ASE Project Report - GROUP 2

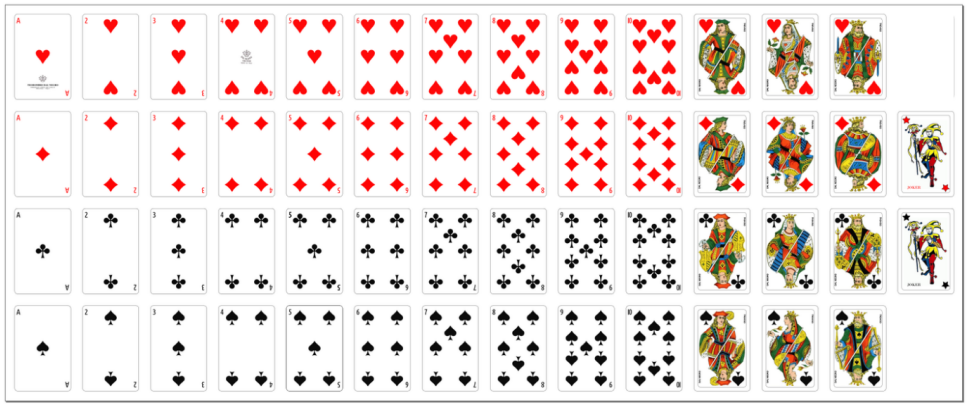
Advanced Software Engineering

First Name	Last Name	Student ID	E-Mail
Filippo	Morelli	608924	f.morelli38@studenti.unipi.it
Federico	Fornaciari	619643	f.fornaciari@studenti.unipi.it
Ashley	Spagnoli	655843	a.spagnoli9@studenti.unipi.it
Marco	Pernisco	683674	m.pernisco@studenti.unipi.it

Contents

1 Cards Overview

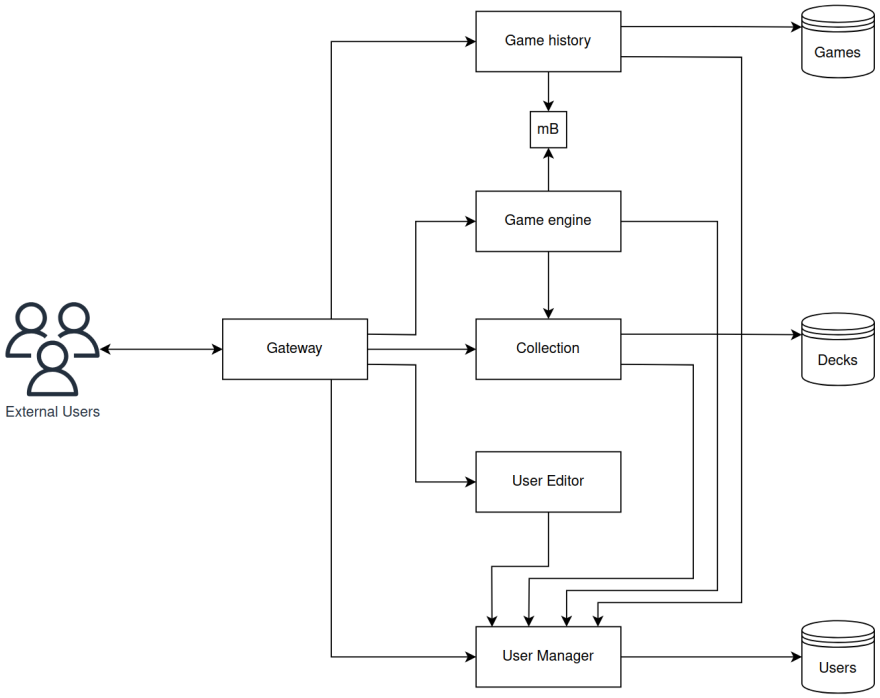
As cards we used the classical 52 + Joker deck.
As anyone could expect, the two features assigned to those cards are the number and the suit, while the Joker is a special card with no features, that beats any other card.



2 System Architecture

2.1 Architecture schema and main microservices

Below is the high-level architectural drawing of the system, illustrating the interactions between microservices.



All microservices are containerized using Docker and are written in Python. Some of them use the Flask framework and some FastAPI.
Here is a list of the main microservices of the system:

Microservice	Framework	Description
Game history	Flask	Stores and retrieves past game data
Game engine	Flask	Handles the game runtime
Collection	Flask	Handles cards and users decks
User Editor	FastAPI	Manages user profiles and settings
User Manager	FastAPI	Manages user authentication and authorization
Gateway	FastAPI	Routes requests to appropriate microservices

2.2 Design choices and microservices interactions

- **Centralized Authorization:** every microservice connects to the User Manager to verify the token header, this is for two main reasons:
  - The users can change their usernames that are stored in the User-DB, this is a way for microservices to verify they have the updated username.
  - In the future we may want to add token revocation, having a single microservice handle that would make it easier.

