# Carusi Cards Game

## Advanced Software Engineering Lab Project

a.a. 2025/2026

Francesco Alizzi
Andrea Riolo Vinciguerra
Rebecca Rodi
Francesco Scarfato

January 11, 2026

# Contents

# 1 Group and Members

Our group is the *Carusi S.r.l.* group. The development tasks were distributed as follows.

**Francesco Alizzi** Implementation of the Player Service and player profile-related APIs.

**Andrea Riolo Vinciguerra** Implementation of the Authentication Service and Matchmaking Service.

**Rebecca Rodi** Implementation of the Card Catalogue Service and friendship-related APIs in Player Service.

**Francesco Scarfato** Implementation of Game Engine Service.

# 2 Cards Overview

The cards set is based on the twenty regions of Italy.

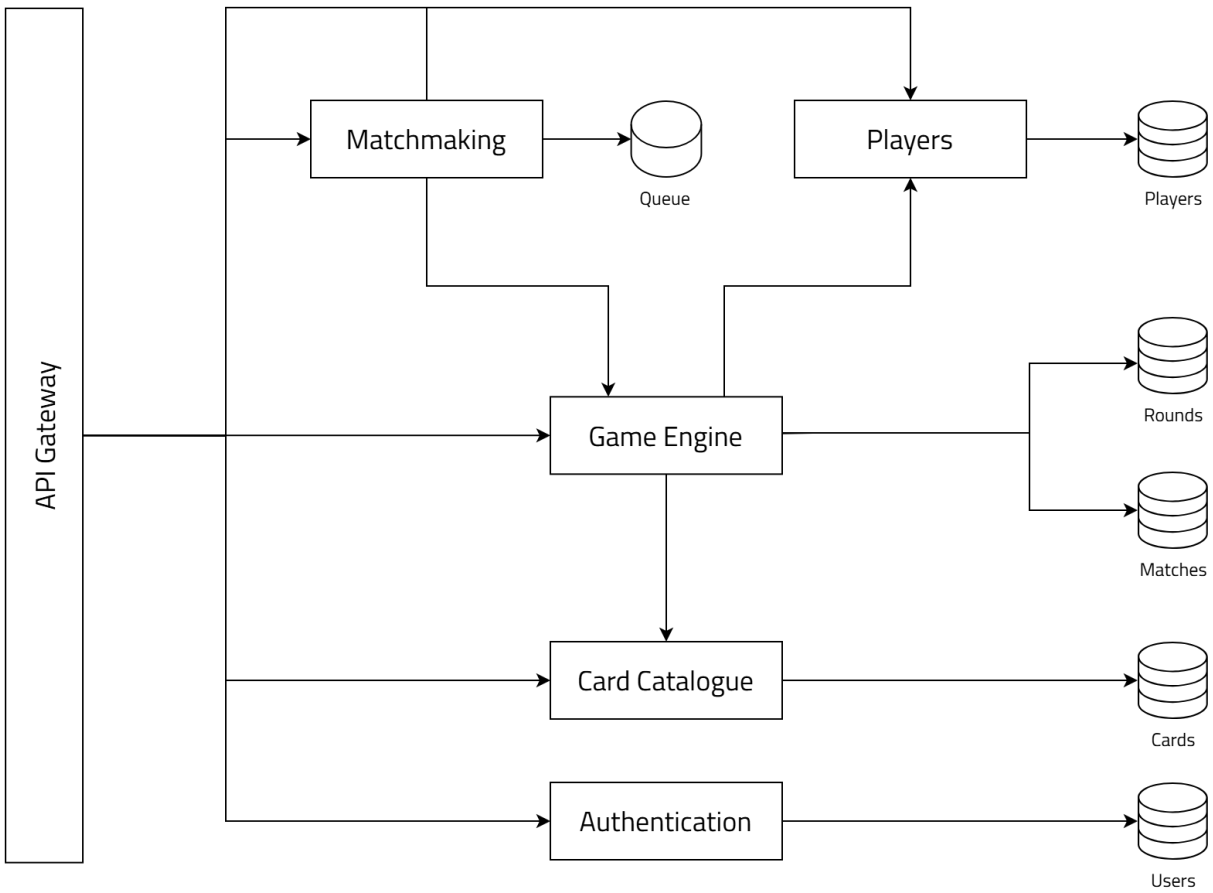
Each card is represented by an image of the region, its total score and the category subscores which it is based upon. The categories are defined as follows.

- *Economy* reflects the financial health and industrial output of the region, primarily based on the PIL (*Prodotto Interno Lordo*, or GDP).
- *Environment* is a comprehensive metric covering both the natural and cultural assets. It accounts for the historical heritage, as well as the density of monuments and the beauty of the regional scenery.
- *Food* represents the richness of the culinary tradition.
- *Special* is a wildcard score assigned to highlight unique regional traits and other characteristics not captured by the other categories.
- *Total* is a composite metric representing the arithmetic mean of the four subscores listed above.

