Advanced Software Engineering

University of Pisa, Department of Computer Science
Project Report
Academic Year 2024-2025

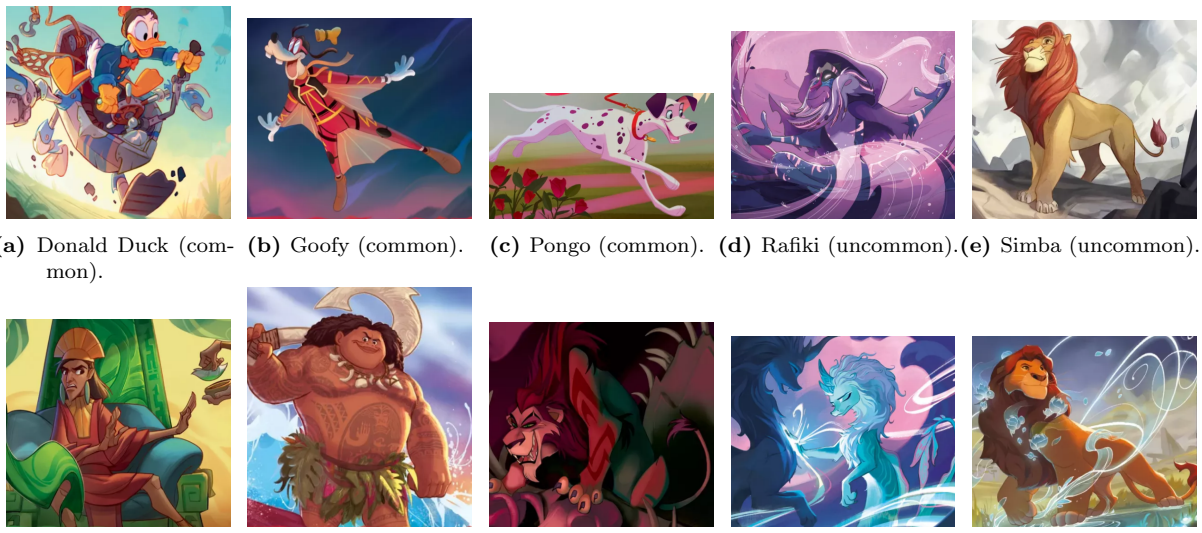
Marco Carollo
Francesca Funaioli
Claudio Moramarco

Gachana repository

Contents

1 Gacha Overview

The Gachana application implements a gacha game that allows players to buy gachas and sell them into auctions. The 20 gachas available in the application are inspired by the Lorcana game. Some of the gachas of the game are shown in Figure 1. When a player rolls for a gacha using in-game currency (Lorcano), they have a 40% probability of obtaining a *common* gacha, a 20% probability of obtaining a *uncommon* gacha, a 20% probability of getting a *rare* gacha, a 15% probability of getting a *super rare* gacha and a 5% probability of getting a *legendary* gacha.



(a) Donald Duck (common). **(b)** Goofy (common). **(c)** Pongo (common). **(d)** Rafiki (uncommon). **(e)** Simba (uncommon).

(f) Kuzco (rare). **(g)** Maui (rare). **(h)** Scar (rare). **(i)** Sisu (super rare). **(j)** Mufasa (legendary).

Figure 1: Gachas in the Gachana application.

2 Architecture

The application adopts the microservice architectural pattern and its architecture, represented using MicroFreshener, is shown in Figure 2. When executing the analyze tool of MicroFreshener, no smells were detected with the exception of the Endpoint-Based-Interaction-Smell. The system allows for two different types of users: players and admins. The access to each microservice is not direct, but it is mediated using an API Gateway implemented using Nginx. In particular, there are two different API gateways, one for the Docker network of the player and one for the Docker network of the admin. Each one of the microservices is associated with a database. The microservices and the databases are each deployed as Docker containers. In particular, each microservice is implemented through a Python image, while the databases are implemented through MySQL Docker images. The containers are then run using Docker Compose, which also defines a volume for each database in order to make data persistent when the containers are stopped. The application has been tested using Postman collections and automated workflows on the Github repository.

2.1 Microservices

The four microservices are Currency, Gacha, Market and User. As mentioned above, all microservices are implemented using Python images.

