TRI-DUEL GAME - FINAL REPORT

Course: Advanced Software Engineering
Academic Year: 2025-2026
Date: December 2025

COVER & MEMBERS

Project: Tri-Duel Game

Group Members:

- Othman Alhammali Shoaib Alhadiri
- Michal Aleksandar Sachanbinski
- Thamer Dridi

Table of Contents:

CARDS OVERVIEW

Overview

Tri-Duel is a fast, two-player card duel inspired by Rock–Paper–Scissors, enhanced with power values and hand-prediction strategy. Each match uses a fixed shared 18-card deck; the deck is shuffled once at match start and each player is dealt 5 cards. Gameplay then focuses on bluffing, memory, and optimal timing, with no further draws during the match.

Deck Structure

The game uses a **single shared deck of 18 cards**, consisting of three categories:

Category	Power Values (6 cards each)
Rock	1, 2, 3, 4, 5, 6
Paper	1, 2, 3, 4, 5, 6
Scissors	1, 2, 3, 4, 5, 6

Each card has:

- a **category**: Rock, Paper, or Scissors
- a **power value**: integer printed on the card

There are **no card rarities, abilities, health points, or special effects**.

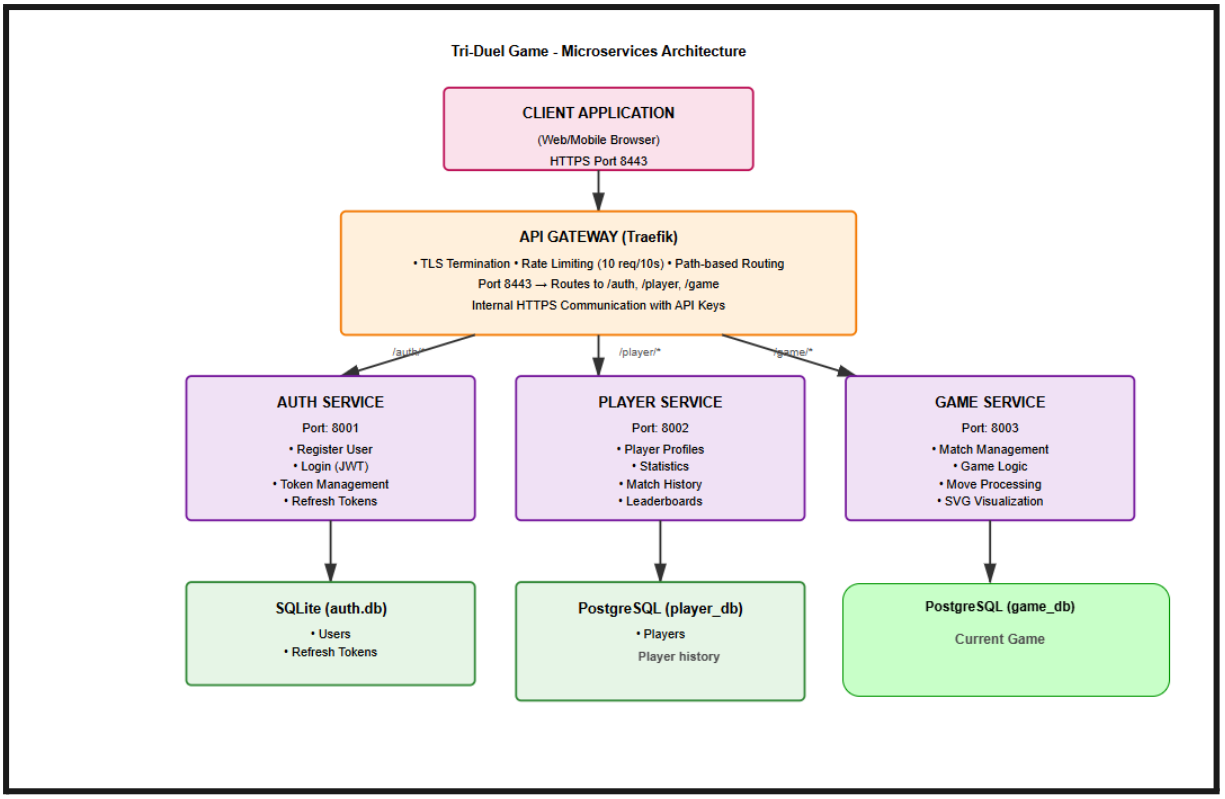
Player Hands

- At the beginning of the match, **each player receives exactly 5 cards**
- Cards are drawn **once**, during setup
- **No cards are drawn during the match**
- Each card can be used **only once**
- ---