The scores of *Economy*, *Environment* and *Food* were assigned on the basis of real data, meanwhile the *Special* score was assigned according to subjective criteria.


Figure 1: Overview of all cards used in the service.

# 3 Architecture




The distributed system architecture is based on a microservices-oriented decoupled pattern, ensuring modularity and independent scalability of the core game logic. The following table details the functional responsibilities for each service.

| Service | Description |
|---|---|
| **API Gateway** | Serves as the single entry point for all client requests, preventing direct client access to internal services. It routes external requests to the appropriate microservices based on the requested endpoint. |
| **Authentication** | Handles user-related functionalities, provides centralized identity management and secure session persistence via stateless JWT issuance. |
| **Players** | Handles players-related functionalities, such as profile management and friendship features. |
| **Matchmaking** | Manages the player queue in order pair users for game sessions. |
| **Game Engine** | Manages the core game logic, including state transitions and scoring. |
| **Card Catalogue** | Provides an interface for retrieving card data. |

All of the microservices described above are implemented in Python. Nginx is the

only third-party sofware used as the API Gateway.

## 3.1 Design Choices

The architectural design splits data and functionalities among microservices, ensuring that each component serves as the authoritative owner of the data most relevant to its specific domain logic.

- **Authentication Service** manages *Users Database*. Since this service handles authentication and authorization logic, it must own the persistent user profiles to enforce security and access control centrally.
- **Players Service** manages *Players Database*. As this service is responsible for all information related to players, it must own the persistent player profiles and friendship relations to ensure data integrity.
- **Card Catalogue** manages the *Cards Database*. This service acts as interface to the Cards Database and serves as the authoritative source for card data for the entire system, ensuring consistency.
- **Matchmaking Service** manages the *Queue Database*. This service owns the temporary data regarding the players currently in the queue. This state is highly volatile and only relevant until a match is successfully formed.
- **Game Engine** manages the *Matches Database* and *Rounds Database*. This service manages the persistent data related to active and completed matches, as well as individual rounds within those matches, since it is responsible for the core game logic and state transitions.

We decided to split the database used by Authentication Service and Players Service into a Users Database and a Players Database in order to protect users' sensitive data. In fact, the first one contains user credentials, while the second one does not contain any private or confidential data.

## 3.2 Inter-Service Communication

The key service-to-service dependencies are justified as follows.

- **Matchmaking** to **Players**: Matchmaking Service is connected to Players Service in order to ensure data integrity, by validating that the requesting user possesses a player profile. This prevents unauthorized users from entering the queue and start a game.
- **Matchmaking** to **Game Engine**: Matchmaking Service is connected to Game Engine Service in order to orchestrate the transition from the queue to an active game session by triggering a match initialization, after pairing two enqueued players.
- **Game Engine** to **Players**: Game Engine Service is connected to Players Service in order to validate permissions. This ensures match history is only shared between players with a friendship relation, thereby enforcing data privacy.

- **Game Engine** to **Card Catalogue**: Game Engine Service is connected to Card Catalogue in order to validate deck submissions against the authoritative card registry. This prevents invalid or tampered cards from being used in matches, assuring fairness and security.

# 4 User Stories

Each user story is realised by one or more endpoints and involves one or more microservices as follows. The numbering as well as the headings used below corresponds to the indexes of the user stories defined in the project documentation.

**Account**

2. `POST /register`
   Gateway ⟶ Authentication ⟶ *Users* ⟶ Authentication ⟶ Gateway

3. `POST /login`
   Gateway ⟶ Authentication ⟶ *Users* ⟶ Authentication ⟶ Gateway

4. (a) `GET /players/me`
   Gateway ⟶ Players ⟶ *Players* ⟶ Players ⟶ Gateway

   (b) `PATCH /players/me`
   Gateway ⟶ Players ⟶ *Players* ⟶ Players ⟶ Gateway

**Cards**

6. `GET /cards`
   Gateway ⟶ Cards ⟶ *Cards* ⟶ Cards ⟶ Gateway

7. `GET /cards/<card_id>`
   Gateway ⟶ Cards ⟶ *Cards* ⟶ Cards ⟶ Gateway

**Game**

8. (a) `POST /enqueue`
   Gateway ⟶ Matchmaking ⟶ Players ⟶ *Players* ⟶ Players ⟶ Matchmaking ⟶ *Queue* ⟶ Matchmaking ⇢ Game Engine ⇢ *Matches* ⇢ Game Engine ⇢ Matchmaking ⟶ Gateway

   (b) `POST /internal/matches/create`
   Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ Gateway

9. `POST /matches/<match_id>/deck`
   Gateway ⟶ Game Engine ⟶ Card Catalogue ⟶ *Cards* ⟶ Card Catalogue ⟶ Game Engine ⟶ Gateway

