

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

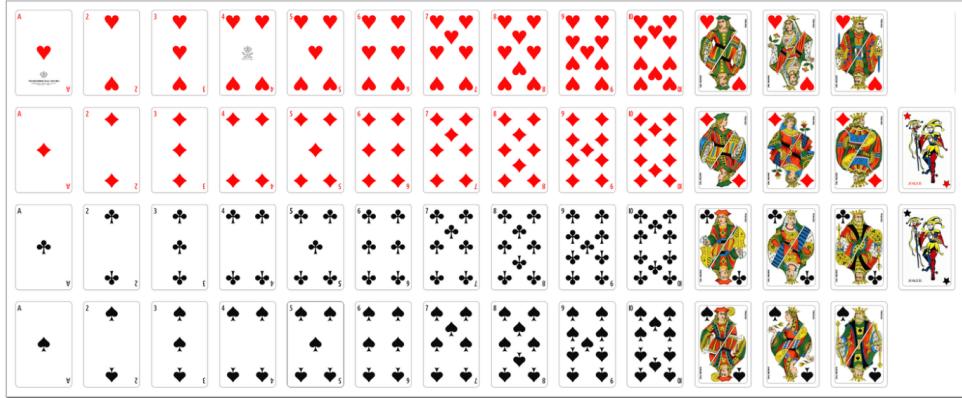
1	Cards Overview	3
2	System Architecture	3
2.1	Design choices and microservices interactions	4
3	User Stories	4
4	Rules of the Game	5
4.1	Deck Building Constraints	5
4.2	Combat Hierarchy	5
4.3	Gameplay Flow	6
4.4	Example of a Match	6
5	API Calls of the Game Flow	6
5.1	Deck Building Phase	6
5.2		7
6	Testing	7
7	Security	7
7.1	Data Security	7
7.2	Authorization and Authentication	7
7.3	Security Analyses	7
8	Threat Model	7
9	Use of Generative AI	7
9.1	Final Remarks	8

10 Additional Features	8
10.1 Green User Stories	8
10.2 Cloud Storage of Decks	8
10.3 Client	8
10.4 Endpoint-based Service Interaction Smell - Proof of Concept	8
11 Build and Run Instructions	8
11.1 Prerequisites	8
11.2 Quick Start Guide	8
11.3 Development & Testing	9
11.3.1 Environment Configuration	9
11.3.2 Testing the Workflow	9
11.3.3 Key API Endpoints	9
11.4 Maintenance	10

1 Cards Overview

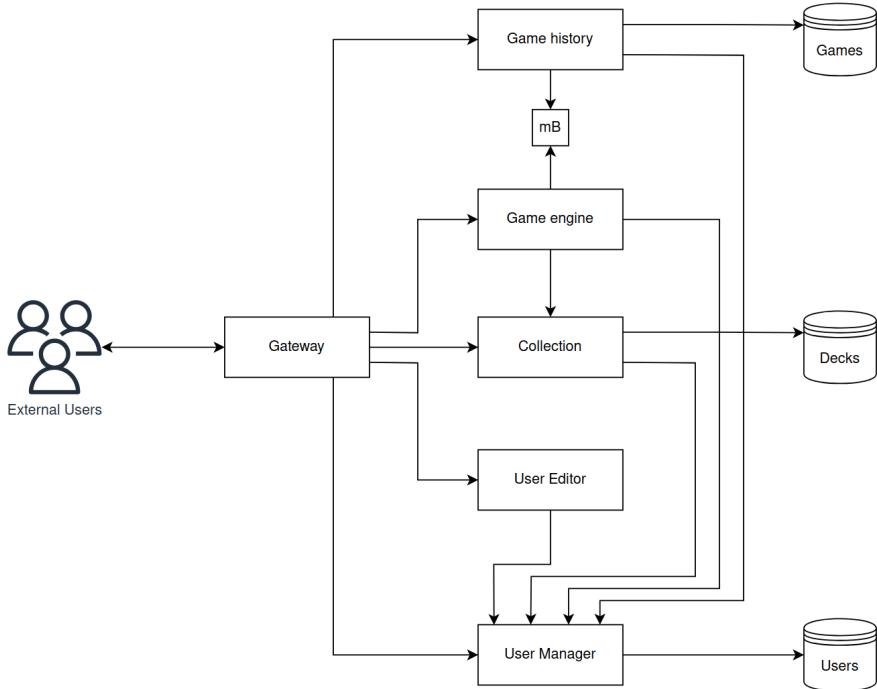
As cards we used the classical 52 + Joker deck.

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

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.