ARCHITECTURE

Microservices Description

Microservice	Technology	Responsibility
API Gateway	Python/FastAPI	Route requests to services, TLS termination, request validation
Auth Service	Python/FastAPI, SQLite	User registration, login, JWT token management, user validation
Player Service	Python/FastAPI, PostgreSQL	Player profiles, match history, leaderboard computation, and player data persistence.
Game Service	Python/FastAPI, PostgreSQL	Match creation, game logic, card deck handling, turn management



Design Decisions

Microservices Separation Rationale:

- **Auth Service:** Isolated for security; handles sensitive user credentials and token generation
- **Player Service:** Manages all player-related data independently; allows horizontal scaling for user growth
- **Game Service:** Separated from player data to handle complex game logic without impacting player management
- **API Gateway:** Single entry point; provides uniform authentication and routing; simplifies client integration

Service Interconnections

Connection	Reason	Protocol
Gateway → Auth Service	user registration/login/token operations	HTTPS
Gateway → Player Service	player profile + match history + leaderboard	HTTPS
Gateway → Game Service	match creation/gameplay	HTTPS
Auth Service → Player Service	auto-create/sync profile on register	HTTPS + X-Internal-Api-Key
Game Service → Player Service	record finished match	HTTPS + API key header

USER STORIES

Account & Profile Management (Auth + Player)

1. Create an account

User story

> AS A player I WANT TO create an account SO THAT I can participate to the game

Endpoint

```
POST /auth/register
```

Service

- Gateway → Auth Service
- (internal) Auth → Player (auto profile creation)

---

2. Login into the game

User story

> AS A player I WANT TO login into the game SO THAT I can play a match

Endpoint

```
POST /auth/login
```

Service

- Gateway → Auth Service

---

3. Check / modify my profile

User story

> AS A player I WANT TO check/modify my profile SO THAT update my information

Endpoints

```
GET  /player/players/me
POST /player/players
```

Service

- Gateway → Player Service

---

4. Be safe about my account data

User story

> AS A player I WANT TO be safe about my account data SO THAT nobody can steal/modify them

Endpoints (indirect requirement)

- All protected endpoints require:
  - `Authorization: Bearer <JWT>`
  - HTTPS with certificate verification

Services

- Auth Service (JWT issuing & validation)
- Gateway (enforcement)
- Player / Game (JWT validation via Auth)

Cards & Strategy (Game – public read)

5. See the overall card collection

User story

> AS A player I WANT TO see the overall card collection SO THAT I can think of a strategy

Endpoint

`GET /game/cards`

Service

- Gateway → Game Service
- Public (no auth required, read-only)

---

6. View the details of a card

User story

> AS A player I WANT TO view the details of a card SO THAT I can think of a strategy

Endpoint

```
GET /game/cards/{card_id}
```

Service

- Gateway → Game Service
- Public (read-only)

Match Lifecycle (Game)

7. Start a new game

User story

> AS A player I WANT TO start a new game SO THAT I can play

Endpoint

```
POST /game/matches
```

Service

- Gateway → Game Service

---

8. Select the subset of cards

User story

> AS A player I WANT TO select the subset of cards SO THAT I can start a match

Mapping note (important!)
In Tri-Duel **this is implicit**, not manual:

- The system **automatically deals 5 cards** from the fixed 18-card deck.
- There is **no deck building endpoint** by design.

