ASE Project Report

Group SRB

Advanced Software Engineering - Academic Year 2024-2025

Group Members

Tosic Natalija

Stevanovic Jana

Panic Veljko

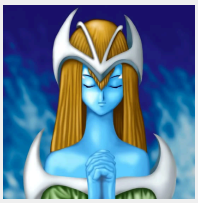
Salmi Abderrahmane

Ceccotti Lorenzo

Table of Contents

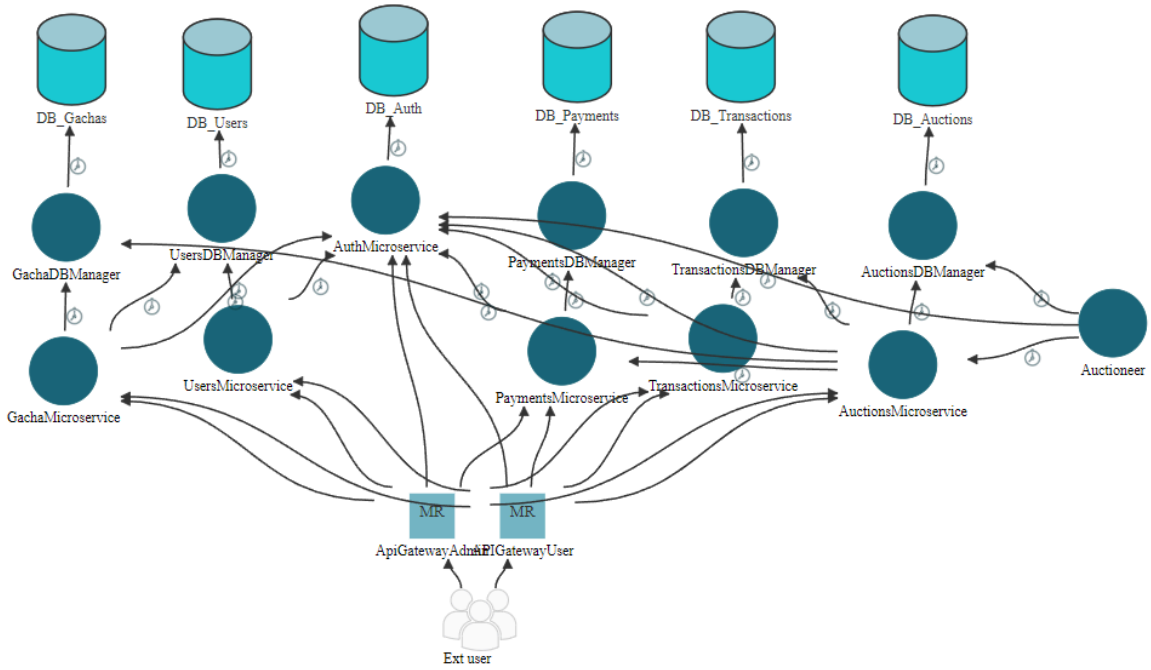
1. Gachas Overview

The gacha system in our project revolves around a collection of rare and unique items inspired by Yu-Gi-Oh! cards. Each gacha item has an associated rarity percentage, which determines the likelihood of it being rolled.
Here is a list of some of the Gacha items that populate the system:

Image	Name	Description	Rarity Percentage
	Kuriboh	An adorable monster that defends you in battles.	24.75%
	Giant Soldier of Stone	A giant made of stone that stands as a solid defense.	12.50%
	Mystical Elf	A sorceress with high defensive power.	5.00%
	Spirit of the Harp	A spirit that brings harmony and high defense.	1.25%
	The Shadow Who Controls the Dark	A sinister shadow with hidden power.	0.025%

2. Architecture

2.1. Architecture Diagram



2.2. Microservices Overview

Microservice	Responsibility	Language & Technology
ApiGatewayAdmin	Handles all admin-related requests and routes them to the appropriate microservices.	Python (Flask)
ApiGatewayPlayer	Handles all player-related requests and routes them to the appropriate microservices.	Python (Flask)
AuthMicroservice	Handles authentication operations like registration, login, and logout, generating and validating access tokens for both admins and players.	Python (Flask, PyJWT, SQLAlchemy)