10. `POST /matches/<match_id>/moves/<round_id>`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

11. `GET /matches/<match_id>`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ Gateway

12. (a) `GET /matches/<match_id>`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ Gateway

    (b) `GET /matches/<match_id>/history`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

13. `GET /matches/<match_id>`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ Gateway

14. `GET /matches/<match_id>/round`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

15. (a) `GET /matches/<match_id>`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ Gateway

    (b) `GET /matches/<match_id>/history`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

**Others**

17. `GET /matches/history/<player_id>`*
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

18. Same as above.*

19. `GET /leaderboard`
    Gateway ⟶ Game Engine ⟶ *Matches* ⟶ Game Engine ⟶ *Rounds* ⟶ Game Engine ⟶ Gateway

# 5 Game Rules

The game is a round-based strategy game played between two players.

Before the match, each player constructs a deck of cards before the match begins, choosing from a collection of available cards. Once the decks have been submitted and validated, the game begins.

The objective is to win the majority of rounds, by playing cards strategically over multiple rounds. During each round, a random category is picked between *Total*, *Economy*, *Food*, *Environment* and *Special*, and both players simultaneously choose a card from their hand. The chosen cards are revealed, and the player with the higher score for that category wins the round.

The game lasts for a predetermined number of rounds until the end. The player with the most wins is declared the winner.

# 6 Game Flow

Assuming two players are authenticated, the game proceeds through the following API calls.

1. `POST /enqueue`: each player must enqueue to find another player and start a game.
2. `GET /status`: each player must poll the status endpoint to check if a player has been found and a match has been created.
3. `POST /matches/<match_id>/deck`: each player must submit their deck for the match.
4. (a) `GET /matches/<match_id>/round`: each player can check the current round status.
   (b) `POST /matches/<match_id>/moves/<round_id>`: each player must submit their move for the current round.
5. `GET /matches/<match_id>`: each player can check the match status.
6. `GET /matches/<match_id>/history`: each player can check the match history.
7. `GET /leaderboard`: each player can check his ranking and overall score.

Step 4 must be repeated until the match ends.

# 7 Testing

## 7.1 Unit Testing

Unit tests are implemented using **Pytest**, initializing Flask application contexts and test clients, and are located in the `tests/` directory.

Dependencies such as database sessions or external API calls are managed via isolated test databases or mocked where necessary to ensure speed and determinism.

## 7.2 Isolation Testing

Isolation tests are designed with specific Docker Compose files, and are located in the `tests/compose` directory.

We use **Newman** to execute comprehensive API test suites defined in `docs/postman/`. These tests validate the OpenAPI contract, checking status codes, headers, and JSON schemas against the running container.

In addition, all the databases intended for the isolation tests, and related to each single microservice, are mocked using **SQLite** in-memory databases to ensure fast and reliable test execution.

In order to test Game Engine Service, we mock the communication with Card Catalogue Service and avoid the communication with Players Service, so that it is possible to verify the correct behavior of the game logic without relying on the actual services. For instance, the cards are loaded directly from the JSON file.

## 7.3 System and Integration Testing

A dedicated Postman collection, located in `docs/postman/`, orchestrates complex user flows that span multiple services, such as a user registering and logging (Authentication Service), player profile creation (Player Service), joining and playing a match (Matchmaking Service and Game Engine Service).

**Continuous Integration**
The testing process is fully automated via **GitHub Actions**. The pipeline defines specific jobs for each testing stage:

1. **unit-tests**: Runs the Pytest suite on the codebase.
2. ***-isolation-tests**: Builds individual service images, spins them up, and runs Newman.
3. **integration-tests**: Builds the entire application image, and runs system-wide tests.
4. **static-analysis & build-and-scan**: Executes security audits and vulnerability scans.
5. **pip-audit**: Performs security audits on Python dependencies.

# 8 Security

## 8.1 Data

### 8.1.1 Input Sanitization

The username field provided in the request body sent by the client must be sanitized before being processed.

The username is used by the Players Service to store personal information about the user as a player. To ensure the input is safe, the following operations are performed:

1. verify that the key exists and the value is not empty;
2. validate if the length is within the allowed range;
3. ensure the string contains only allowed characters.

### 8.1.2 Encryption at Rest

To protect users' sensitive information, all data stored in the Users Database is encrypted at rest. This sensitive information primarily consists of user credentials.

The database itself performs field-level encryption, automatically encrypting data when written and decrypting it when read by the Players Service.

## 8.2 Authorization and Authentication

The system implements a stateless, distributed authentication architecture. Instead of relying on centralized session validation — which introduces latency and a single point of failure — the architecture utilizes JWTs (JSON Web Tokens) signed with asymmetric cryptography.