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

2.1 Design choices and microservices interactions

- **Centralized Authorization:** (draft) Every microservice connects to the User Manager to verify the token header (so that I can get dynamic username). User manager logs out users when they change username, this means that if we used decentralized auth, a logged-in user could see its old username using an old token, also, we don't have a way to force the log-out of a user with decentralized
- **User Information Update:** We've separated the user profile editing endpoints from the user manager to keep the user manager smaller and allow horizontal scalability. The User Editor then connects to a single endpoint of the User Manager that allows to update user information in the DB.
-
- **Separated in-game logic:** The Game Engine microservice handles the whole game logic, it
- **Hard-coded cards:** Since as cards we've chosen 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.

3 User Stories

- Account
 1. Create an account SO THAT I can participate to the game
 2. Login into the game SO THAT I can play a match
 3. Check/modify my profile SO THAT I can update my information
 4. Be safe about my account data SO THAT nobody can steal/modify them
- Cards
 5. See the overall card collection SO THAT I can think of a strategy
 6. View the details of a card SO THAT I can think of a strategy
- Game
 7. Start a new game SO THAT I can play
 8. Select the subset of cards SO THAT I can start a match
 9. Select a card SO THAT I can play my turn
 10. Know the score SO THAT I know if I am winning or losing
 11. Know the turns SO THAT I can know how many rounds there are till the end

- 12.** See the score SO THAT I know who is winning
- 13.** Know who won the turn SO THAT I know the updated score
- 14.** Know who won a match SO THAT I know the result of a match
- 15.** Ensure that the rules are not violated SO THAT I can play a fair match
- Others
- 16.** View the list of my old matches SO THAT I can see how I played
- 17.** View the details of one of my matches SO THAT I can see how I played
- 18.** View the leaderboards SO THAT I know who are the best players
- 19.** Prevent people to tamper my old matches SO THAT I have them available

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	$10 + 5 = 15 \checkmark$
Ace	A	7	$A + 8 = 15 \checkmark$
Jack	J	11	$J + 4 = 15 \checkmark$
Queen	Q	12	$Q + 3 = 15 \checkmark$
King	K	13	$K + 2 = 15 \checkmark$

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:

$$\heartsuit > \diamondsuit > \clubsuit > \spadesuit$$

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

5.1 Deck Building Phase

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

5.2

6 Testing

7 Security

7.1 Data Security

7.2 Authorization and Authentication

7.3 Security Analyses

8 Threat Model

9 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. 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 hallucinating. 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 slop 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 regenerate 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.

9.1 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.

10 Additional Features

10.1 Green User Stories

10.2 Cloud Storage of Decks

To allow players to keep their decks stored in the cloud, we added

10.3 Client

10.4 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.

11 Build and Run Instructions

This project is a multi-service card game platform designed for two players. It features authentication, deck building, a match-simulation engine, and history tracking. All services are containerized and orchestrated via Docker Compose.

11.1 Prerequisites

Before starting, ensure your environment meets the following requirements:

- **Docker & Docker Compose** (Required for orchestration).
- **Python 3.10+** (Optional, only required for local non-containerized development).

11.2 Quick Start Guide

1. Clone the repository

```
git clone https://github.com/ashleyspagnoli/ASE_project.git  
cd ASE_project/src
```

2. **Build and launch services** Run the following command to build the images and start the containers:

```
docker compose up --build
```

3. **Verify Service Status** Once the containers are running, the architecture exposes the following endpoints:

Service Name	Responsibility	Local URL
User Manager	Authentication & JWT	https://localhost:5004
Collection	Deck Management	http://localhost:5003
Game Engine	Core Logic & Matchmaking	http://localhost:5001
Game History	Match Logging	http://localhost:5002

11.3 Development & Testing

11.3.1 Environment Configuration

Each microservice is configured via environment variables. For specific configuration keys, refer to the `Dockerfile` and `requirements.txt` located in each service's directory.

11.3.2 Testing the Workflow

A Postman collection is provided for end-to-end testing. Import `game_workflow.postman_collection.json` into Postman to simulate a full lifecycle:

- User Registration and Login (Token generation).
- Deck creation and validation.
- Matchmaking and gameplay simulation.

11.3.3 Key API Endpoints

Authentication

- `/users/register`: user registration [POST]
- `/users/login`: user login [POST]
- `/users/validate-token`: internal JWT token validation [GET]

Deck Management

- `/collection/cards`: get the collection of cards [GET]
- `/collection/decks`: create a new deck [POST]

Game Engine

- `/game/connect`: connect to start playing the game [POST]
- `/game/matchmake`: manual request to start the match [POST] (TO REMOVE)

- `/game/play/{game_id}`: play a card [POST]
- `/game/state/{game_id}`: get the state of the game [GET]

Game History

- `/leaderboard`: get the whole leaderboard [GET]
- `/matches`: get the history of your played matches [GET]
- `/addmatch`: memorize a new match [POST]

11.4 Maintenance

To stop the application and remove containers/networks, run:

```
docker compose down
```