  This is done through the */users/validate-token* endpoint of the User Manager.

- **Shared persistance smell solutions:**
  - We connected a single microservice to each database (Games, Decks and Users).
  - Since as cards we've choosen the classical French-suited deck, that we don't expect to change, we've decided to hard-code their behavior and their images in the microservices, without relying on databases.

- **Game History -> User-Manager:** Game History needs to connect to User Manager also to get a list of (updated) usernames from a list of user-ids to return the leaderboard to users.
  This is done through the `/users/usernames-by-ids` endpoint of the User Manager.

- **User Editor:** We've separated the user profile editing and data viewing endpoints from the user manager to keep the user manager smaller and allow horizontal scalability. The User Editor then connects to dedicated endpoints of the User Manager that allow it to update user information in the DB User-DB.

- **Separated in-game logic:** The Game Engine microservice, as the name suggest, handles the whole game logic, while delegating to Game History and Collection the handling of old games and decks.
  This is done through the endpoints:
  - `/internal/update-password`
  - `/internal/update-email`
  - `/internal/update-username`
  - `/users/my-username-my-email`

- **Cards and decks:** The Collection microservice allows the users to see all the cards, their images and to handle their decks.
  The Game Engine will connect to this microservice to get the deck selected from the user: the user provides the deck's number and the Game Engine asks for the corresponding cards to the Collection. This is done through the endpoint `/collection/user-decks?user={user_id}&slot={slot_number}`.

- **Game History and Leaderboard:** The Game Engine delegates the Game History to store finished games. How this is handled is discussed in detail in the Additional Features section.
  This microservices then allow users to see their old matches, and allows everyone to see the leaderboard (think about a website that shows the best players to unauthenticated users).

3 User Stories

- Account

  2. Create an account SO THAT I can participate to the game
  - /users/register (Gateway → User Manager → Users DB → User Manager → Gateway)

  3. Login into the game SO THAT I can play a match
  - /users/login (Gateway → User Manager → Users DB → User Manager → Gateway)

  4. Check/modify my profile SO THAT I can update my information
  - /usereditor/change-username
  - /usereditor/change-password
  - /usereditor/change-email
  - /usereditor/view-data

  (Gateway → User Editor → User Manager → MongoDB → User Manager → User Editor → Gateway)
  Note: Profile checking is handled via token validation or login response.

  5. Be safe about my account data SO THAT nobody can steal/modify them
  - Implemented via JWT Authentication, Password Hashing in User Manager, and HTTPS communication.

- Cards

  6. See the overall card collection SO THAT I can think of a strategy
  - /collection/cards (Gateway → Collection → File System [cards.json] → Collection → Gateway)

  7. View the details of a card SO THAT I can think of a strategy
  - /collection/cards/card_id (Gateway → Collection → File System [cards.json] → Collection → Gateway)

- Game

  8. Start a new game SO THAT I can play
  - /game/match/join (POST with deck_slot) - Join matchmaking queue with selected deck (Gateway → Game Engine → Collection → Game Engine → Gateway)
    To check matchmaking status: /game/match/status (GET) (Gateway → Game Engine → Gateway)

  9. Select the subset of cards SO THAT I can start a match
  - For deck creation: /collection/decks (Gateway → Collection → Collection DB → Collection → Gateway)

    - /game/deck/game_id (Gateway → Game Engine → Gateway)

10. Select a card SO THAT I can play my turn
    - /game/hand/{game_id} (GET) - View current cards in hand (Gateway → Game Engine → Gateway)
    - /game/play/{game_id} (POST) - Play selected card (Gateway → Game Engine → Gateway)
11. Know the score SO THAT I know if I am winning or losing
    - /game/state/game_id (Gateway → Game Engine → Gateway)
12. Know the turns SO THAT I can know how many rounds there are till the end
    - /game/state/game_id (Gateway → Game Engine → Gateway)
13. See the score SO THAT I know who is winning
    - /game/state/game_id (Gateway → Game Engine → Gateway)
14. Know who won the turn SO THAT I know the updated score
    - /game/state/game_id (Gateway → Game Engine → Gateway)
15. Know who won a match SO THAT I know the result of a match
    - /game/state/game_id (Gateway → Game Engine → Gateway)
16. Ensure that the rules are not violated SO THAT I can play a fair match
    - Handled internally by Game Engine during /game/play/game_id execution.
    - Double checks for zero trust between Game Engine and Collection.

