:** Service availability (Auth, Card, Game, Leaderboard, Logs microservices).

3) Identifying Attack Surfaces

The primary points of exposure are:

- **Authentication Endpoints:** `/api/auth/register` and `/api/auth/login` (accessible publicly without a token). /api/auth/refresh (accessible with refresh token only, no access token required)
- **Input Parameters:** All request bodies and query parameters used across all protected endpoints (e.g., card indices for `/play-card`, limits for `/api/leaderboard`).
- **API Gateway:** Ports 8080 (HTTP) and 8443 (HTTPS) - the single entry point for all services. Nginx configuration and SSL/TLS certificate validation. SSL private key (RSA 4096-bit) and certificate stored on host filesystem (nginx/ssl/) with volume mount to container.
- **Internal Interfaces:**
    - Communication between microservices, specifically:
    - Auth Service's token validation endpoint (/api/auth/validate)
    - Inter-service API key authentication (zero-trust model)
    - Service-to-service HTTP requests (Game Service → Card Service, all services → Logs Service)
- **Database Layer:**
    - PostgreSQL connection on port 5432 (exposed for external tools but should only be accessible within Docker network)
    - Database credentials stored in environment variables
- **Environment and Configuration:**
    - Docker Compose configuration and container security
    - Environment variables (JWT_SECRET_KEY, GAME_HISTORY_KEY, API keys, DATABASE_URL)
    - Host filesystem storage of secrets:
        - .env file containing all encryption keys (GAME_HISTORY_KEY), service API keys (5 keys for zero-trust), JWT secret
        - nginx/ssl/ directory containing SSL private keys and certificates
        - File permission vulnerabilities on host OS
        - Backup and snapshot exposure risk
        - Host OS compromise impacts all stored secrets
- **CI/CD Pipeline:**

- Build process and third-party dependencies
- GitHub Actions workflows (if applicable)

4) Identifying Trust Boundaries (TB)

- **TB1**: Internet ↔ API Gateway - External untrusted clients to system entry point
- **TB2**: API Gateway ↔ Microservices - Gateway routes to internal services (JWT validation required)
- **TB3**: Microservices ↔ Microservices - Inter-service communication (zero-trust with API keys)
- **TB4**: Microservices ↔ Database - Services to PostgreSQL data persistence
- **TB5**: Auth Service ↔ Other Services - Centralized token validation authority

5) Identifying Threats (STRIDE)

Following is the descriptions of threats grouped by STRIDE, followed by the asset and the trust boundaries.

Spoofing Identity (S)

- S1: Attacker impersonates legitimate user by stealing JWT access/refresh tokens (A2; TB1, TB2)
- S2: Malicious service spoofs another microservice without valid API key (inter-service) (A3; TB3)
- S3: Man-in-the-middle attack on HTTP traffic (before HTTPS redirect) (A2,A4; TB1)
- S4: Session hijacking via stolen session tokens (A2; TB1,TB5)
- S5: SSL private key theft from host file system enabling man-in-the-middle attacks (A3; TB1, new TB6: Host Filesystem)

Tampering (T)

- T1: SQL injection through user inputs (registration, login, game parameters)(A1,A4; TB1, TB4)
- T2: Modification of game history in database (bypassing HMAC verification)(A5; TB4)
- T3: JWT token modification to escalate privileges or extend expiration. (A2; TB1,TB2)
- T4: Tampering with game state during gameplay to cheat(A4; TB1, TB2)
- T5: Environment variable manipulation in Docker containers(A3,A6; TB4)
- T6: Tampering with .env file to modify GAME_HISTORY_KEY or service API keys, bypassing encryption/authentication (A3, A5; new TB6: Host Filesystem)
- T7: SSL certificate/key replacement on host file system to enable persistent MITM attacks (A3; TB1, new TB6: Host Filesystem)

Repudiation (R)

- R1: User denies performing actions (insufficient logging)(A1,A5; TB1,TB2)
- R2: Admin denies unauthorized access to sensitive operations(A1; TB2,TB5)
- R3: Lack of audit trail for game history modifications(A5; TB4)

Information Disclosure (I)