AuctionDBManager	Manages the database for auctions, including creating, updating, and retrieving auction data.	Python (SQLAlchemy, Flask)
AuctionsMicroservice	Handles auction operations for both admins and players such as bidding and auction creation.	Python (Flask)
Auctioneer	Checks if there are any open auctions that need to be closed. If so it sends a request to the AuctionMicroservice to handle them.	Python
GachaDBManager	Manages the database for gacha items, including item creation, updates, and retrieval.	Python (SQLAlchemy, Flask)
GachaMicroservice	Provides player and admin functionalities for gacha operations, such as rolling and managing items.	Python (Flask)
PaymentsDBManager	Manages the database for in-game currency transactions, including purchases and refunds.	Python (SQLAlchemy, Flask)
PaymentsMicroservice	Handles in-game currency transactions, including player purchases and refunds.	Python (Flask)
TransactionsDBManager	Manages the database for player transaction history and records of in-game activities.	Python (SQLAlchemy, Flask)
TransactionsMicroservice	Handles player transaction records and ensures data consistency for all in-game activities.	Python (Flask)
UsersDBManager	Manages the database for player and admin profiles, including authentication and profile updates.	Python (SQLAlchemy, Flask)

UsersMicroservice	Provides functionalities for player and admin account management, such as getting information about a player, deleting the player, banning the player etc.	Python (Flask)

2.3. Connections Between Microservices

Microservices	Reason for Connection
GachaMicroservice & AuthMicroservice	The **GachaMicroservice** communicates with the **AuthMicroservice** to validate the access token sent in the request, ensuring that the user is authorized to perform the operation. It also retrieves the user's profile information to check if the user has sufficient in-game currency for the roll.
GachaMicroservice & GachaDBManager	The **GachaMicroservice** interacts with the **GachaDBManager** to fetch the list of available gacha items and select one based on the roll probability and rarity settings. It also to uses it to add the newly rolled gacha item to the player's collection
GachaMicroservice & UsersDBManager	After the roll, the **GachaMicroservice** communicates with the UsersDBManager to update the player's in-game currency balance, deducting the cost of the roll.
AuthMicroservice & UserDBManager	**AuthMicroservice** interacts with the **UserDBManager** to validate the access token sent in the request, ensuring that the user is authorized to perform the operation. It also retrieves the user's profile information that is needed by other microservices. It also creates the user during registration and gets all user info in the userInfo endpoint.
AuctionMicroservice & PaymentsMicroservice	The **AuctionMicroservice** interacts with the **PaymentsMicroservice** to manage in-game currency transactions. For example, it deducts currency when a player bids and credits currency when a player's auction is won.
AuctionMicroservice & GachaDBManager	The **AuctionMicroservice** retrieves gacha item details from the **GachaDBManager** to display item information in the auction listings and validate item availability.

3. Player User Stories

User Story	Endpoint	Microservices
I WANT TO create my game account/profile SO THAT I can participate in the game	POST /api/player/register	Player API Gateway, AuthMicroservice, UserDBManager
I WANT TO delete my game account/profile SO THAT I can stop participating in the game	DELETE /api/player/delete/{userId}	Player API Gateway, UsersMicroservice, UserDBManager, AuthMicroservice
I WANT TO modify my account/profile SO THAT I can personalize my account/profile	PUT /api/player/update/{userId}	Player API Gateway, UsersMicroservice, UserDBManager
I WANT TO login and logout from the system SO THAT I can access and leave the game	POST /api/player/login POST /api/player/logout	Player API Gateway, AuthMicroservice
I WANT TO be safe about my account/profile data SO THAT nobody can enter in my account and steal/modify my info	All player endpoints with token validation	AuthMicroservice
I WANT TO see the system gacha collection SO THAT I know what I miss of my collection	GET /api/player/gacha/system-collection	Player API Gateway, Auth Microservice, Gacha Microservice, Gacha DB Manager
I WANT TO see the info of a system gacha SO THAT I can see the info of a gacha I miss	GET /api/player/gacha/system-gacha/{gachaId}	Player API Gateway, Auth Microservice, Gacha Microservice, Gacha DB Manager
I WANT TO see my gacha collection SO THAT I know	GET /api/player/gacha/player-co	Player API Gateway, Auth Microservice, Gacha