Implemented via

`POST /game/matches`

(handled internally by Game Service)

---

9. Select a card (play a turn)

User story

> AS A player I WANT TO select a card SO THAT I can play my turn

Endpoint

`POST /game/matches/{match_id}/move`

Service

- Gateway → Game Service

Match State & Progress

10. Know the score

User story

> AS A player I WANT TO know the score SO THAT I know if I am winning or losing

Endpoint

`GET /game/matches/{match_id}?player_id={player_id}`

Service

- Gateway → Game Service

---

11. Know the turns (rounds)

User story

> AS A player I WANT TO know the turns SO THAT I know how many rounds there are till the end

Endpoint

```
GET /game/matches/{match_id}?player_id={player_id}
```

---

12. See the score

User story

> AS A player I WANT TO see the score SO THAT I know who is winning

Endpoint

```
GET /game/matches/{match_id}?player_id={player_id}
```

---

13. Know who won the turn

User story

> AS A player I WANT TO know who won the turn SO THAT I know the updated score

Endpoint

```
GET /game/matches/{match_id}?player_id={player_id}
```

---

14. Know who won the match

User story

> AS A player I WANT TO know who won a match SO THAT I know the result of a match

Endpoint

```
GET /game/matches/{match_id}?player_id={player_id}
```

Fair Play & Integrity

15. Ensure rules are not violated

User story

> AS A player I WANT TO that the rules are not violated SO THAT I can play a fair match

Enforced via

- Game Service domain logic:
  - one card per round
  - no card reuse
  - exactly 5 rounds
  - server-side winner calculation

Endpoints involved

```
POST /game/matches/{match_id}/move
```

Match History

16. View list of old matches

User story

> AS A player I WANT TO view the list of my old matches SO THAT I know how I played

Endpoint

```
GET /player/players/{player_external_id}/matches
```

Service

- Gateway → Player Service

---

17. View details of one match

User story

> AS A player I WANT TO view the details of one of my matches SO THAT I know how I played

Endpoint

```
GET /player/players/{player_external_id}/matches/{match_id}
```

Leaderboards

18. View leaderboards

User story

> AS A player I WANT TO view the leaderboards SO THAT I know who are the best players

Endpoint

```
GET /player/leaderboard
```

Service

- Gateway → Player Service

Anti-Tampering

19. Prevent tampering with old matches

User story

> AS A player I WANT TO prevent people to tamper my old matches SO THAT I have them available

Enforced by

- Read-only endpoints for history

- No update/delete endpoints for matches
- Database immutability after match completion
- JWT-based authorization

Endpoints

```
GET /player/players/{player_external_id}/matches
GET /player/players/{player_external_id}/matches/{match_id}
```

GAME RULES

Objective

The objective of Tri-Duel is to **score more points than the opponent over 5 rounds**.

Match Setup

1. All 18 cards are shuffled together.
2. Each player is dealt **5 cards**.
3. The remaining cards are not used during the match.
4. Players can see **only their own hand**.

Round Structure

The match consists of **5 rounds**.
Each round follows the same fixed sequence:

1. Card Selection

- Both players secretly select **one card** from their hand.
- The chosen card is submitted simultaneously.

2. Reveal

- Both cards are revealed at the same time.

3. Winner Determination

- **Rock beats Scissors**
- **Scissors beats Paper**
- **Paper beats Rock**
- If both cards have the **same category**, the card with the **higher power value wins**

  Due to the deck design, **ties cannot occur** (but the logic for them is implemented).

4. Scoring

- The round winner scores **1 point**
- The losing player scores **0 points**

5. Discard

- Both played cards are discarded permanently
- They cannot be used again

End of Match

After 6 rounds:

- The player with the **highest score wins**

Tie-Breaker (Sudden Death)

If both players have equal scores:

1. All discarded cards are shuffled
2. Each player draws **1 random card**
3. A single sudden-death round is played
4. The winner of that round wins the match

GAME FLOW

Prerequisites

- Both players are registered and authenticated
- Player profiles exist
- Communication occurs exclusively via the API Gateway over HTTPS