- I1: Exposure of JWT_SECRET_KEY or GAME_HISTORY_KEY in logs/repositories/or .env file on host file system (A3; TB3, new TB6: Host Filesystem)
- I2: Database credentials leaked via environment variables(A3; TB4)
- I3: Sensitive user data (passwords, tokens) exposed in API responses(A1,A2; TB1,TB2)
- I4: Encrypted game history decrypted if key is compromised (A3,A5; TB4)
- I5: Service API keys exposed, allowing unauthorized inter-service access (A3; TB3)
- I6: Error me sages revealing system architecture or stack traces (A6; TB1,TB2)
- I7: SSL private key (RSA 4096-bit) exposed from host file system (battlecards.key) (A3; new TB6: Host Filesystem)
- I8: .env file exposure via insecure backups, snapshots, or file permission misconfigurations revealing all secrets (A3; new TB6: Host Filesystem)

Denial of Service (D)

- D1: Brute-force attacks on login endpoint overwhelming Auth Service (A7; TB1)
- D2: Resource exhaustion via unlimited game creation requests(A7; TB1,TB2)
- D3: Database connection pool exhaustion from malicious queries(A7; TB4)
- D4: DDoS attack on API Gateway (port 8080/8443) (A7; TB1)
- D5: Docker container resource starvation (A7; TB4)

Elevation of Privilege (E)

- E1: Regular user gains admin privileges by manipulating JWT claims (A2; TB2, TB5)
- E2: Bypassing authentication to access protected endpoints directly (A1,A2,A4; TB1,TB2)
- E3: Exploiting service-to-service trust to access unauthorized data (A3,A4; TB3)
- E4: Container escape to access host system or other containers (A6; TB4)
- E5: Database privilege escalation via SQL injection (A1,A4; TB4)
- E6: Host file system compromise providing access to all encryption keys, service API keys, and SSL private keys, enabling full system compromise (A3; new TB6: Host Filesystem)

6) Mitigating Threats (Mitigation)

Authentication and Sessions (S1, S4, D1, E1, E2)

- JWT tokens with 24-hour expiration; refresh tokens for renewal tracked in database.
- Bcrypt password hashing with salt; account lockout after 3 failed attempts (15-minute duration).
- Centralized token validation via Auth Service /api/auth/validate endpoint.
- Concurrent session management: strict one-session-per-user enforcement.
- User action logging via Logs Service for authentication and security events.

Authorization and Access Control (S2, E2, E3, T4)

- Bearer token required on all protected endpoints; deny by default.
- Owner-scoped queries to prevent IDOR (e.g., user_id = session.user_id for profiles/games).
- Zero-trust inter-service authentication using API keys (AUTH_SERVICE_API_KEY, etc.).
- Service-to-service requests validated; no implicit trust between microservices.

Input Validation and Injection Prevention (T1, T3, D2)

- Comprehensive sanitization against SQL injection, XSS, command injection, path traversal.
- Input length limits: 50 chars username, 1000 chars JSON payloads.
- Server-side validation for card types, usernames, game IDs, and all user inputs.

Data Protection and Integrity (T2, I1, I2, I3, I4, I5)

- - HTTPS/TLS for all communication (ports 8080→8443 redirect); RSA 4096-bit self-signed certificates (when using build-and-start.sh) stored on host file system with volume mount to nginx container.
- - Game history encryption using GAME_HISTORY_KEY with HMAC integrity verification; returns 409 on tampering.
- - Secrets stored in environment variables (.env gitignored); no hardcoded credentials. When using build-and-start.sh, .env file persists on host file system containing: GAME_HISTORY_KEY, 5 service API keys, JWT secret, database credentials.
- - Generic error messages; no stack traces or system details exposed to clients.

Game Logic and Business Rules (T4, R1, R3)

- Server-side game state management; clients cannot directly manipulate results.
- Game history archived by backend only; audit trail via Logs Service for all game actions.
- HMAC verification prevents post-game tampering of archived match data.

Infrastructure and Container Security (E4, T5, D3, D4, D5)

- All containers run in non-root user mode to prevent host privilege escalation.
- Docker network isolation; services not directly exposed externally (gateway-only access).
- Database connection pooling with limits to prevent exhaustion.
- Volume mount security: SSL certificates and .env secrets mounted as read-only where possible; container-to-host file access minimized.
- Host security hardening: Deployment host requires proper file permissions, user access controls, and regular security updates to protect persisted secrets.

Development and CI/CD (I1, I6, E4)

- Static analysis (Bandit) and dependency scanning (pip-audit) in CI/CD pipeline.
- Docker image vulnerability scanning; no critical/high-severity vulnerabilities deployed.
- Automated security testing in GitHub Actions workflow.