how many gacha I need to complete the collection	llection/{userId}	Microservice, Gacha DB Manager
I WANT TO see the info of a gacha of my collection SO THAT I can see all of the info of one of my gacha	GET /api/player/gacha/player-collection/{userId}/gacha/{gachaId}	Player API Gateway, Auth Microservice, Gacha Microservice, Gacha DB Manager
I WANT TO use in-game currency to roll a gacha SO THAT I can increase my collection	POST /api/player/gacha/roll/	Player API Gateway, Auth Microservice, Gacha Microservice, Gacha DB Manager, Users DB Manager
I WANT TO buy in-game currency SO THAT I can have more chances to win auctions	POST /api/player/currency	Player API Gateway, PaymentsMicroservice, TransactionDBManager, UserDBManager
I WANT TO be safe about the in-game currency transactions SO THAT my in-game currency is not wasted or stolen	All currency-related endpoints with token validation	
I WANT TO see the auction market SO THAT I can evaluate if buy/sell a gacha	GET /api/player/auction/market	AuctionsMicroservice AuthMicroscervice AuctionDBManager
I WANT TO set an auction for one of my gacha SO THAT I can increase in-game currency	POST /api/player/auction/create	AuctionsMicroservice AuthMicroscervice AuctionDBManager GachaDBManager
I WANT TO bid for a gacha from the market SO THAT I can increase my collection	POST /api/player/auction/bid/{auction_id}	AuctionsMicroservice AuthMicroscervice AuctionDBManager PaymentsMicroservice
I WANT TO view my transaction history SO THAT I can track my market movement	GET /api/player/market-transaction	TransactionsMicroservice AuthMicroscervice TransactionsDBManager
I WANT TO receive a gacha when I win an auction SO	Triggered by auction win logic in backend	AuctionsMicroservice AuthMicroscervice

THAT only I have the gacha I bid for		AuctionDBManager GachaDBManager
I WANT TO receive in-game currency when someone wins my auction SO THAT the gacha sell works as I expect	Triggered by auction end logic in backend	AuctionsMicroservice AuthMicroscervice AuctionDBManager PaymentsMicroservice
I WANT TO receive my in-game currency back when I lost an auction SO THAT my in-game currency is decreased only when I buy something	Triggered by auction loss logic in backend	AuctionsMicroservice AuthMicroscervice AuctionDBManager PaymentsMicroservice
I WANT TO that the auctions cannot be tampered SO THAT my in-game currency and collection are safe	Input validation and secured transactions and token validation	

4. Market Rules

In a nutshell, the auction market allows players to sell their gacha items to others. Auctions are time-limited, and players bid using in-game currency. The highest bidder at the end of the auction wins the item, and the seller receives the final bid amount.
In designing the auction market system, we established several rules to ensure a good user-friendly experience for the players. Below are the core rules of the market:

Scenario	Rule
Winning an auction:	The highest bidder wins the auction at the end of the timer. The gacha item is added to the winner's collection, and the final bid amount is transferred to the seller.
When an auction ends without any bids:	If no bids are placed before the auction expires, the gacha item is automatically returned to the seller's collection without

	any currency exchange.
When someone else bids higher:	If a player places a bid and is outbid by another player, the system refunds the entire bid amount to the previous bidder immediately.
Bidding on your own auction:	Players cannot place bids on auctions they created. The system validates the ownership of the auction item before accepting bids.
Bidding while already the highest bidder:	Players cannot bid again on an auction where they are already the highest bidder. This prevents unnecessary currency deductions and redundant actions.
Placing a bid without sufficient currency:	The system validates the player's in-game currency balance before accepting a bid. If the balance is insufficient, the bid is rejected, and an error message is displayed.
Fraud prevention and tamper-proofing:	All transactions are logged in the TransactionsDBManager, and bids are verified through the PaymentsMicroservice to ensure integrity and avoid tampering.

5. Testing

5.1 Unit Testing

We implemented unit tests for all microservices to ensure that each endpoint works as expected in isolation. Key facts about the unit testing process include:

- **Mock Data**: Each microservice uses mock data to simulate the functionality of its database and dependencies. For instance, the `GachaMicroservice` operates with a local mocked list of gacha items during unit tests.

- **Postman Collections**: All unit tests were executed using Postman collections specifically designed for each microservice.
- **GitHub Actions Workflow**: Unit tests are automatically executed through a GitHub Actions workflow, triggered on every push to the `main` branch.

5.2. Integration Testing

Integration testing verifies the behavior of microservices when they communicate via the gateways. Key facts include:

- **Gateway Verification**: Both the `ApiGatewayPlayer` and `ApiGatewayAdmin` were tested to confirm they correctly route requests to the appropriate microservices.
- **End-to-End Workflow**: Integration tests simulate complete player and admin workflows, including gacha rolls, auction bids, and currency purchases, ensuring data consistency across microservices.
- **Database Connectivity**: Real databases were used during integration tests to validate query consistency and responses.