In this schema, the Authentication Server retains exclusive access to the private key, while the other microservices utilize the corresponding public key to independently verify the token's integrity and authenticity. This verification logic is implemented via the `@jwt_required` decorator, which intercepts incoming requests to validate the signature as the basic steps are performed as follows:

1. it grabs the public key used for validation, through Docker Compose's secrets mechanism;
2. it verifies the digital signature on the token passed through the HTTP headers;
3. it allows extracting useful authentication data, such as the user ID.

This approach ensures horizontal scalability, as validation overhead is distributed across the microservices rather than concentrated on the authentication service.

The private and public keys are used to sign and verify the JWTs, respectively. The keys are stored securely using Docker secrets to prevent unauthorized access.

The format of the Access Token is the one provided by the Flask JWT extension. An example of a decoded JWT access token is shown below. It includes some of the standard fields, such as:

- `alg`, which specifies the cryptographic algorithm used to sign the token;

- `typ`, which declares the type of token;
- `sub`, which identifies the subject of the token, typically a unique identifier for the user.


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```


Figure 2: JWT access token header.


```
{
  "fresh": false,
  "iat": 1765911626,
  "jti": "db246459-ca33-44a5-b520-6a107ce992e4",
  "type": "access",
  "sub": "0",
  "nbf": 1765911626,
  "csrf": "19452d2c-e6de-4b6e-aa83-77a18fa51a55"
}
```


Figure 3: JWT access token payload.

Once the token is expired, the client can request a new one by providing a valid refresh token to the Authentication Server. When a refresh token becomes invalid or expires, the user must re-authenticate by providing their credentials again to regain access protected the endpoints.

## 8.3 Analyses

### 8.3.1 Static Analysis

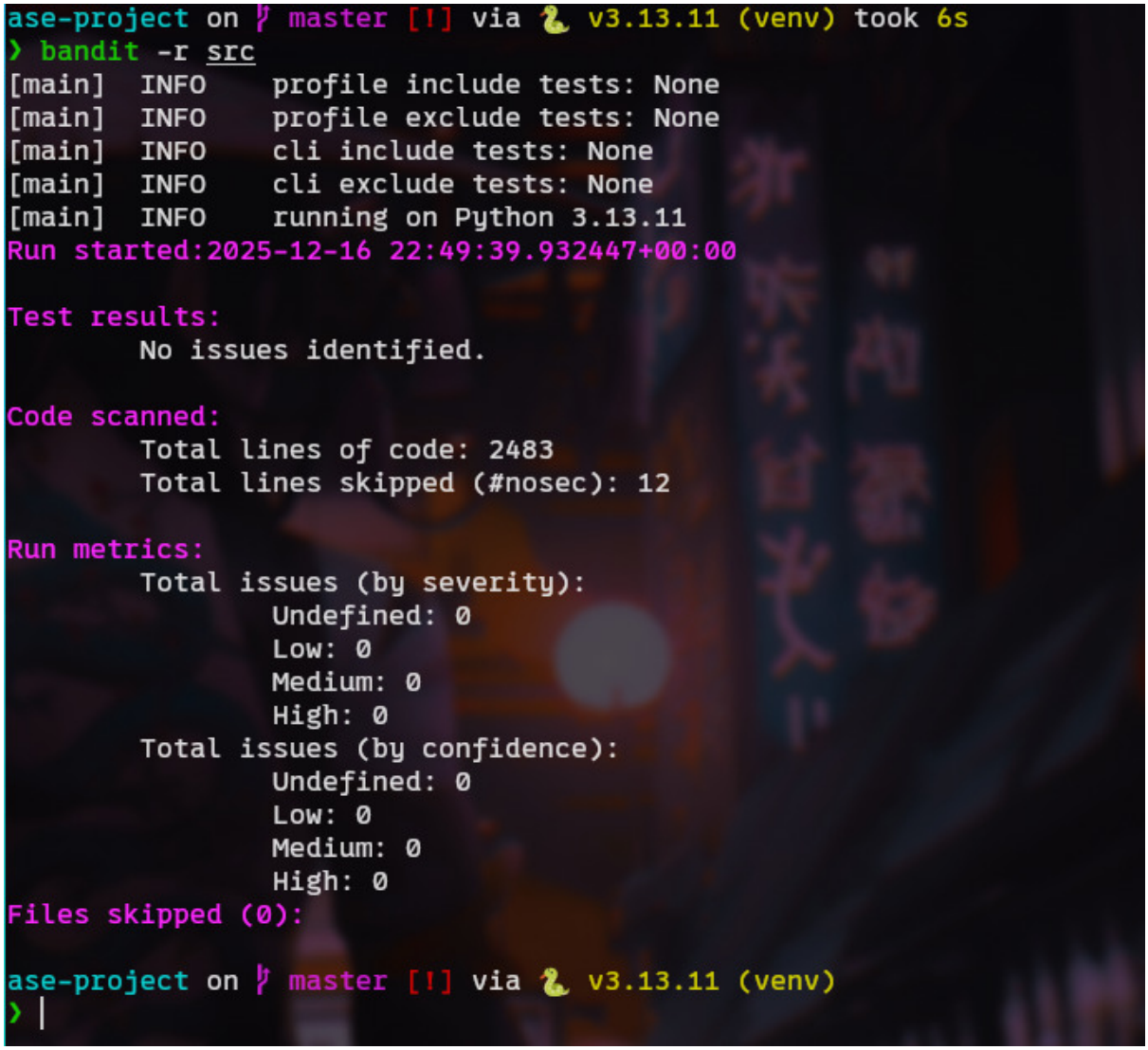
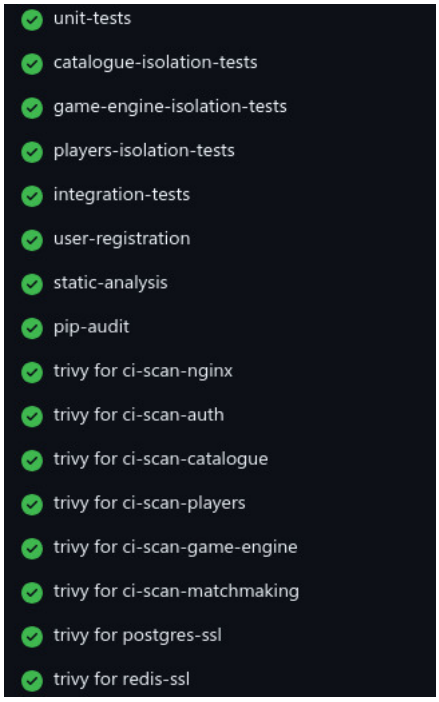