Host Filesystem Security (S5, T6, T7, I7, I8, E6)

- SSL private keys (RSA 4096-bit) stored at ssl with restricted file permissions (recommended: 600 for keys, 644 for certificates).
- .env file containing all secrets stored in .env with restricted permissions (recommended: 600).
- .env file gitignored to prevent version control exposure.
- Host OS security relies on proper user access controls, file system permissions, and OS hardening.
- Backup security: Deployment hosts should exclude .env and nginx/ssl/ from automated backups or use encrypted backup solutions.
- Single point of failure: Host file system compromise exposes all encryption keys (GAME_HISTORY_KEY), service API keys (5 keys), SSL private keys, potentially JWT secret and database credentials.
- Production recommendation: Use secret management services (AWS Secrets Manager, HashiCorp Vault, Azure Key Vault) instead of file system storage for production deployments.
- Certificate rotation: SSL certificates valid for 365 days; manual rotation required before expiration.

Use of Generative AI

Overview

Generative AI tools were used throughout the development process to accelerate implementation, improve code quality, and enhance documentation. We maintained a transparent approach, using AI as an assistant rather than a replacement for understanding and decision-making.

AI Tools Used

Tool	Primary Use Cases	Frequency
ChatGPT	Code generation, debugging, architecture suggestions	Medium
GitHub Copilot	Code completion, generation	High

Specific Use Cases

1. Architecture Design

How AI was used:

- Generated initial microservice structure suggestions based on game requirements
- Provided feedback on database schema design

Human contribution:

- Final architecture decisions based on project requirements
- Selection of appropriate patterns for our scale
- Docker Compose configuration

2. Code Implementation

How AI was used:

- Generated boilerplate Flask route handlers
- Suggested implementations for JWT authentication middleware
- Helped implement InputSanitizer class with regex patterns
- Created Fernet encryption/decryption helper functions

Human contribution:

- Code review and testing of all AI-generated code
- Integration of generated code into existing codebase
- Customization for project-specific requirements
- Bug fixes and security hardening

3. Security Implementation

How AI was used:

- Suggested bcrypt configuration for password hashing
- Provided JWT token structure and validation examples
- Generated HMAC integrity check implementation
- Recommended input sanitization patterns

Human contribution:

- Security requirement analysis
- Selection of appropriate encryption libraries
- Configuration of security parameters (work factors, key sizes)
- Testing of security mechanisms
- Account lockout logic implementation

4. Testing

How AI was used:

- Generated Postman collection templates
- Suggested test case scenarios
- Helped write Locust performance test scripts
- Provided mocking examples for unit tests

Human contribution:

- Definition of comprehensive test coverage requirements
- Creation of realistic test scenarios
- Integration test orchestration
- Analysis of test results

5. Documentation

How AI was used:

- Generated OpenAPI/Swagger documentation structure
- Helped draft README instructions
- Suggested improvements to code comments
- Assisted with report section writing

Human contribution:

- Technical accuracy verification
- Project-specific details and context
- Screenshots and diagrams
- Final report structure and organization

Honest Assessment

What We Could Not Have Done Without AI

- Rapid prototyping of microservice templates
- Quick implementation of standard security patterns (JWT, bcrypt)
- Generation of comprehensive tests

Time Saved vs. Learning Trade-offs

Estimated time saved: ~40-50% on implementation and documentation

Conclusion

Generative AI significantly accelerated our development process while allowing us to maintain a deep understanding of the system. We used AI as an intelligent assistant for routine tasks and initial implementations, which freed time for higher-level problem-solving, security analysis, and thorough testing. This led to improved productivity, faster prototyping, and clearer documentation.

However, we also observed limitations when using AI. Some generated solutions required careful validation, as they could be incomplete, overly generic, or not fully aligned with the specific project context. This made human oversight essential, particularly for design decisions, security considerations, and edge cases. Overall, the combination of AI assistance and human judgment resulted in a more robust and well-documented project than we could have achieved alone within the given timeframe.

Additional Features

Feature Category 1: Security Enhancements (Beyond Requirements)

1.1 Account Lockout with Live Countdown

What is this feature? An automatic security system that locks user accounts after 3 consecutive failed login attempts for 15 minutes, with a live countdown timer displayed to users showing exactly when they can retry.

Why is it useful?