- Others

17. View the list of my old matches SO THAT I can see how I played
    - /history/matches (Gateway → Game History → History DB → Game History → Gateway)
18. View the details of one of my matches SO THAT I can see how I played
    - /history/matches (Gateway → Game History → History DB → Game History → Gateway)

    Note: The list endpoint returns match details.
19. View the leaderboards SO THAT I know who are the best players
    - /history/leaderboard (Gateway → Game History → History DB → Game History → Gateway)
20. Prevent people to tamper my old matches SO THAT I have them available
    - Implemented via internal service isolation (only Game Engine writes to History via RabbitMQ/Internal API) and Database access controls.

4 Rules of the Game

4.1 Deck Building Constraints

Each player constructs a **personal deck** of exactly **9 cards**. The deck must adhere to the following constraints:

- **Composition:** The deck must contain exactly **1 Joker** and **8 Suited Cards**.
- **Suit Distribution:** You must include exactly **2 cards** from each suit (♥, ♦, ♣, ♠).

- **Cost Limit:** For any suit, the total point value of the two cards **must not exceed 15**.

Card Point Costs
When calculating your deck limits, use the following costs.

Card Type	Ranks	Cost per Card	Example Pair Limit
Numbers	2 – 10	Face Value	10 + 5 = 15✓
Ace	A	7	A + 8 = 15✓
Jack	J	11	J + 4 = 15✓
Queen	Q	12	Q + 3 = 15✓
King	K	13	K + 2 = 15✓

4.2 Combat Hierarchy

When two cards are played, the winner is decided by the following hierarchy.

1. The Combat Triangle The core mechanics function like Rock-Paper-Scissors:

- **Numbers (2–10)** beat **Aces**.
- **Aces** beat **Face Cards (J, Q, K)**.
- **Face Cards (J, Q, K)** beat **Numbers**.

2. The Joker The **Joker** beats all other cards automatically. (If both players play a Joker, it is a tie).

3. Tie-Breakers If the combat rules above do not determine a clear winner (e.g., Number vs Number, or Face vs Face), compare the specific ranks:

1. **Higher Rank Wins:** (e.g., 9 beats 6, King beats Jack).
2. **Equal Rank → Suit Priority:** If ranks are identical (e.g., ♥9 vs ♦9), check suits:

   ♥ > ♦ > ♣ > ♠

3. **Mirror Match:** If players play the *exact same card* (Rank and Suit), both players win the round and gain a point.

4.3 Gameplay Flow

1. **Setup:** Both players shuffle their pre-built decks.
2. **Initial Draw:** Each player draws **3 cards** to form their starting hand.
3. **The Round:**
   - **Draw Phase:** At the end of every turn, each player draws **1 card** to get back to **3**.
   - **Battle Phase:** Both players play one card **face down**, then reveal it at the same time.
   - **Scoring:** Determine the winner based on the Combat Hierarchy. The winner earns **1 point**. If it's a tie, both players earn **1 point**.
4. **Victory:** The game ends immediately when one of the players reaches **5 points**.

4.4 Example of a Match

Alice's Deck: ♥A, ♥7, ♦A, ♦7, ♣K, ♣2, ♠K, ♠2, Joker.
Bob's Deck: ♥2, ♥3, ♦2, ♦3, ♣3, ♣4, ♠2, ♠3, Joker.

Turn	Alice	Bob	Result Reasoning	Score (A-B)
1	♥A	♣K	**Ace beats Face** (Special Rule)	1 – 0
2	♦A	♦3	**Number beats Ace** (Special Rule)	1 – 1
3	♣K	♣3	**Face beats Number** (Special Rule)	2 – 1
4	♠K	♠2	**Face beats Number** (Special Rule)	3 – 1
5	♥7	♣4	Both Numbers: 7 > 4	3 – 2
6	♦7	♠3	Both Numbers: 7 > 3	3 – 3
7	♣2	Joker	**Joker beats Everything**	3 – 4
8	Joker	♥2	**Joker beats Everything**	4 – 4
9	♥7	♠3	Both Numbers: 7 > 3	**5 – 4**

5 API Calls of the Game Flow

All the following requests require the `Authorization` header containing the JWT token of the authenticated user:

```
Authorization: Bearer <your_jwt_token>
```

5.1 Deck Building Phase

We've separated the deck building from the game (More information in Cloud Storage of Decks)

- **GET /collection/cards**: Retrieve all available cards.
- **POST /collection/decks**: Create a new deck.

```
Body:
{
  "deckSlot": 1,
  "deckName": "My Deck",
  "cards": ["hA", "h5", "d5", "dK", "c7", "c8", "sA", "s5"]
}
```

- **GET /collection/decks**: Retrieve the user's decks.