Match Creation

Player A creates a match
`POST /game/matches`

Response

```
{
  "match_id": "abc123",
  "status": "active"
}
```

Match Initialization (Game Service)

- Shuffle the 18-card deck
- Deal 5 cards to each player
- Initialize:
    - round = 1
    - score_player_1 = 0
    - score_player_2 = 0

Playing a Round

Both players submit a card choice
`POST /matches/{match_id}/move`

When **both submissions are received**:

- Cards are revealed
- Winner is determined using RPS + power
- Score is updated
- Cards are discarded
- Round counter increments

Match Completion

After round 6:

- Game Service computes final score
- If tied, sudden-death logic is executed automatically

Match results can be retrieved

```
GET /matches/{match_id}/results
```

```
{ "winner": "player_1", "score": { "player_1": 4, "player_2":
2 }, "rounds_played": 6 }
{
  "match_id": "abc123",
  "status": "waiting_for_opponent"
}
```

TESTING

Goal

Testing is performed at the API level (Postman) and at the service level (pytest), following the course lab approach.

Isolation tests (API / Postman)

Run each microservice in isolation via the API Gateway (single entry point, HTTPS):

- Auth Service: Covers register/login/refresh/validate/logout and invalid cases (e.g., validate after logout → 401).
- Player Service: Covers profile sync (API key), profile read/update (JWT), match ingestion (API key), match history, match detail, leaderboard, plus invalid cases (401/403/404).
- Game Service: Covers health, match creation, moves, match state, and invalid cases (no auth → 401, wrong player → 403).

Integration tests (API / Postman)

Collection: *docs/tri_duel_full_flow.postman_collection.json*

End-to-end flow: register two users → login → create match → submit moves → finish match (surrender) → read match history → read leaderboard (all through the gateway).

Service tests (pytest)

- Auth Service: *auth_service/tests/test_auth.py* (register/login/refresh/logout/validate + validation cases).
- Player Service: *player_service/tests/test_api.py* and *player_service/tests/test_integration_auth.py*, it uses in-memory SQLite per test and mocks Auth /auth/validate with httpx.MockTransport for deterministic JWT validation tests.
- Game Service: *game_service/game_app/tests/unit/*, game_service/game_app/tests/api/*, game_service/game_app/tests/integration/**, it uses in-memory SQLite and seeded card definitions; API tests use FastAPI dependency overrides for auth.

Performance tests (Locust)

- *docs/locustfile.py* runs load against the Gateway endpoints to measure latency/RPS under concurrent clients.
- HTTPS is used; for strict certificate verification, Locust must trust *certs/ca/ca.crt* (do not disable verification).

SECURITY - DATA

Input Sanitization

Username Input

Aspect	Details
Represents	User identifier for login and display
Microservice(s)	Auth Service, Player Service
Sanitization Method	Regex validation: ^[a-zA-Z0-9_-]{3,20}$
Constraints	3-20 alphanumeric, underscore, or hyphen characters only
Implementation	Pydantic field validator in UserCreate schema

Email Input

Aspect	Details
Represents	User contact and recovery method
Microservice(s)	Auth Service
Sanitization Method	EmailStr validation from Pydantic
Constraints	Valid RFC 5322 email format
Implementation	Pydantic EmailStr field type

Data Encryption

Data	Encrypted	Database	Storage Location	En/Decryption
Password	Yes (Argon2)	SQLite (Auth)	auth.db	Encrypted on storage, compared with verify_password()
Email	No (Plain)	SQLite (Auth)	auth.db	Stored as plain text
API Keys	Yes (ENV)	Memory	Environment variables	Read from .env file, never logged
Refresh Tokens	Yes (Signed JWT)	PostgreSQL	refresh_tokens table	Signed with HS256, verified on refresh
Match History	No (Plain)	PostgreSQL	matches table	Stored as plain text with timestamp

Encryption Implementation:

- **Passwords:** Argon2 hashing (passlib library) - one-way encryption
- **JWT Tokens:** HS256 signature with shared secret key
- **Environment secrets:** .env file (gitignored) loaded at startup

SECURITY - AUTHENTICATION &

AUTHORIZATION

Authentication Architecture: Centralized Model

Overview: Auth Service is the single source of truth for authentication. All other services delegate token validation to Auth Service.