- **Prevents brute force attacks**: Attackers cannot repeatedly guess passwords
- **User-friendly security**: Clear feedback with countdown reduces user frustration
- **Audit trail**: All lockout events are logged for security monitoring

How is it implemented?

Backend Components:

- Auth Service tracks failed login attempts in database
- Columns added to `users` table: `failed_login_attempts`, `account_locked`, `locked_until`
- Algorithm:
  On login failure:
  failed_attempts += 1
  if failed_attempts >= 3:
  account_locked = True
  locked_until = current_time + 15 minutes
  log_security_event("ACCOUNT_LOCKED")
  On login success:
  failed_attempts = 0
  account_locked = False

Frontend Integration:

- JavaScript countdown timer displays remaining lockout time
- Updates every second using `setInterval()`
- Disables login button during lockout period

How is it used?

User Experience:

1. User enters wrong password (attempt 1/3)
2. System responds: "Invalid credentials. 2 attempts remaining."
3. After 3rd failed attempt: "Account locked for 15 minutes due to too many failed attempts."
4. Frontend displays live countdown: "You can try again in 15 min."
5. After timeout or password reset: Account automatically unlocks

API Endpoints:

```
POST /api/auth/login
Response (when locked):
{
  "error": "Account locked",
  "locked_until": "2024-12-12T18:30:00Z",
  "remaining_seconds": 897
}
```

Testing:

- `test_account_lockout.py` makes exactly 3 failed login attempts
- Verifies lockout state in database
- Tests automatic unlock after 15 minutes
- Validates password reset bypasses lockout

1.2 Concurrent Session Control

What is this feature? Strict enforcement of single active sessions per user account—only one device can be logged in at any time, with automatic detection and rejection of simultaneous login attempts.

Why is it useful?

- **Prevents account sharing**: One account cannot be used by multiple players simultaneously
- **Detects compromised accounts**: Administrators are alerted if someone else attempts to log in
- **Game integrity**: Prevents multi-device cheating strategies
- **Session transparency**: Users can see active device and browser information

How is it implemented?

Session Tracking:

- Each login generates unique `session_id` (UUID v4)
- Stored in database table: `CREATE TABLE sessions ( session_id VARCHAR(36) PRIMARY KEY, user_id INTEGER REFERENCES users(id), user_agent TEXT, ip_address VARCHAR(45), created_at TIMESTAMP DEFAULT NOW(), last_activity TIMESTAMP DEFAULT NOW());`
- `session_id` included in JWT payload for validation

Concurrent Detection:

- On each authenticated request:
  - Extract `session_id` from JWT token
  - Query database for active session
  - If `session_id` doesn't match: Force logout with HTTP 401
  - Update `last_activity` timestamp

Device & IP Tracking:

- User-Agent header parsed to identify browser/device
- Client IP address logged for security audit
- Displayed to users in session management interface

How is it used?

User Experience:

1. User logs in on Desktop (Chrome) → Session A created
2. User attempts login on Mobile (Safari) → Rejected with message:
   a. "Account already logged in from another device (Chrome on Windows)"
   b. "Last activity: 2 minutes ago from IP: 192.168.1.10"
3. User can view active session via Session Management API

API Endpoints:

```
GET /api/auth/sessions
Response:
{
  "active_session": {
    "session_id": "abc-123",
    "device": "Chrome 120.0 (Windows)",
    "ip_address": "192.168.1.10",
```

```
    "created_at": "2024-12-12T17:00:00Z",
    "last_activity": "2024-12-12T17:45:00Z"
  }
}

DELETE /api/auth/sessions/{session_id}
→ Manually logout specific session
```

Testing:

- `test_concurrent_sessions.py` simulates multiple browsers using different User-Agent headers (using the session_id from the JWT)
- Verifies rejection of concurrent login attempts
- Tests session information accuracy

1.3 Comprehensive Security Logging

What is this feature? Centralized audit logging of all security-relevant actions through dedicated Logs Service, providing administrators with complete visibility into authentication events, security incidents, and user activities.

Why is it useful?

- **Forensic analysis**: Track security incidents and suspicious activities
- **Compliance**: Audit trails for security requirements
- **Monitoring**: Real-time detection of attacks (brute force, session hijacking)
- **Accountability**: Complete record of who did what and when

How is it implemented?