5.3. Performance Testing

Performance testing focused on identifying bottlenecks and ensuring the system handles high traffic efficiently. Key facts include:

- **Locust**: We used Locust to simulate high traffic on the gateways and critical microservices. For example, the `GachaMicroservice` was scaled to test its ability to handle a high volume of roll requests.
- **Scaling Microservices**: Bottleneck microservices, such as `AuthMicroservice` and `GachaMicroservice`, were scaled during Locust testing to evaluate performance improvements.

5.4. Particular Challenges and Solutions

- **Mocking External Services**: In some cases, mocking services like the `AuthMicroservice` was necessary for unit tests due to its dependency on token validation.

6. Security – Data

6.1. Input Sanitization

Selected Input:

- **Operation:** Gacha item creation
- **Used Endpoint:** POST /api/admin/gacha
- **Selected Input:** The JSON input for creating a gacha item.
- What It Represents: This input represents the information required to create a new gacha item in the system, like gacha name, description, image url, and rarity percent.

Microservice(s) Involved:

- **Admin API Gateway:** Sanitizes incoming request's input.
- **Gacha Microservice:** Validates the request by making sure all required fields are present in the input with their correct type (ex: description must be a string).
- **Gacha DB Manager:** Processes the sanitized and validated input to create a new gacha item.

Sanitization Approach:

The `sanitize_data` function was used to sanitize the input JSON payload to protect the system from injection attacks (like SQL injections) and other malicious inputs. This

function was applied in the API Gateway before forwarding the request to the other microservices.

Key Features:

- **SQL Injection Prevention:**
  - Removes patterns such as `;`, `--`, `||`, and `&&` to mitigate SQL injection attempts.
  - Strips SQL keywords like `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc.
- **XSS (Cross-Site Scripting) Protection:**
  - Uses the `bleach` library to clean HTML or JavaScript injection attempts.
- **Recursive Sanitization:**
  - Traverses nested JSON structures to sanitize all string fields.

Example:

```
// ORIGINAL INPUT
{
    "description": "<script>alert('XSS')</script>",
    "image": "https://example.com/image.jpg'; DROP TABLE gachas; --",
    "name": "New Test Item <img src='x' onerror='alert(1)'>",
    "rarityPercent": 50.9
}

// SANITIZED INPUT
{
    "description": "&lt;script&gt;alert('XSS')&lt;/script&gt;",
    "image": "https://example.com/image.jpg",
    "name": "New Test Item &lt;img src='x' onerror='alert(1)'&gt;",
    "rarityPercent": 50.9
}
```

6.2. Data at Rest Encryption

The data that is encrypted at rest is the player and admin password. It is stored in the account and admin databases, respectively. During registration, the password coming from the client is concatenated with a random salt and that is hashed. The hashed password is then stored in the database along with the salt. During login the password passed from the client is in the same way hashed using the same salt, hashing algorithm and secret. For login the two password hashes are compared.

7. Security – Authentication and Authorization

The scenario selected is the centralized scenario. The steps of token validation are as follows:

- From the api gateway, both for players and admins, the authentication headers are passed onto requests made to specific microservices.
- The first thing that is done in an endpoint in a microservice that requires a user to be authorized is sending a token verification request to the AuthMicroservice. For verifying a player token endpoint GET /helloPlayer is used and for verifying an admin token the endpoint GET /helloAdmin is used. If the microservice requires additional player information it calls the GET /api/player/userInfo endpoint from the AuthMicroservice which returns the user information of the user whose access token is passed.

The keys used to sign the token are kept in an .env file that is passed to the AuthMicroservice using docker secrets and then is accessed in the microservice at path `/run/secrets/token_secret`

The payload of the access token:

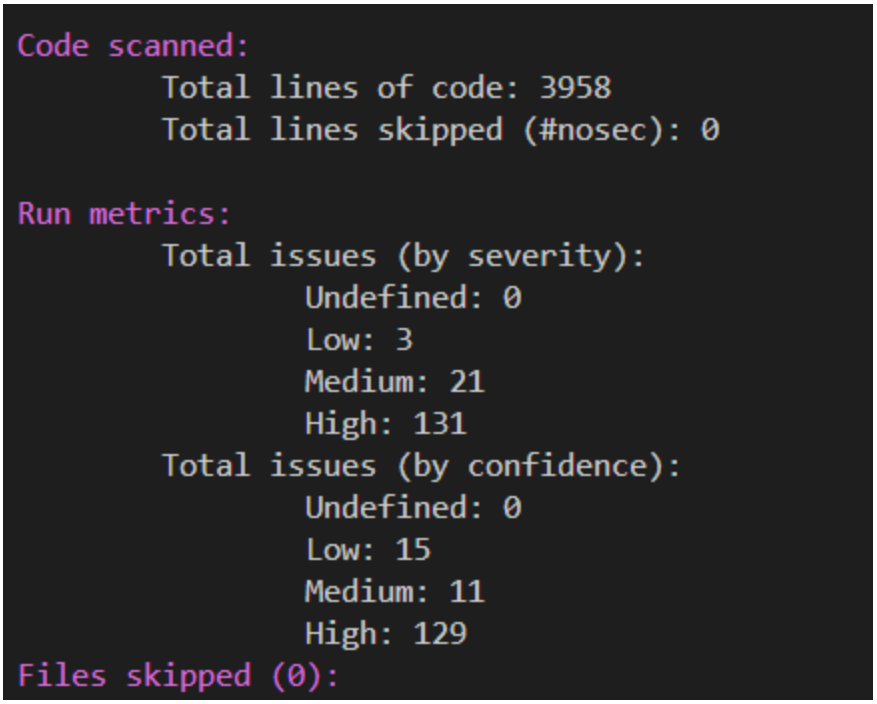
- `iss`: the issuer of the token, "Ase Project"
- `exp_time`: time until when the token is valid
- `username`: the username of the user
- `roles`: the roles the user has, "player" fro players and "admin" for admins
- `userId`: the id of the user

8. Security – Analyses

TODO (1-3 Pages)

8.1. Static Analysis Report

- **Tool Used**: Bandit
- **Command**: `bandit -r .`



- **Comment:** There are no real issues that we can address because all these issues are either:
    - **Issue: [B501:request_with_no_cert_validation]**: which is not really an issue in our case since this is not a real production app.
    - **Issue: [B201:flask_debug_true]:** same reason.
    - And some other issues that are related to the mock services, like using mock auth data results in **Issue: [B105:hardcoded_password_string]**