5.2 Game Phase

- **POST /game/match/join**: Join the matchmaking queue with a pre-selected deck.

```
Body:
{
  "deck_slot": 1
}
```

Note: The deck is automatically loaded when a match is found. Both players draw 3 cards to start.

- **GET /game/match/status**: Check the status of the matchmaking process (waiting/-matched).
- **GET /game/hand/{game_id}**: Retrieve the player's current hand.
- **POST /game/play/{game_id}**: Execute a play (e.g., play a card).

```
Body:
{
  "card": {
    "value": "K",
    "suit": "clubs"
  }
}
```

- **GET /game/state/{game_id}**: Retrieve the current state of the game.

5.3 Test Match

A complete match with also authentication can be executed through `/docs/tests/test_match.py`.

6 Testing

The project implements a testing strategy covering unit tests, integration tests, and performance tests to ensure system reliability, correctness, and scalability across all microservices.

6.1 Unit Testing

All tests are located in `/docs/tests` as requested.

Unit tests are executed using dedicated Dockerfile_test files like we've seen in the lectures. We made the following decisions for mocking.

- For all microservices the databases are mocked using the mongomock library, installed in the Dockerfile_test file.
- To mock the RabbitMQ for Game History we disabled the loop pinging the RabbitMQ microservice. This meant that we needed custom endpoints to add matches and users to the DB in the testing environment. `/addmatches` and `/addusernames`.
- In Game History we also needed to mock the function to get the usernames by ids of User Manager.

Unit tests execution:

- **Collection**:

```
docker build -f collection/Dockerfile_test -t collection-test .
docker run -p 5000:5000 collection-test
newman run docs/tests/collection_ut.postman_collection.json --insecure
```

- **Game History**:

```
docker build -f game_history/Dockerfile_test -t history-test .
docker run -p 5000:5000 history-test
newman run docs/tests/game_history_ut.postman_collection.json --insecure
```

- **User Manager**:

```
docker build -f user-manager/Dockerfile_test -t user-manager-test .
docker run -p 5004:5000 user-manager-test
newman run docs/tests/user_manager_ut.postman_collection.json --insecure
```

6.2 Integration Testing

The integration test suite comprises 11 test categories with 74 total requests and 93 assertions, providing comprehensive coverage of all microservices interactions.

- **IT-001: Complete Game Workflow - Happy Path**: Tests end-to-end user journey from registration to game completion.
- **IT-002: Authentication & Authorization**: Verifies login, token validation, and access control enforcement.
- **IT-003: Deck Validation**: Ensures deck building rules and constraints are correctly enforced.
- **IT-004: Game History & Leaderboard**: Checks match history retrieval and leaderboard accuracy.
- **IT-005: Cross-Service Data Consistency**: Confirms data isolation and consistency across microservices.
- **IT-006: Advanced Game Scenarios**: Tests gameplay edge cases including invalid card plays and game state verification.
- **IT-007: Error Handling & Edge Cases**: Validates error responses and handling of invalid or edge-case inputs.
- **IT-008: User Editor Integration**: Verifies user profile management, username changes, and data consistency.
- **IT-009: Complete Game Playthrough**: Tests all game engine endpoints through a complete match workflow including matchmaking, hand retrieval, card playing, and game state management.
- **IT-010: Complete Game Until Winner**: Simulates a complete 4-round game from start to finish and verifies match history persistence.
- **IT-011: Leaderboard and Statistics**: Validates leaderboard functionality, pagination, match history retrieval, and statistics tracking.

Integration tests execution:

- **Docker Compose**:

```
# Ensure all services are running
cd src
docker compose up --build

# Run integration tests (from project root)
newman run docs/tests/integration.postman_collection.json --insecure
```

Those tests are also automatically executed at each push on GitHub with GitHub Actions. The unit tests will start after the pip-audit actions doesn't find vulnerabilities.
The integration test will start after the unit tests finish successfully.

6.3 Performance Testing with Locust

Performance tests were conducted using Locust (`docs/locustfile.py`) to simulate realistic user workflows and measure system behavior under load.

Tested Workflow:

- User registration and login
- Deck creation
- Matchmaking and gameplay (card play, hand retrieval)
- History and leaderboard queries

Three user profiles (quick, normal, slow) with different wait times were used to mimic varied usage patterns.