Logs Service:

- Dedicated microservice for receiving and storing security events
- Database table structure: `CREATE TABLE security_logs ( id SERIAL PRIMARY KEY, user_id INTEGER, username VARCHAR(50), event_type VARCHAR(50), ip_address VARCHAR(45), user_agent TEXT, details JSONB, timestamp TIMESTAMP DEFAULT NOW());`

Logged Events:

Event Type	Trigger	Details Captured

LOGIN_SUCCESS	Successful authentication	username, IP, device
LOGIN_FAILURE	Wrong password	username, IP, attempt count
ACCOUNT_LOCKED	3 failed attempts	username, IP, lock duration
SESSION_CREATED	New login	session_id, device, IP
SESSION_ENDED	Logout or expiry	session_id, duration
CONCURRENT_ATTEMPT	Second login attempt	existing device, new device
PASSWORD_CHANGED	Profile update	username, IP
TOKEN_REFRESHED	Access token renewal	username, session_id

Service Integration:

```
# In Auth Service
def log_security_event(event_type, user_id, details):
    requests.post('http://logs-service:5005/api/logs/security', json={
        'event_type': event_type,
        'user_id': user_id,
        'username': details.get('username'),
        'ip_address': request.remote_addr,
        'user_agent': request.headers.get('User-Agent'),
        'details': details
    })
```

How is it used?

Admin Monitoring:

```
GET /api/logs/security?user=alice&event_type=LOGIN_FAILURE
Authorization: Bearer {ADMIN_TOKEN}

Response:
{
    "event_type": "LOGIN_FAILURE",
    "username": "alice",
    "ip_address": "192.168.1.100",
    "timestamp": "2024-12-12T17:30:00Z",
    "details": {"attempt": 1, "reason": "invalid_password"}
  },
```

Security Analysis Use Cases:

- Detect brute force attacks: Multiple LOGIN_FAILURE events from same IP

- Track compromised accounts: LOGIN_SUCCESS from unusual locations
- Monitor lockout effectiveness: Ratio of ACCOUNT_LOCKED to LOGIN_FAILURE
- Session analysis: Average session duration, device distribution

1.4 Session Management API

What is this feature? RESTful API endpoints allowing users to view active sessions across all devices and manually revoke sessions for enhanced security control.

Why is it useful?

- **User control**: Ability to remotely logout from other devices
- **Security awareness**: Users can see if unauthorized access occurred
- **Lost device protection**: Can revoke access from stolen/lost devices
- **Transparency**: Clear visibility into account activity

How is it implemented?

API Endpoints:

Endpoint	Method	Description
`/api/auth/sessions`	GET	View current active session
`/api/auth/sessions/all`	GET	List all recent sessions (last 30 days)
`/api/auth/sessions/{id}`	DELETE	Revoke specific session
`/api/auth/sessions/all`	DELETE	Logout from all devices

Database Schema:

```
CREATE TABLE sessions (
    session_id VARCHAR(36) PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    user_agent TEXT,
    ip_address VARCHAR(45),
    created_at TIMESTAMP,
    last_activity TIMESTAMP,
    revoked BOOLEAN DEFAULT FALSE,
    revoked_at TIMESTAMP NULL
);
```

Implementation Logic:

```
@app.route('/api/auth/sessions/<session_id>', methods=['DELETE'])
@jwt_required()
```

```
def revoke_session(session_id):
    current_user = get_jwt_identity()

    # Verify session belongs to current user
    session = db.query("SELECT * FROM sessions WHERE session_id = ? AND
user_id = ?",
                       [session_id, current_user.id])

    if not session:
        return {"error": "Session not found"}, 404

    # Revoke session
    db.update("UPDATE sessions SET revoked = TRUE, revoked_at = NOW()
WHERE session_id = ?",
              [session_id])

    log_security_event("SESSION_REVOKED", current_user.id, {"session_id":
session_id})

    return {"message": "Session revoked successfully"}, 200
```

How is it used?

Frontend UI:

```
═══════════════════════════════════════
Active Sessions
───────────────────────────────────────

Current Session:
  Device: Chrome 120.0 (Windows)
  IP: 192.168.1.10
  Last Activity: 2 minutes ago
  [This Device]

Recent Sessions (Inactive):
  Device: Safari 17.0 (iPhone)
  IP: 192.168.1.50
  Last Activity: 3 hours ago
  [Revoke]

  Device: Firefox 121.0 (Linux)
  IP: 192.168.1.25
```

```
  Last Activity: 2 days ago
  [Revoke]

[Logout All Devices]
═══════════════════════════════════════
```

Use Case Example:

1. User notices unfamiliar session in list
2. User changes password for additional security

Feature Category 2: Frontend and User Experience

2.1 Browser-Independent Multiplayer

What is this feature? True real-time multiplayer gameplay where two players can compete simultaneously from different browsers, devices, or locations without page refreshes or polling delays.