- The *Currency microservice* allows players to buy in-game currency using real money and to get a history of their transactions, while it allows admins to inspect all transactions and to get a specific user's transactions. A transaction here is intended as an exchange of real money for in-game currency.
- The *Gacha microservice* allows users to inspect the gachas they own, as well as checking which gachas they do not yet own and to roll for a gacha, while it allows admins to inspect the gacha

collection, to add, remove and modify specific gachas. It is connected to the Currency microservice, needed when rolling a gacha[1] and when a gacha is deleted[2].

- The *Market microservice* allows players to create an auction for a gacha they own, to see ongoing auctions, to make and accept offers, as well as seeing their own market history; it allows admins to see the details of an auction, modify it and get the whole market history. It is connected to all the other microservices[3].
- The *User microservice* allows players to create an account, login and logout, while it allows admins to get information on registered users, modify and delete their accounts.

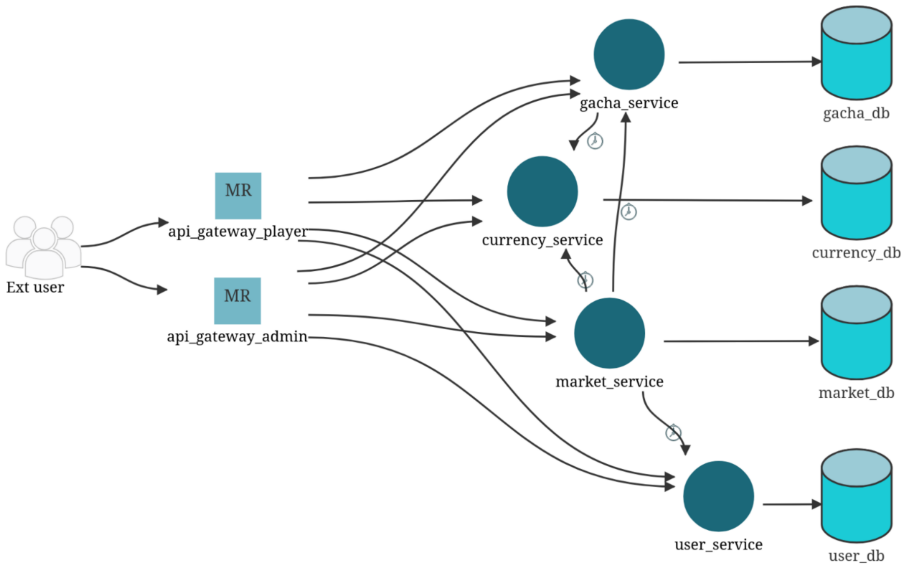


Figure 2: Gachana architecture represented in MicroFreshener.

2.2 Databases

The structure of the tables contained in the databases is the one shown in Figure 3. In particular, the *Currency DB* contains the tables `Wallets` and `Transaction_History`, which respectively store information on the amount of in-game currency and on the past transactions of a player. The *Gacha DB* contains the tables `Gacha` and `Collection`: the first one stores data on all the gachas available in the application, while the second stores information about users' collections. The *Market DB* contains the tables `Market`, `Market_History` and `Offers`, which respectively store data on ongoing auctions, the offers related to each auction and the data related to completed auctions. The *User DB* contains the table `UserData`, storing data on the users, such as their username, password, role, etc.

3 User Stories

Tables 1 and 2 show the endpoint implemented by the Gachana application, along with the user story implemented and the request body content.

[1] When rolling a gacha, an internal request is sent from the Gacha microservice to the currency microservice, invoking the `check_and_deduct` function to decrease the user's currency.

[2] When an admin deletes a gacha, a request is sent from the Gahca microservice to the Currency microservice, invoking the `refund` function to give back to the user the amount of currency a gacha is worth.

[3] It invokes the `login` and the `refund` endpoints, of the User and Currency microservices respectively, when invoking the endpoint `list`. It invokes the `collection` endpoint of the Gacha miscroservice when an user posts a new auction. It invokes the `check_and_deduct` endpoint of the Currency microservice. It invokes the `login`, `refund`, `collection/add`, `remove` and `add_currency`, from the User, Currency, Gacha, Gacha and Currency.