Execution:

- 100 concurrent users, spawn rate 20 users/sec
- No errors observed during the test
- The slowest part of the system was the one handled by User Manager. This was probably because of the Argon2 hashing algorithm used for passwords, which is intentionally slow to prevent brute-force attacks.

7 Security

7.1 Security – Data

7.1.1 Input Sanitization

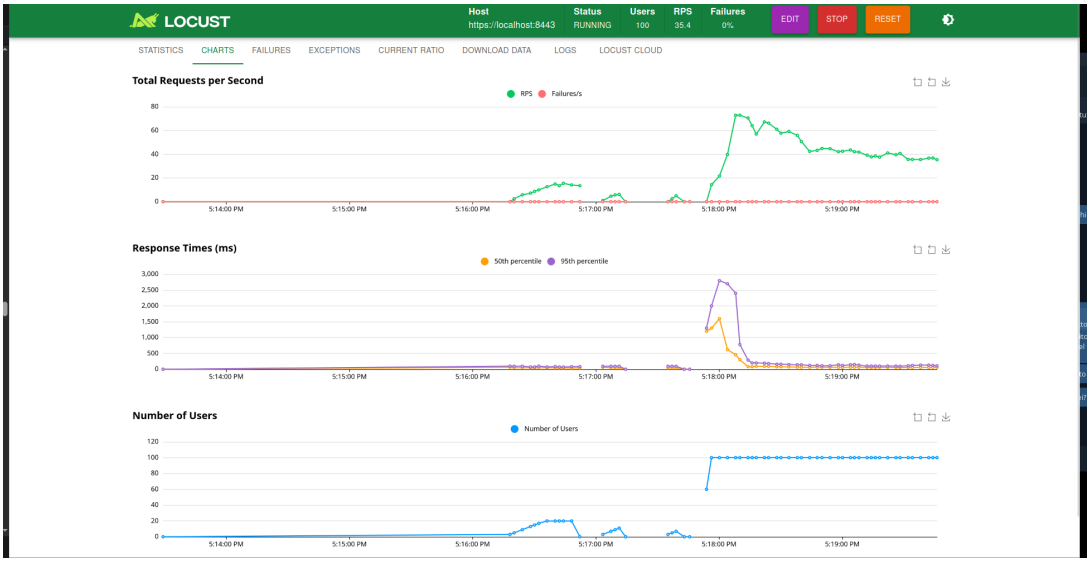
Target: User email addresses (Registration/Modification).

Threats Mitigated: NoSQL injection, XSS, Duplicate accounts.

Implementation:

- **Type Validation:** Pydantic models enforce strict typing (e.g., `email: str`).
- **Duplicate Prevention:** SHA-256 hashing of emails for secure, consistent search keys without exposing raw data.
- **Query Safety:** Explicit type casting of query parameters (e.g., `page=int`) to prevent injection.

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/collection/decks	100	0	540	1600	1700	783.6	129	1695	215	0	0
POST	/game/match/join	1183	0	61	670	1900	147.96	39	2571	104.56	8.9	0
GET	/history/leaderboard	1136	0	57	130	200	67.51	38	468	3	9.3	0
GET	/history/matches	1139	0	70	170	270	82.31	47	564	3	9.2	0
POST	/users/login	100	0	2000	2900	3000	1970.14	760	2972	292	0	0
POST	/users/register	100	0	2000	3000	3300	1931.47	368	3328	71	0	0
GET	game/hand	1727	0	67	490	660	117.56	39	1331	28.72	9.1	0
POST	game/play	693	0	86	560	660	162.22	38	1180	41.29	0	0
	Aggregated	6178	0	68	700	2500	182.82	38	3328	43.14	36.5	0



7.1.2 Data Encryption at Rest

Scope: Sensitive user data (Email, Username) in `user-db`.

Mechanism:

- **Algorithm:** Fernet (Symmetric encryption).
- **Key Management:** 32-byte cryptographically secure key, stored as Docker secret (`/run/secrets/user_d`
- **Process:**
  - **Write:** Encrypt data before insertion.
  - **Read:** Decrypt data on retrieval.
  - **Search:** Use hashed values for lookups to avoid decrypting entire collections.

7.2 Security – Authentication and Authorization

7.2.1 Architecture

Model: Centralized Authentication via `user-manager`.

Flow:

1. **Login:** Client credentials → API Gateway → `user-manager` → JWT issued.
2. **Access:** Client sends JWT → Microservice → Validate with `user-manager`.

7.2.2 Passwords hashing

The Argon2 algorithm used for password hashing already implements double hashing with salt.