Why is it useful?

- **Authentic multiplayer**: Players interact in real-time like traditional card games
- **No installations**: Works in any modern browser without plugins

How is it implemented?

Technical Architecture:

- RESTful API with polling for game state updates (HTTP-based, no WebSockets needed)
- Each player has independent frontend session
- Game state stored server-side in database
- Frontend polls game state every 2-3 seconds during active rounds

Game State Synchronization:

```
// Player 1 plays card
POST /api/games/{game_id}/play-card
→ Updates game state in database

// Player 2's frontend polls for updates
setInterval(async () => {
```

```
  const gameState = await fetch(`/api/games/${game_id}`);
  if (gameState.both_players_ready) {
    displayResolveButton();
  }
}, 2000);

// Either player resolves round
POST /api/games/{game_id}/resolve-round
→ Calculates winner, updates scores

// Both players' frontends receive updated state
GET /api/games/{game_id}
→ Shows round result, updates score display
```

How is it used?

Gameplay Flow:

1. Player 1 creates game, invites Player 2 (by username)
2. Both players log in from separate browsers
3. Each player selects their 22-card deck independently
4. For each round:
    a. Both draw 3-card hands
    b. Both select and play one card
    c. Either player can resolve round
    d. Both see updated scores automatically
5. Game ends after 7 or 8 rounds, winner declared

Technical Requirements:

- Two browsers/devices with network access to server
- Valid user accounts for both players
- Modern browser supporting Fetch API

2.2 Live Feedback & Real-Time Updates

What is this feature? Dynamic user interface that provides instant visual feedback for all user actions, game state changes, and system events without requiring page refreshes.

Why is it useful?

- **User confidence**: Immediate confirmation that actions were registered

- **Error prevention**: Warnings before invalid actions
- **Engagement**: Animated feedback makes gameplay more satisfying
- **Status awareness**: Always know current game state and turn status

How is it implemented?

Countdown Timers:

```
// Account lockout countdown
function startLockoutCountdown(locked_until_timestamp) {
  const interval = setInterval(() => {
    const remaining = locked_until_timestamp - Date.now();
    if (remaining <= 0) {
      clearInterval(interval);
      enableLoginButton();
      displayMessage("Account unlocked. You may now log in.");
    } else {
      const minutes = Math.floor(remaining / 60000);
      const seconds = Math.floor((remaining % 60000) / 1000);
      updateDisplay(`Locked for:
${minutes}:${seconds.toString().padStart(2, '0')}`);
    }
  }, 1000);
}
```

Visual Feedback System:

- **Success animations**: Green checkmark, fade-in effects
- **Error indicators**: Red borders, shake animations
- **Loading states**: Spinners during API requests
- **Progress indicators**: Rounds completed, score changes

Real-Time Game State:

```
// Poll game state during active gameplay
async function pollGameState() {
  const response = await fetch(`/api/games/${gameId}`);
  const game = await response.json();

  // Update UI based on state
  updateScoreDisplay(game.scores);
  updateRoundCounter(game.current_round);
```

```
  if (game.status === 'waiting_for_players') {
    showWaitingIndicator();
  } else if (game.status === 'ready_to_resolve') {
    showResolveButton();
  } else if (game.status === 'completed') {
    displayWinner(game.winner);
  }
}
```

How is it used?

User Experience Examples:

Action	Feedback
Click "Play Card"	Card animates to center, "Waiting for opponent" message
Login success	Green checkmark, "Welcome back!" message, auto-redirect
Login failure	Red border on password field, "Invalid credentials" alert
Game created	"Game created! Waiting for opponent..." with spinner
Round won	"+1 Point" animation, score updates with bounce effect
Account locked	Red alert banner with live countdown timer

2.3 Intuitive Navigation & User Flow

What is this feature? Carefully designed navigation structure with clear back buttons, logical page transitions, automatic redirects for authentication, and contextual navigation that adapts to user state.

Why is it useful?