Figure 4: Bandit's final table.

### 8.3.2 Docker Scout

Instead of Docker Scout, we used Trivy. Trivy results can be seen in the GitHub Workflow Actions.

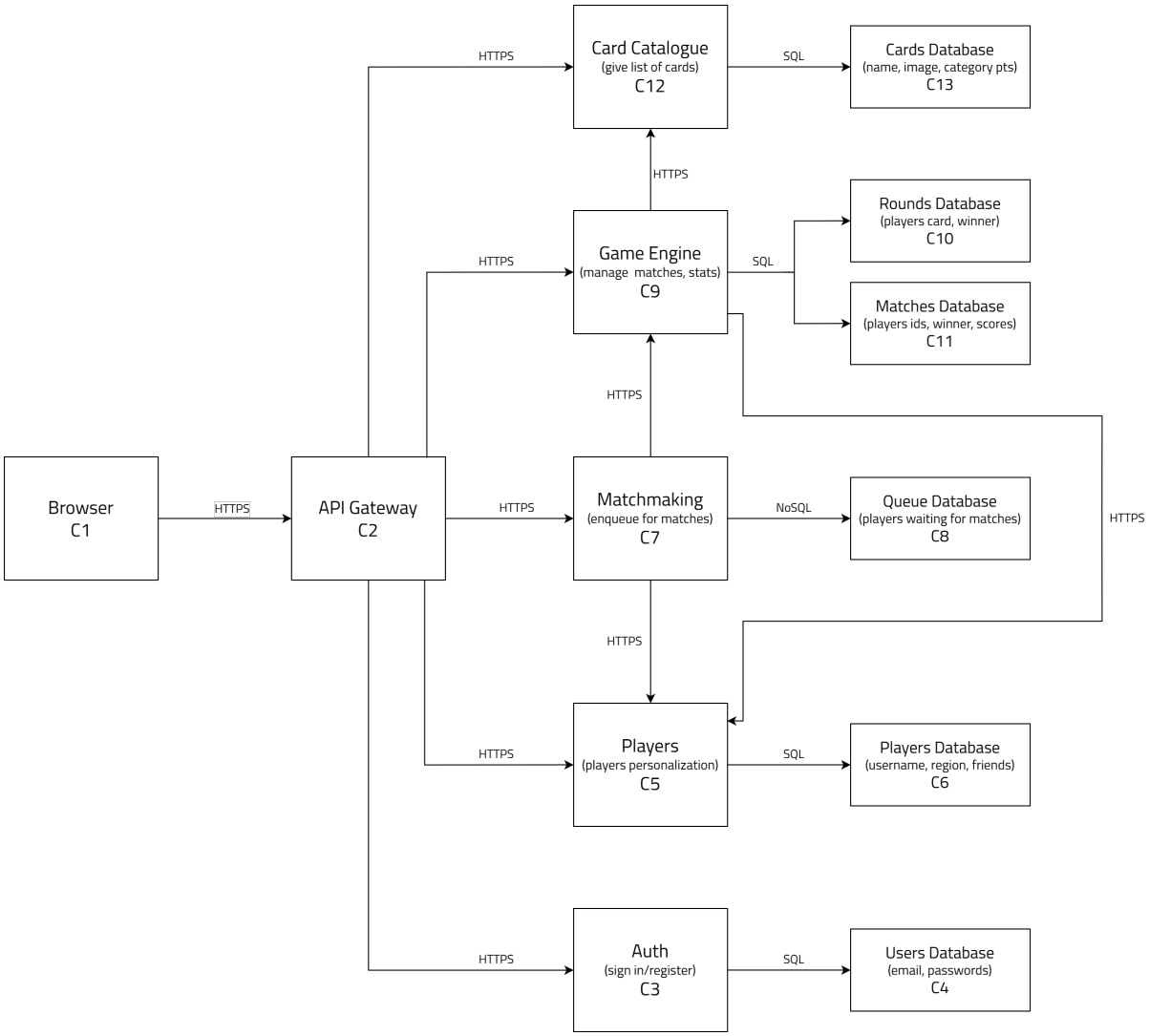
As proof of execution, the screenshots below show an example log of the Nginx service.




```
Running Trivy with options: trivy image ci-scan-nginx
2025-12-16T22:54:36Z    INFO    [vulndb] Need to update DB
2025-12-16T22:54:36Z    INFO    [vulndb] Downloading vulnerability DB...
2025-12-16T22:54:36Z    INFO    [vulndb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-db:2"
32.31 MiB / 78.52 MiB [----------------------->_____________________________] 41.15% ? p/s ?73.02 MiB / 78.52 MiB [------------------------------------------------->____] 92.99% ? p/s ?78.52 MiB /
78.52 MiB [-------------------------------------------------------------->] 100.00% ? p/s ?78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 76.97 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[-------------------------------------------->] 100.00% 76.97 MiB p/s ETA 0s78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 76.97 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[-------------------------------------------->] 100.00% 72.00 MiB p/s ETA 0s78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 72.00 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[-------------------------------------------->] 100.00% 72.00 MiB p/s ETA 0s78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 67.36 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[-------------------------------------------->] 100.00% 67.36 MiB p/s ETA 0s78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 67.36 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[-------------------------------------------->] 100.00% 63.01 MiB p/s ETA 0s78.52 MiB / 78.52 MiB [-------------------------------------------->] 100.00% 63.01 MiB p/s ETA 0s78.52 MiB / 78.52 MiB
[--------------------------------------------------] 100.00% 29.81 MiB p/s 2.8s2025-12-16T22:54:40Z      INFO    [vulndb] Artifact successfully downloaded       repo="mirror.gcr.io/aquasec/trivy-db:2"