7.2.3 JWT Implementation

Configuration:

- **Algorithm:** HS256.
- **Secret Storage:** Docker secret (`/run/secrets/jwt_secret_key`), accessible only to `user-manager`.
- **Payload:**
  - `sub`: Username (Subject).
  - `id`: Database ObjectId.
  - `exp`: Expiration timestamp (30 minutes).

7.2.4 Token Lifecycle

- **Validation:** `pyJWT` library automatically verifies signature and expiration.
- **Expiration Policy:** For simplicity and security, we do not implement refresh tokens. Users must re-authenticate after 30 minutes of inactivity. This reduces the attack surface by limiting token lifetime.
- **Error Handling:** Expired/Invalid tokens return HTTP 401, triggering client-side logout.

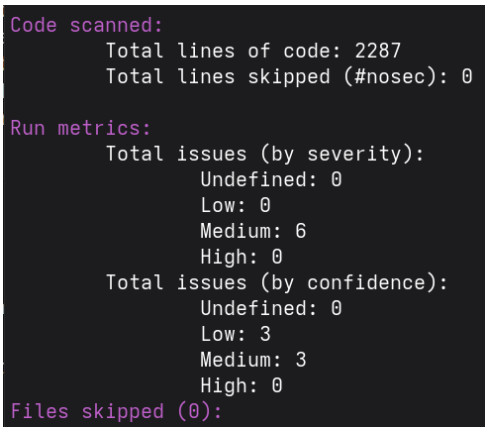
7.3 Security – Analyses

7.3.1 Static Analysis with Bandit

Bandit was used to perform static application security testing on the Python codebase.

Command executed:

```
bandit -r src/
```



7.3.2 Dependency Vulnerability Scanning

Pip Audit: Executed as a github action. It found vulnerabilities before but not after merging the Dependabot pull request.

Dependabot (GitHub): Enabled on the repository to automatically scan for vulnerable dependencies.

- Updates python-multipart from 0.0.9 to 0.0.18
- Updates requests from 2.31.0 to 2.32.4
- Updates cryptography from 42.0.7 to 44.0.1

This also solved issues raised by pip-audit

7.3.3 Container Image Vulnerability Analysis

Docker Scout: Usage: all container images where scanend for vulnerabilities using Docker Scout directly from Docker Desktop.

Mitigation Actions:

- **RabbitMQ:** 2 critical, 9 high-severity vulnerabilities because of the golang and stdlib versions in the linux image used by 3-management. Solved upgrading to `rabbitmq:4.2.1-alpine`.
- **MongoDB:** This was more of an issue,
  - The mongo:latest image we used in development had 6 high-severity vulnerabilities.
  - All the versions of the official mongo image have either a lot of vulnerabilities or run only on windows servers (windows OS images).
  - "latest" images are not an option as a final shipment version, this would mean the images possibly breaking for an update.

  - All the companies we've found providing mongo linux images moved to a paid model, providing only "latest" versions for free for development environments.
  - The only option we had were old legacy images made by those companies in the past. Fortunately we found *circleci/mongo:4.0-xenial*, a 4 years old image that relied on a really light-weight ubuntu version with 0 vulnerabilities.
  - This is definitely not a future-proof solution. The best path forward would probably be to build a custom image where we install mongoDB on top of a lightweight alpine or debian image.

- **JWT:** The User Manager microservice used an abandoned library to handle the JWT called python-jose with a high severity vulnerability. Fortunately there was a new and updated library that had the same functions, called PyJWT.
- FastAPI 0.111.0 has critical vulnerabilities caused by a starlette dependency. Solved by upgrading to 0.124.4.
- python:3.12.3-slim has critical vulnerabilities caused by the debian image used. Solved by upgrading to 3.12.12-slim.

Trivy: Command executed:

```
trivy image --severity HIGH,CRITICAL <image_name>
```

Findings:

- **Exposed Secrets:** Trivy detected hardcoded keys in repository
- **Academic Context:** As agreed with the Professor, keys are intentionally included since it's a project with educational purposes.
- **Production Mitigation:** In a real deployment, keys would be:
  - Generated dynamically per environment or, even better, we could've used certificates provided by a CA.
  - Stored in secure secret management systems
  - Never committed

  The same principles would apply for databases passwords hardcoded in the docker compose file.

7.4 Security – Threat Model

7.4.1 Assets

- **User Credentials**:
  - Passwords (hashed with Argon2).
  - Emails (encrypted with Fernet).
- **Session Tokens**: JWTs (HS256) used for stateless authentication.
- **Game Data**:
  - Match history (MongoDB `db-history`).
  - Deck collections (MongoDB `db-decks`).