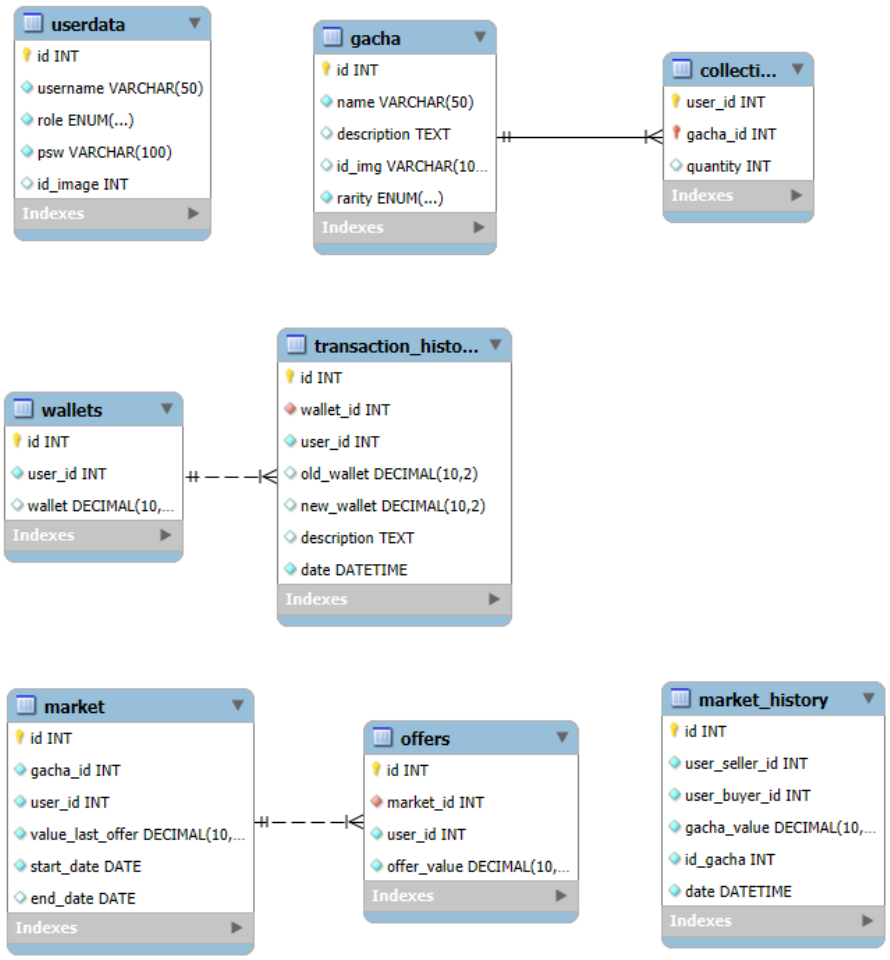


Figure 3: Database architecture

4 Market Rules

The Market microservice allows users to interact with a virtual market to manage auctions of gacha items. Players can *create auctions* to sell gacha items they own by specifying: the gacha they want to sell, a starting price and an end date. Auctioned items are temporarily removed from the player's inventory. Other players can *bid* on active auctions: bids must be higher than the current price or the last recorded bid. Obviously, the owners of the auction themselves cannot bid on their own auctions. The auction owner has the option to *accept a bid* at any time, as long as the auction has not yet expired. This action marks the auction close early and it allows the transaction to be finalized. In fact, when a bid is accepted, the gacha item is automatically transferred to the accepted bidder, ensuring that the item changes hands without further manual intervention. At the same time, the bid amount is deducted from the buyer's virtual balance and credited directly to the seller's account, thus completing the sales cycle. In the event that an auction reaches its end date without valid bids being accepted, the system automatically handles the closing. The gacha item remains with the original owner, as the auction is not concluded.

The system offers users an easy way to monitor their activity on the market. Each player can access the history of their auctions and bids, viewing the details of each transaction, including bids made, accepted or expired. For administrators, transparency is further extended: they can consult the complete history of all auctions in the market, obtaining a detailed overview of all activity. This level of traceability not only promotes a fairer and more organized market, but also helps in the management of any problems or disputes.