2025-12-16T22:54:40Z    INFO    [vuln] Vulnerability scanning is enabled
2025-12-16T22:54:40Z    INFO    [secret] Secret scanning is enabled
2025-12-16T22:54:40Z    INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-12-16T22:54:40Z    INFO    [secret] Please see also https://trivy.dev/v0.65/docs/scanner/secret#recommendation for faster secret detection
2025-12-16T22:54:43Z    INFO    Detected OS     family="alpine" version="3.23.0"
2025-12-16T22:54:43Z    WARN    This OS version is not on the EOL list  family="alpine" version="3.23"
2025-12-16T22:54:43Z    INFO    [alpine] Detecting vulnerabilities...   os_version="3.23" repository="3.23" pkg_num=71
2025-12-16T22:54:43Z    INFO    Number of language-specific files       num=0
2025-12-16T22:54:43Z    WARN    Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/v0.65/docs/scanner/vulnerability#severity-selection for details.

Report Summary

┌──────────────────────────────┬────────┬─────────────────┬─────────┐
│            Target            │  Type  │ Vulnerabilities │ Secrets │
├──────────────────────────────┼────────┼─────────────────┼─────────┤
│ ci-scan-nginx (alpine 3.23.0) │ alpine │        0        │    -    │
└──────────────────────────────┴────────┴─────────────────┴─────────┘

Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)
```

# 9 Threat Model

## 9.1 Work from a Model




## 9.2 Identify Assets

- **A1: Identity & Session Data**
  Credentials (password hash), session identifiers.
- **A2: Configuration Secrets**
  Service certificates and keys, JWT signing keys.
- **A3: Game State & Logic**
  Data for playing: cards, matches, rounds, queue.
- **A4: User PII (Personally Identifiable Information)**
  Stored user data such as usernames and regions, and potential future sensitive data (e.g. birthdates) subject to GDPR regulation.
- **A5: Infrastructure & Availability**
  Docker containers.

- **A6: Logs**
  Nginx logs.
- **A7: Ranking & Reputation**
  Integrity of player scores, match history/results, leaderboards, and reputation systems (protection against tampering or cheating).