- **Infrastructure Secrets**:
  - TLS Certificates and Private Keys (mounted via Docker Secrets).
  - Database Encryption Keys.
  - JWT Secret Key.

7.4.2 Attack Surface

- **API Gateway (Port 8443)**: The single external entry point. Exposes REST endpoints for authentication, game logic, and user management.
- **RabbitMQ Management Interface (Port 15672)**: Exposed only to localhost (127.0.0.1), used for monitoring message queues.
- **Internal Service Ports (Port 5000)**: Not exposed to the host network, but accessible within the `guerra-ase` Docker network.

7.4.3 Trust Boundaries

- **External Client / API Gateway**: The primary boundary. All external traffic must pass through the Gateway, which enforces TLS.
- **API Gateway / Internal Microservices**: Traffic is forwarded over HTTPS. The Gateway verifies service certificates against the local certificates.
- **Microservices / Databases**: Services connect to their respective MongoDB instances. Credentials are provided via environment variables.
- **Microservices / RabbitMQ**: Asynchronous communication is secured via TLS and username/password authentication.

7.4.4 STRIDE Analysis

- **Spoofing**:
  - *Threat*: Impersonating a user.
  - *Mitigation*: JWT authentication required for all protected endpoints. Tokens are signed with a secret key known only to `user-manager`.
  - *Threat*: Impersonating a microservice.
  - *Mitigation*: Internal communication uses TLS. Services must present a valid certificate signed with internal certificates.
- **Tampering**:
  - *Threat*: Modifying game moves or history in transit.
  - *Mitigation*: End-to-end TLS encryption (Client → Gateway → Service).
  - *Threat*: Modifying user data at rest.
  - *Mitigation*: Sensitive fields (emails) are encrypted in the database.
- **Repudiation**:
  - *Threat*: User denying a match result.
  - *Mitigation*: The `game_history` service logs all match outcomes to a persistent MongoDB database.

- **Information Disclosure**:
  - *Threat*: Leaking user emails.
  - *Mitigation*: Emails are stored as Fernet-encrypted strings.
  - *Threat*: Leaking internal architecture via error messages.
  - *Mitigation*: The API Gateway catches internal exceptions and returns generic HTTP errors to the client.
- **Denial of Service**:
  - *Threat*: Overloading the system with requests.
  - *Mitigation*: The API Gateway acts as a reverse proxy. Heavy operations (like history logging) are offloaded to RabbitMQ to prevent blocking the main application flow.
- **Elevation of Privilege**:
  - *Threat*: Accessing administrative or other users' data.
  - *Mitigation*: Route segregation in the API Gateway. The `user-manager` enforces ownership checks (users can only edit their own profile).

8 Use of Generative AI

During the project development we've made extensive use of various AI models, experimenting with them and learning how to use those tools as efficiently as possible. We've mainly taken advantage of what was given to us for free as students: GitHub Copilot Pro and Gemini Pro. Given the recent advancments in AI, this was actually the first time we found it being reliable enough to use it as a coding assistant.
We've used them in the following two areas.

- Researching informations and tools: this has gotten a lot better in the past years, with models combining their knowledge with informations found on the web, providing extensive explanations and sources, while almost never allucinating. Researching using LLMs has definitely almost replaced our reliance on classical search engines.
- Writing Code. To which we've made the following considerations.
  - The non-premium AI models of Copilot Pro (GPT-4.1, GPT-4o, GPT-5 mini, Grok Code Fast 1) were pretty bad and mostly generated bad-quality results we lost time in reviewing and then discarding. This was true even in writing something as simple as latex code, e.g. GPT-4.1 had a hard time understanding what an hyperref was.
  - We tried various AI coding VSCode extensions: Github Copilot, Roo Code, Gemini Code Assist etc. We found the Copilot extension as the most robust implementation of agentic behavior for vscode, and, together with the free premium requests, we ended up using it almost exclusively.
  - We found Claude Sonnet 4.5 as the best premium model for one-shot code generation, it has been useful to generate some initial drafts for the microservices and to re-generate whole functions.
  - We found the Copilot Gemini 3.0 Agent behaving in a more *human* way, trying to get the best result while doing as least work as possible. E.g. for big changes in a single file that meant almost a whole refactoring, it often tried to delete the whole file and start from scratch.

- The inline suggestions were a dividing topic, while some of us found them useful and kept them on, some others found them distracting and fatigue-inducing, where they suggested mostly wrong code while constantly drawing the attention of the user.
- Even if ChatGPT models are still the most popular ones, we found them working on par, if not worse, than the others, hence we made less extensive use of them. This resonates with the recent "Code Red" raised by OpenAI after the release of Gemini 3.0.
- We occasionally attempted to resolve errors through iterative AI prompting without fully diagnosing the root cause, which yielded mixed results.