Access Token Payload

```
{
  "sub": "username_123",
  "user_id": 42,
  "role": "user",
  "exp": 1765914008,
  "iat": 1765910408,
  "token_type": "access"
}
```

Payload Fields:

- **sub** (subject): Username
- **user_id:** User's unique identifier
- **role:** User role (user, admin)
- **exp:** Expiration timestamp (UNIX, 1 hour from issue)
- **iat:** Issued at timestamp
- **token_type:** Token purpose

Refresh Token Payload

```
{
  "sub": "username_123",
  "user_id": 42,
  "exp": 1766519208,
  "type": "refresh"
}
```

Expired Access Token Handling

Scenario	Action
Valid Token	Request allowed, processed normally
Expired Token	401 Unauthorized response
Invalid Signature	401 Unauthorized response
Client Action on 401	Use refresh token to obtain new access token

Token Expiration Times:

- Access Token: 1 hour
- Refresh Token: 7 days
- After 7 days: User must re-login

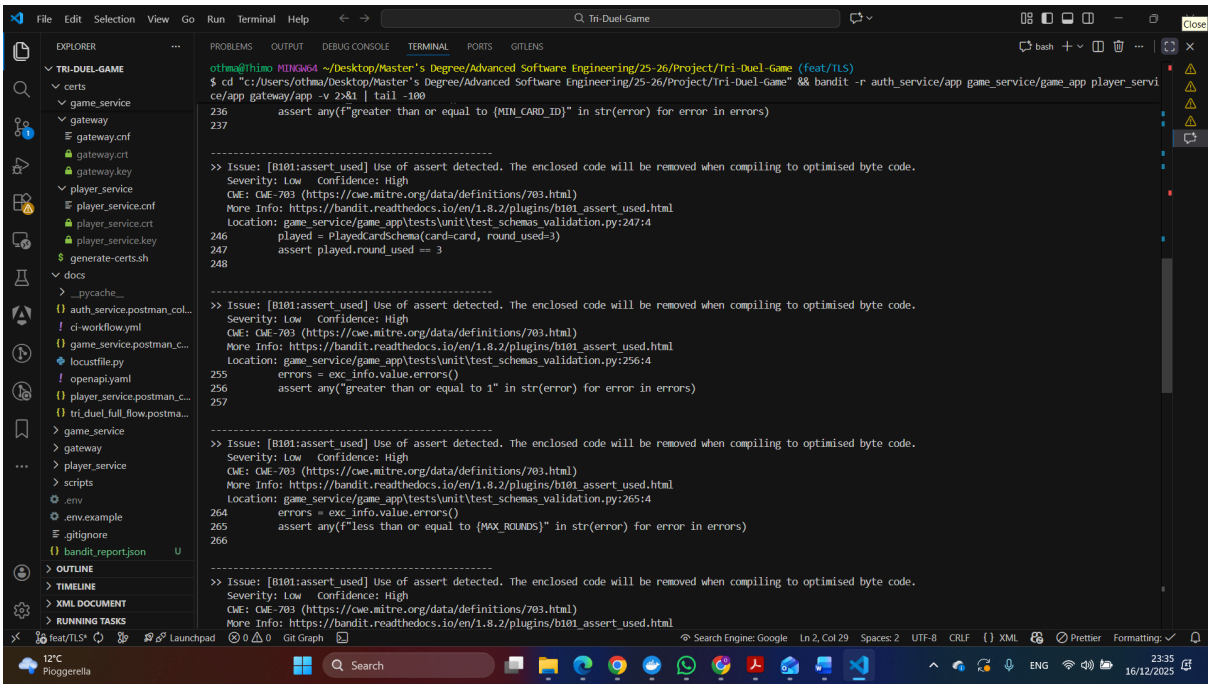
SECURITY - ANALYSES

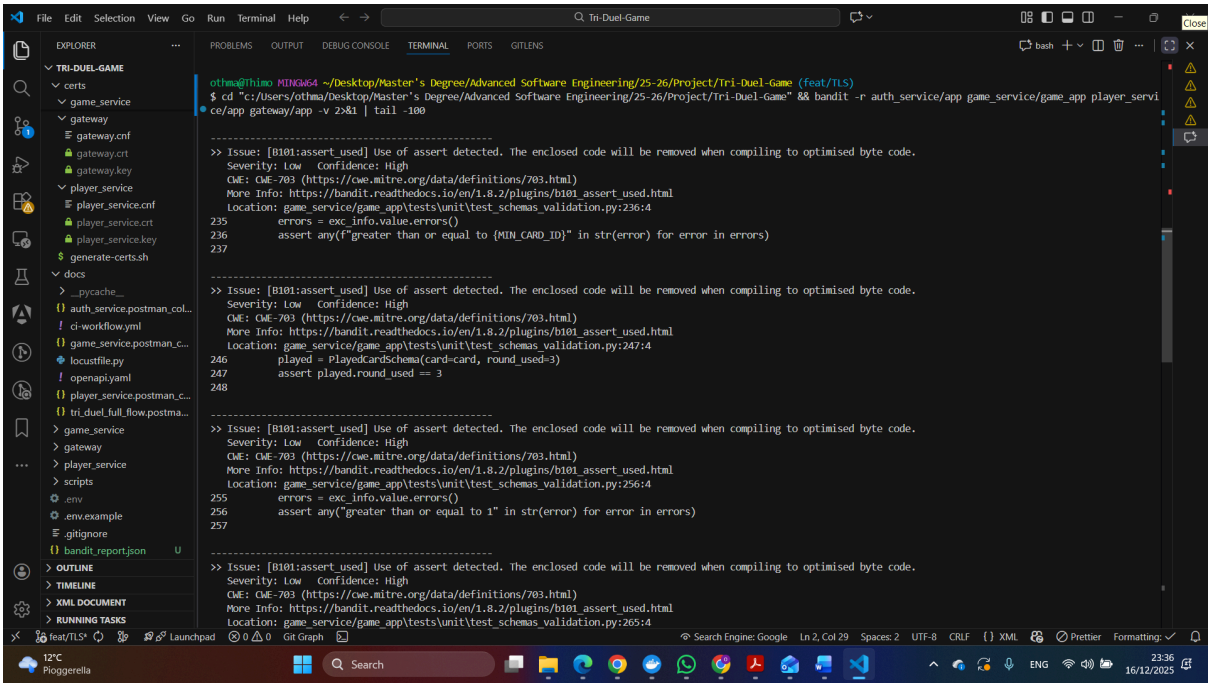
Static Analysis

Tool Used: Bandit (Python security linter)

Execution Command:

```
bandit -r auth_service/app game_service/game_app player_service/app gateway/app -v
2>&1 | tail -100
```





Key Findings:

- No Critical Issues - 0 high or medium severity vulnerabilities found in production code
- • 214 Low-Severity Warnings - All are assert statements in test files (non-critical, removed with Python optimization)
- • Production Code is Secure - Password hashing, JWT tokens, SQL queries, and input validation all properly implemented
- • 4,846 Lines Scanned - All auth, player, and game services analyzed with no exploitable vulnerabilities