## 9.3 Identify Attack Surfaces

- **Network Attack Surface**: Consists of the API Gateway (Nginx), which serves as the entry point for all external traffic.
- **External Attack Surface**: Composed of the public REST API endpoints defined in the OpenAPI specification.
- **Internal Attack Surface**: Represented by the internal Docker network, inter-service communication, and access to persistent datastores protected by credentials injected via Docker secrets.
- **Supply Chain Attack Surface**: Includes external Python dependencies and base Docker images. Vulnerabilities in these third-party components could be exploited to compromise the application runtime.

**External Attack Surfaces (no JWT required)**

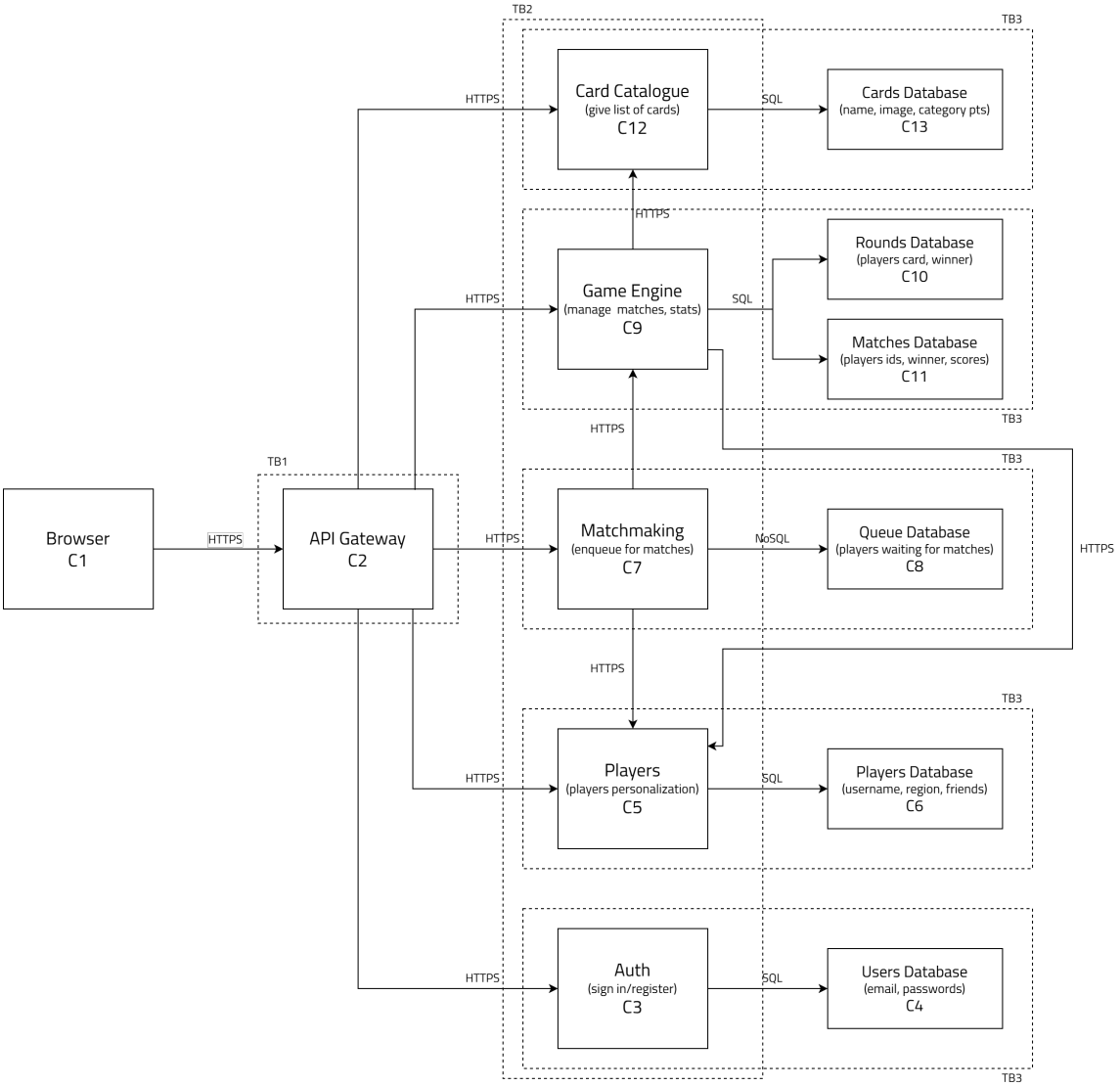
- `/login`
- `/register`

**Internal Attack Surfaces**

- `/internal/players/validation`
- `/internal/players/friendship/validation`
- `/internal/matches/create`
- `/internal/cards/validation`

## 9.4 Identify Trust Boundaries

- **TB1: External Network Boundary (Internet vs. API Gateway)**
- **TB2: Application Boundary (Microservices)**
- **TB3: Persistence Boundary (Microservices vs. Databases)**




## 9.5 Identify Threats (STRIDE)

### S - Spoofing

- **T1: JWT Token Impersonation (Spoofing Identity)**
  An attacker steals a valid access token (via XSS) gaining access to authenticated endpoints like `/players/me` as the victim (Affects **A1**; Crosses **TB1**).