8.1 Final Remarks

We found experienced and smart human beings better in any way but speed compared to the most powerful AI models that are publicly available. At the moment LLMs can be a useful tool for those people to work faster, but definitely not as a replacement of them.
Reasoning models behaved way better, showing the robustness of more rigid and iterative workflows, applied to the more efficient and smaller MoE (Mixture of Experts) models, compared to one-shot larger models. This resonates with what is shown in the course for cloud services, where decomposing complex tasks (monoliths) into smaller, more easily verifiable tasks (microservices), often yields better results.

9 Additional Features

9.1 Green User Stories

The project implements two green user stories that enhance the player's gameplay experience by providing essential game state information.

9.1.1 Viewing Cards in Hand

22. who's turn it is SO THAT I know if i can play or not

- `GET /state/{game_id}` (Gateway → Game Engine → Gateway)
- Response: current round state (empty, one card played, both cards played)

24. To be able to see the cards in my hand SO THAT I know that card I can play next

- `GET /hand/{game_id}` (Gateway → Game Engine → Gateway)
- Returns: JSON array of card objects (`value`, `suit`)

9.2 Cloud Storage of Decks

To allow players to keep their decks stored in the cloud, we modified the way they handle decks. Users must first create the decks connecting to the endpoint `collection/decks`. They have 5 slots (5 possible decks).
Then that deck will be stored on the decks database.
Users will then be able to choose one of their decks when starting a game with the endpoint `game/match/join`.

9.3 Client

The project includes a Python-based Command Line Interface (CLI) client located in the `/client` directory. This client serves as the frontend for the game, communicating with the backend services via the API Gateway. It uses the `rich` and `questionary` libraries to provide an interactive terminal experience.

To run the client using Docker, execute the following command (Linux):

```
cd client/
docker build -t ase-client . --no-cache
docker run -it --rm \
  --add-host=host.docker.internal:host-gateway \
  -e API_GATEWAY_URL="https://host.docker.internal:8443/" \
  -e GATEWAY_CERT_PATH="./gateway_cert.pem" \
  ase-client
```

Ensure that the backend services are running before starting the client.

9.3.1 User Authentication

- **Login/Register:** Users are prompted to authenticate via the API Gateway and Authentication microservice upon startup.
- **Token Management:** The client locally stores the username and JWT token. The token is included in the headers of every request (defined in "**apicalls.py**").
- **Prerequisite:** Users must navigate to "**Decks**" to create at least one deck before playing.

9.3.2 Card Collection and Deck Management

- **Capabilities:** Users can view the collection, create new decks (Deck Page), view existing decks, and delete decks (View Deck).
- **Visuals:** Selecting a card in the local version opens its image in a new window (not in Containerized version).
- **Deck Creation:** Simplified logic grants the maximum value for every suit during creation.

9.3.3 Game Engine Interaction

- **Testing Setup:** Requires two terminal instances logged in with different user accounts.
- **Matchmaking:** Select "**Play a Match**" to enter the queue and await an opponent.
- **Gameplay:** The interface displays real-time game state (hand, scores, opponent name) and enables interactive card play per turn.
- **Conclusion:** Displays final results and automatically updates the user's game history.

9.3.4 Game History

- **Access:** Located within the "**Leaderboard**" section of the main menu.
- **Options:** Users can toggle between the full Global Leaderboard and their Personal Match History.
- **Navigation:** The leaderboard includes a pagination system to browse all players.

9.4 Endpoint-based Service Interaction Smell - Proof of Concept

The project requirements asked to make up for the **Wobbly service interaction** smell. We solved it mainly with timeouts as suggested.
Regarding the **Endpoint-based service interaction** smell, the requirements said to ignore it since it would've been difficult to solve.
We decided to partially solve it anyway in a single interaction between the **Game History** and **Game Engine** microservices as a proof-of-concept.
In this interaction we replaced the timeout with RabbitMQ, a Message Broker, that solved both the **Wobbly service interaction** and the **Endpoint-based service interaction** smells. This meant a new docker container for RrabbitMQ and running code to fill up the RabbitMQ queue in Game Engine and to ping it for new data in Game History.

9.5 Test-match

As already said in Test Match we made a simple script to test the system. You can find it in `/docs/tests/test_match.py`.

9.6 pip-audit Actions

Together with the actions for unit tests and integration tests, we implemented a GitHub Action that runs `pip-audit` to automatically check for known vulnerabilities in our Python dependencies.