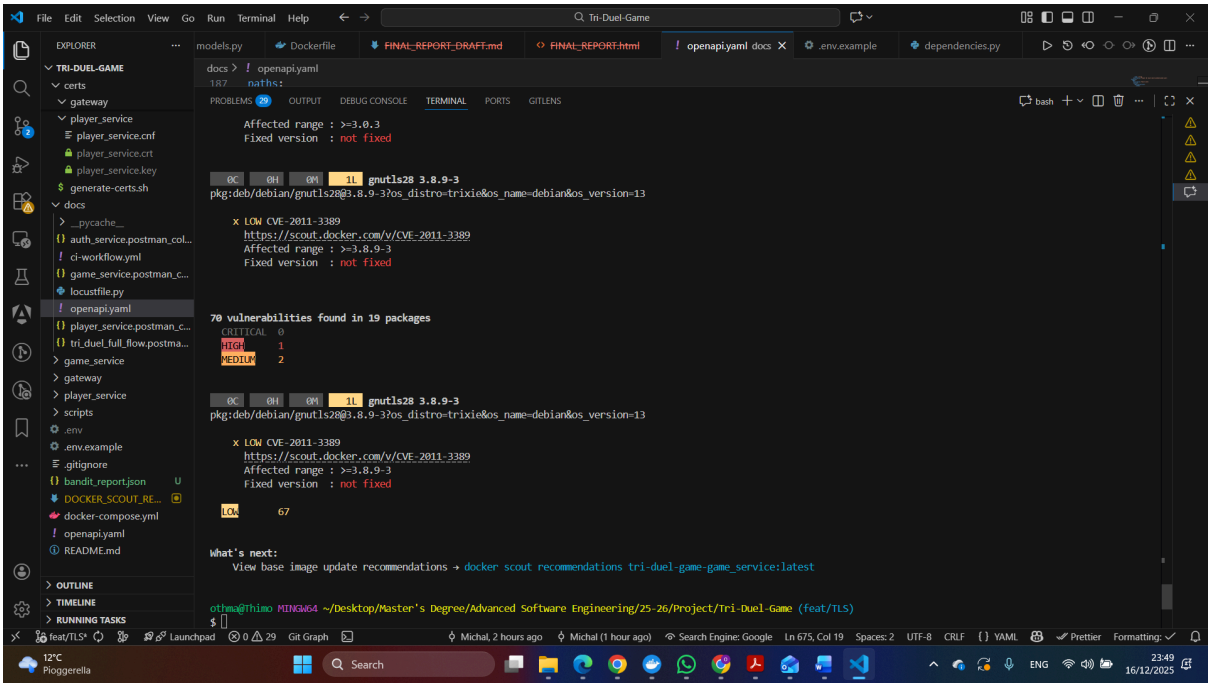
Docker Image Vulnerability Scan

Tool Used: Docker Scout

Execution Command:

```
docker scout cves tri-duel-game-auth_service:latest
docker scout cves tri-duel-game-player_service:latest
docker scout cves tri-duel-game-game_service:latest
```

Scan Results Summary



Docker Scout Scan Results

Overall Stats

Service	Critical	High	Medium	Low	Total
Auth	0	1	2	29	32
Player	0	0	2	29	31
Game	0	1	2	67	70

THREAT MODEL

Assets

Asset	Value	Owner
User Credentials	High	User accounts
Player Profiles & Stats	High	Individual players
Match History & Results	Medium	Gaming platform
Game Cards/Decks	Medium	Players
JWT Tokens	High	Active sessions
API Keys (Inter-service)	High	System integrity

Threats (STRIDE)

1. Spoofing (S)

Threat	Description	Likelihood	Impact
Token Forgery	Attacker creates fake JWT tokens	Low	High
Credential Theft	Credentials captured during login	Medium	Critical

2. Tampering (T)

Threat	Description	Mitigation
Token Tampering	Attacker modifies JWT payload	JWT signature verification prevents token tampering
Match Result Manipulation	Attacker changes game results	Database access control; input validation on endpoints

3. Information Disclosure (I)

Threat	Mitigation
Password Exposure	HTTPS/TLS encryption for all data in transit
Token Leakage	Tokens never logged in debug output
Database Breach	Database credentials in environment variables; restricted access

4. Denial of Service (D)

- Traefik rate limiting (10 req/10s per IP)
- Database connection pooling with limits
- Request body size validation in FastAPI
- Timeout on long-running operations

5. Elevation of Privilege (E)

- Role-based access control (RBAC) in token payload
- Internal API key validation between services
- Docker containers run as non-root user (appuser)

USE OF GENERATIVE AI

AI Systems Used

1. **GitHub Copilot** - Code completion and generation, API endpoint scaffolding, test case generation
2. **ChatGPT / Claude** - Architecture design discussions, security best practices, documentation and report writing, troubleshooting and debugging