- **Reduced confusion**: Users always know where they are and how to get back
- **Efficiency**: Fewer clicks to complete common tasks
- **Security**: Automatic protection of authenticated pages
- **User satisfaction**: Smooth, predictable interface behaviour

How is it implemented?

Navigation Hierarchy:

```
Login Page
  ↓ (on success)
```

```
Dashboard
  ├→ View Cards
  ├→ Start New Game
  │   ↓
  │   Deck Selection
  │     ↓
  │     Gameplay (7 rounds)
  │       ↓
  │       Game Results
  │         ↓
  │         Leaderboard
  ├→ View My Games
  ├→ Leaderboard
  └→ Profile Settings
```

Protected Routes:

```
// Frontend route guard
function requireAuth(page) {
  const token = localStorage.getItem('access_token');
  if (!token || isTokenExpired(token)) {
    redirectToLogin();
    displayMessage("Please log in to continue");
  } else {
    renderPage(page);
  }
}
```

Breadcrumb Navigation:

- Clear indication of current location
- Click any level to go back
- Example: `Dashboard > New Game > Deck Selection > Round 3`

Contextual Buttons:

- "Back to Game" during active gameplay
- "Back to Dashboard" on results page
- "Return to Login" on lockout page
- "Cancel" on deck selection before confirming

How is it used?

Common User Flows:

1. **New Player Flow:**

```
Landing Page → Register → Dashboard → View Cards → Start Game
```

2. **Returning Player Flow:**

```
Login → Dashboard → Continue Game OR View Leaderboard
```

3. **Mid-Game Navigation:**

```
Gameplay → [Back] → Dashboard → [Resume Game] → Gameplay
```

4. **Error Recovery:**

```
Invalid Action → Error Message → [Retry] OR [Cancel]
```

2.4 Accessibility Features

What is this feature? Inclusive design considerations that ensure the game is playable and enjoyable by users with varying abilities, following WCAG (Web Content Accessibility Guidelines) principles where practical.

Why is it useful?

- **Wider audience**: Makes game accessible to users with visual, motor, or cognitive challenges
- **Better UX for everyone**: Accessibility improvements benefit all users
- **Professional quality**: Demonstrates attention to detail and user-centered design
- **Legal compliance**: Many jurisdictions require digital accessibility

How is it implemented?

Large Tap Targets:

- Minimum button size: 44x44px (Apple) / 48x48px (Google)
- Adequate spacing between clickable elements: 8px minimum
- Card selection areas enlarged for easy tapping

Readable Typography:

- Minimum font size: 16px for body text
- High-contrast text colors (4.5:1 ratio for normal text)
- Clear font choices: system fonts or web-safe fonts
- Scalable text (rem/em units, not px)

Consistent Layouts:

- Predictable button positions across pages
- Consistent color scheme for actions (green = success, red = danger)
- Standard navigation patterns throughout application

Semantic HTML:

```
<!-- Proper heading hierarchy -->
<h1>Battle Card Game</h1>
<h2>Current Game</h2>
<h3>Round 3 of 7</h3>

<!-- Meaningful button labels -->
<button aria-label="Play Rock card with power 7">
  <img src="rock-7.png" alt="Rock card, power 7">
</button>

<!-- Form labels -->
<label for="username">Username:</label>
<input id="username" type="text" required>
```

Keyboard Navigation:

- Tab order follows logical flow
- Focus indicators visible on all interactive elements
- Enter key submits forms
- Escape key closes modals

How is it used?

Accessibility Features in Action:

Feature	Implementation	Benefit
Alt text for cards	`<img alt="Rock-7">`	Screen readers describe cards

High contrast mode	Dark text on light backgrounds	Readable for vision impaired
Large buttons	48x48px minimum	Easier to tap for motor disabilities
Clear error messages	"Password must be 8+ characters"	Helps users recover from errors
Keyboard shortcuts	Tab, Enter, Escape	Operable without mouse

Summary of Additional Features

Feature Category	Features	Complexity	Impact
Security Enhancements	Account Lockout, Concurrent Sessions, Security Logging, Session Management	High	Critical for production-ready application
Frontend/UX	Browser Multiplayer, Live Feedback, Intuitive Navigation, Accessibility	Medium	Essential for user satisfaction

Justification: All features are **fully implemented, tested, and integrated** with both frontend and backend. They go significantly beyond the base requirements and demonstrate production-quality engineering practices, security awareness, and user-centered design principles.