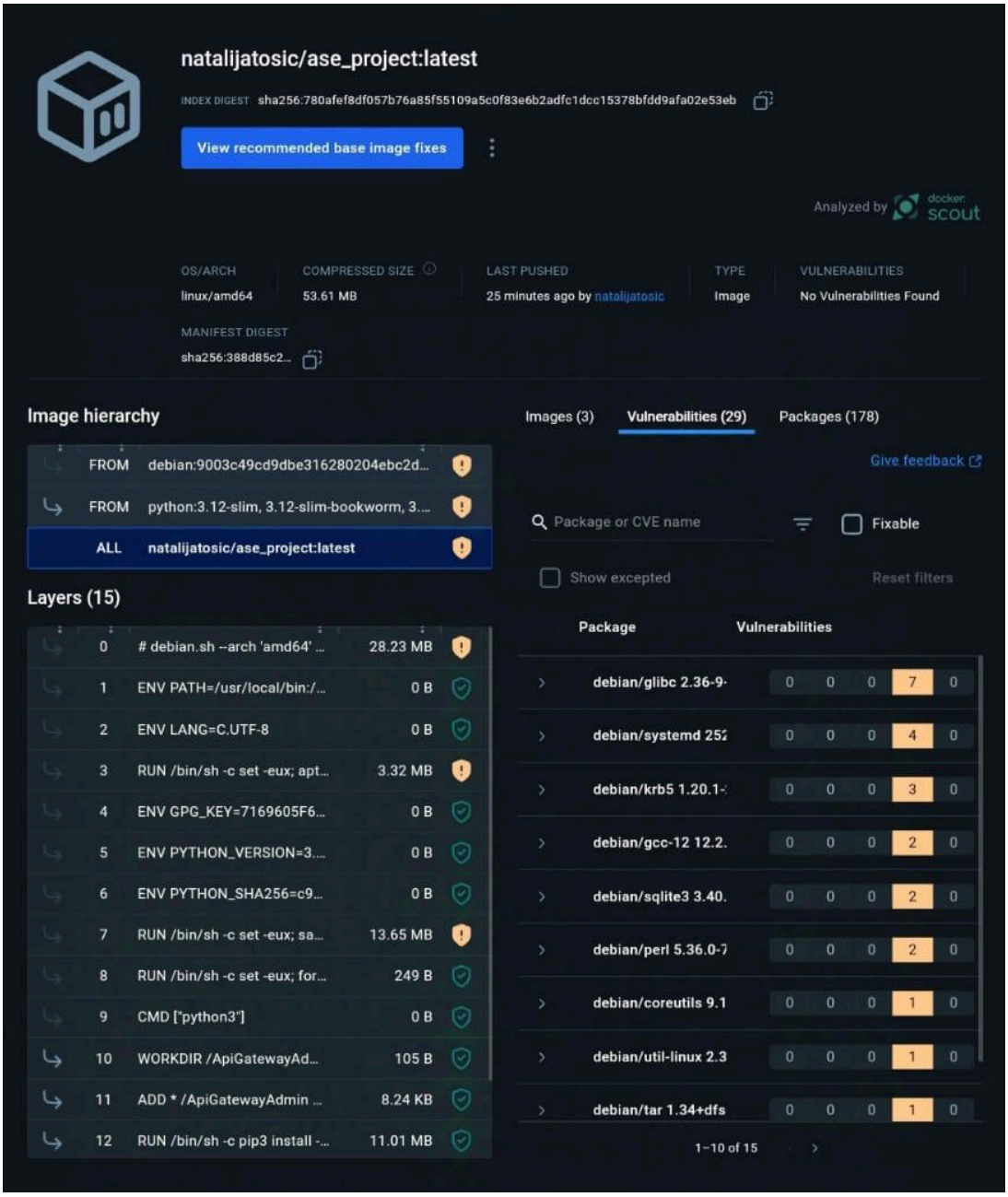
8.2. Docker Image Security

For the Docker images used in this project, we performed a vulnerability analysis using **Docker Scout**. We merged all the images of the microservices into a single Docker image to simplify the process. Below are the details of the analysis:

- **Docker Hub Repository:** `natalijatosic/ase_project`
- **Image Analyzed**: `natalijatosic/ase_project:latest`
- **Number of Vulnerabilities**: **29** vulnerabilities were detected in the scanned image.
- **Packages Affected**: A total of **178 packages** were analyzed, with the vulnerabilities distributed across various dependencies.

The analysis revealed vulnerabilities in packages such as `glibc`, `systemd`, `sqlite3`, `coreutils`, and more.

The following screenshot illustrates the analysis results, showcasing the detected vulnerabilities for the specified image:



9. Additional Features

Green User Story: Different Levels of Roll Rarity

- **What is this feature?** This feature allows players to choose **different levels** of roll rarity when performing a gacha roll. The available levels are "Common," "Rare," "Epic," and "Legendary," each with a **different probability** of rolling a rare gacha item and a **different roll price**.
- **Why is this feature useful?**
    - It provides players with more choices, allowing them to customize their chances of getting rare items.
    - It motivates players to spend more in-game currency to roll higher rarity gacha items.
    - It introduces strategy, as players can decide between rolling a cheaper, more common gacha or a rarer, more expensive one.
- **How is it implemented?** The system uses a configuration file (JSON format) to store the probability and price for each roll rarity. When the player selects a roll rarity (e.g., "Epic"), the system calculates the corresponding probability and roll price. The backend validates the selected rarity, adjusts the price accordingly, and executes the gacha roll with the adjusted probability. The roll's result depends on a weighted random selection based on the rarity percentage, ensuring that rarer gachas are less likely to be rolled, but players have a higher chance when they choose rarer levels.

Green User Story: Banning a player account

- **What is this feature?** This feature allows admin to ban a specific player by his ID, so that his status becomes "BANNED"
- **Why is this feature useful?**
    - It allows the admin to remove unwanted players that are either cheating or not playing by the rules
- **How is it implemented?** This function handles banning a player by updating their status in the system while preserving specific user data. First, it checks if the request is authorized by verifying it with an authentication microservice. If

authorized, the function validates that the request contains a `status` field, which indicates the new status for the player. It then retrieves the user's existing data, such as their in-game currency and profile picture, from a database microservice. If the user exists, this data is combined with the new status into an update payload and sent back to update the user’s record. The function provides appropriate error responses for unauthorized requests, missing users, or invalid input. This ensures a secure and efficient process for managing user bans.