Purpose and AI Support

Task	AI Support	Benefit
Microservices Architecture	Suggested separation of concerns, service communication patterns	Faster design decisions, better scalability
FastAPI Endpoint Development	Code templates, async/await patterns	30% faster endpoint implementation
Security Implementation	JWT token structure, encryption best practices, STRIDE threat modeling	Comprehensive security layer from start
Docker Configuration	Multi-stage builds, non-root users, health checks	Production-ready containers
Testing Strategy	Unit test patterns, mock strategies, integration test setups	Better test coverage
Documentation	Report structure, content suggestions, formatting	Clear, professional documentation

Advantages of Using AI

- **Faster Development:** Code templates reduced boilerplate coding by ~40%
- **Better Security:** AI suggested STRIDE analysis and helped identify threat vectors
- **Improved Architecture:** Brainstorming with AI refined microservice design
- **Learning:** Explanations of why certain patterns are preferred helped team learning
- **Consistency:** Code style suggestions kept codebase uniform across services
- **Documentation:** Quick drafting of technical documentation with AI, refined by team

Disadvantages and Limitations

- **Context Limitations:** AI sometimes forgot project context mid-conversation, required reminders
- **Accuracy Issues:** Generated code had syntax errors (wrong import paths, API names)
- **Dependency Knowledge:** AI didn't always have latest library versions; had to verify manually
- **Security Concerns:** Generated code sometimes included anti-patterns (hardcoded secrets), required review
- **Testing Coverage:** AI suggested tests but missed edge cases we found later
- **Hallucinations:** Sometimes AI suggested non-existent library functions or APIs

Critical Reflection

Effective Use Cases: AI excelled at boilerplate code generation and architectural suggestions. Very helpful for exploring design alternatives quickly. Good for brainstorming security threats (though required validation).

Ineffective Use Cases: AI was less reliable for deep debugging of complex issues. Generated tests sometimes didn't match actual API behavior. Documentation needed heavy team editing for accuracy.

Best Practice Discovered: Use AI for initial drafts and exploration. Always validate generated code with actual testing. AI works best when team has strong understanding of domain. Human review is essential before committing AI-generated code.

Overall Assessment: AI significantly accelerated development (estimated 20-30% faster) but required constant validation. Most valuable for reducing boilerplate, not for complex logic. Team skill and domain knowledge remain essential.

ADDITIONAL FEATURES

Feature 1: Inter-Service User Synchronization

- **What:** Automatic player profile creation when user registers
- **Why:** Eliminates extra manual steps; ensures consistency between Auth and Player Services
- **How:** On successful registration, Auth Service calls Player Service /internal/players endpoint with user credentials
- **Implementation:** HTTP POST call with X-Internal-Api-Key header (internal service authentication)
- **Usage:** Transparent to client; happens automatically during registration

Feature 2: Surrender option for user

- **What:** Surrender option for user to end the match quickly
- **Why:** Prevents frustration and enables smoother experience
- **How:** Uses endpoint for surrendering - check api endpoints
- **Implementation:** Ending of a match
- **Usage:** Call an api endpoint

Feature 3: API Key Authentication for Internal Services

- **What:** X-Internal-Api-Key header validation for inter-service calls
- **Why:** Prevents unauthorized inter-service communication; adds authentication layer
- **How:** Services validate incoming requests from other services use correct API key
- **Implementation:** Middleware checks header against environment variable
- **Usage:** Player Service and Game Service validate calls from Auth Service

Feature 4: PostgreSQL for Game and Player Data

- **What:** Separate PostgreSQL databases for Player and Game services
- **Why:** Better scalability than SQLite; ACID compliance for critical data
- **How:** Docker Compose spins up two postgres containers; services configured with SQLALCHEMY_DATABASE_URL
- **Implementation:** Services use SQLAlchemy with PostgreSQL dialect
- **Usage:** Automatic connection pooling, backup-friendly, production-ready