### T - Tampering

- **T2: Injection Attacks (SQL/NoSQL)**
  Malicious payloads injected into input fields (e.g., username in `/players/search`) manipulate the backend database queries, potentially corrupting data (Affects **A3**, **A4**, **A7**; Crosses **TB3**).

### R - Repudiation

- **T3: Lack of Proof for Game Actions**
  If the Game Engine does not log every move with a timestamp and user signature,

a player could deny having made a losing move or claim the server glitched, and there would be no audit trail to verify the claim (Affects **A7**).

**I - Information Disclosure**

- **T4: Verbose Error Leakage**
  The application returns stack traces or database errors (e.g., "IntegrityError") in the HTTP response body when an exception occurs, revealing internal structure or logic to an attacker (Affects **A5**; Crosses **TB1**).

**D - Denial of Service**

- **T5: Matchmaking Queue Saturation**
  An attacker uses a botnet to flood the `POST /enqueue` endpoint with valid tokens, filling the Redis queue and preventing legitimate players from finding a match (Affects **A3**, **A5**; Crosses **TB2**).

**E - Elevation of Privilege**

- **T6: Bypassing Gateway to Access Internal APIs**
  If the internal network ports (e.g., 5000) are accidentally exposed or if a container is compromised, an attacker could call `/internal/players/validation` directly, bypassing the authentication checks enforced by the Gateway (Affects **A1**, **A5**; Crosses **TB2**).

## 9.6 Mitigate Threats

- **M1: Cryptographic Session Verification (Addresses T1)**
  All authenticated endpoints require a valid RS256-signed JWT. The API Gateway and microservices enforce signature verification using the public key mounted via Docker Secrets, preventing token forgery or tampering.

- **M2: Short-lived Access Tokens (Addresses T1)**
  To minimize the window of opportunity for an attacker using a stolen token, the system issues JWTs with a short expiration time.

- **M3: Strict Input & Logic Validation (Addresses T2, T3, T5)**
  Input data is strictly validated against the OpenAPI schema (types, formats, and patterns) before processing. Furthermore, the Game Engine, for example, implements logic to verify if a player is already enqued.

- **M4: Network Isolation & Access Control (Addresses T6)**
  Internal microservices and databases are isolated within a private Docker network, inaccessible from the public internet. Direct access is blocked by the API Gateway, which acts as the sole entry point.

- **M5: Generic Errors (Addresses T4)**
  The application is configured to return generic error messages to clients, suppressing stack traces to prevent information leakage.

# 10 Use of Generative AI

We utilized various AI systems, such as **Gemini Pro** (provided free of charge by Google for students) and **ChatGPT**.[1]

The main advantage of using AI is that it accelerates understanding and coding, especially when dealing with unfamiliar languages or emerging technologies, or complex new methodologies. While it is essential to review the output for errors, well-crafted prompts yield correct solutions most of the time. AI models are particularly well-trained for software engineering contexts, making them highly effective tools.

On the other hand, potential disadvantage is *over-reliance*. In fact, since AI is fast and easy to use, it may inadvertently discourage deep critical thinking. Regarding limitations, the technology is improving rapidly, so we did not encounter major blockers. However, a persistent issue is *hallucination*. Ideally, the AI should act more like a strict supervisor rather than an overly accommodating assistant, prioritizing accuracy over simply providing an answer.

# 11 Additional Features

The endpoints listed below were not part of the initial project requirements but were added as additional features. These are the ones related to the implementation of the friendship system. As before, the numbering corresponds to the indexes of the user stories defined in the project documentation.

36. `POST /players/me/friends/<username>`
37. `GET /players/me/friends`
    `GET /players/me/friends/<username>`
38. `DELETE /players/me/friends/<username>`
39. `GET /matches/<match_id>/history/<player_id>`

This feature allows players to have friends within the game, enhancing the game experience. It is implemented in the Players Service, and it is also used by the Game Engine so that players can see the match history of their friends.

In addition, the project also included the original design of all cards used in the game.

# 12 Conclusions

The project's objective was to foster teamwork, tackle larger-scale software challenges, and acquire practical, modern skills. We experienced the entire project lifecycle, from scratch to implementation, adopting a productive *DevOps* mindset.

A key example is the use of user stories: while they might seem simple or *gamified*, they are crucial for defining software requirements before coding, ultimately saving time on architectural decisions.

[1] We did not use GitHub Copilot's agent features as we did not have access to them through the university.