User Story	Endpoint	Request body
create game account	`POST /player/user/signup`	`username` and `psw`
login to the system	`POST /player/user/login`	`username` and `psw`
logout from the system	`POST /player/user/logout`	none
delete game account	`DEL /player/user/delete`	none
modify game account	`PATCH /player/user/update`	`username` and `psw`
see gacha collection	`GET /player/gacha/collection`	none
see info of owned gacha	`GET /player/gacha/collection/<gacha_id>`	none
see available gachas	`GET /player/gacha/collection/available`	none
see info of available gacha	`GET /player/gacha/collection/available/<gacha_id>`	none
roll a gacha	`POST /player/gacha/roll`	none
get gacha image	`GET /player/gacha/image/<image_name>`	none
buy in-game currency	`PATCH /player/currency/buy_currency`	`amount`, `card_number`, `expiry_date` and `cvv`
get own transactions	`GET /player/currency/transactions`	none
get own wallet	`GET /player/currency/wallet`	none
see auction market	`GET /player/market/list`	none
start an auction	`POST /player/market/new-auction`	`gacha_id`, `end_date` and `init_value`
bid for a gacha	`POST /player/market/new-bid`	`market_id` and `offer_value`
view transaction history	`GET /player/market/history`	none
see auction offers	`GET /player/market/<market_id>/offers`	none
accept an offer	`POST /player/market/accept`	`market_id` and `buyer_id`

Table 1: Players' User Stories with corresponding Gachana endpoint.

5 Testing

The Gachana repository contains unit, integration and performance tests. Each of these is implemented as one or more Postman collection, which can be found in the `docs` directory. Unit and performance tests also have Gthub Actions workflow files in the `./workflows` directory. One should note that all operations, with the exception of the `signup` endpoint, need an authentication token in the request body. This means that the first operation in each test is a `login` call, in order to obtain a valid token.

5.1 Unit Testing

Unit tests can be executed by running the following command

```
docker compose -f docker-compose.test.yml up --build
```

and then executing the Postman collection of the microservice one wants to test. The testing `docker-compose.test.yml` file runs the four microservices and their databases in separate networks, exposing the port of each microservice. This means that there are four docker networks: `user`, `currency`, `gacha` and `market`. When executing the unit test of microservices Gacha and Market (the only two which depend on other microservices), dependencies are mocked. It was chosen to run the whole docker compose to execute unit tests because it was the simplest way to maintain all secrets and https execution. Still, even if docker compose is used, the single microservice execute in isolation, linked only to their own database.

5.2 Performance Testing

The project utilizes Locust to execute performance testing on player endpoints related to gacha operations. High volumes of requests were simulated to validate the rarity distribution of gacha rolls and the efficiency of the involved endpoints. Authentication is required for accessing player-facing endpoints, as the system enforces the use of a valid JSON Web Token (JWT). Specifically, the `/player/user/login` endpoint is employed to generate the necessary JWT token, which ensures that only authorized users can access gacha services. The authentication process involves providing valid credentials, which are a username and a password. The following endpoints were enabled for testing:

User Story	Endpoint	Request body
login to the system	`POST /admin/user/login`	`username` and `psw`
logout from the system	`POST /admin/user/logout`	none
see all users	`GET /admin/user/list`	none
modify user account	`PATCH /admin/user/<user_id>/update`	`username`, `psw` and `role`
delete user account	`DELETE /admin/user/<user_id>/delete`	none
see gacha collection	`GET /admin/gacha/gachas-list`	none
modify a gacha	`PATCH /admin/gacha/update/<gacha_id>`	`name`, `rarity`, `description` and `name_img`
see a gacha	`GET /admin/gacha/<gacha_id>`	none
add a gacha	`POST /admin/gacha/add/<gacha_id>`	`name`, `rarity`, `description` and `name_img`
remove a gacha	`DELETE /admin/gacha/delete/<gacha_id>`	none
get gacha image	`GET /admin/gacha/image/<image_name>`	none
check user transaction history	`GET /admin/currency/transactions/<user_id>`	none
check all transaction history	`GET /admin/currency/transactions/list`	none
see auction market	`GET /admin/market/history/all`	none
see an auction	`GET /admin/market/auction-details/<market_id>`	none
modify an auction	`PATCH /admin/market/update-auction/market_id>`	`init_value`, `value_last_offer`, `start_date` and `end_date`
see auction offers	`GET /admin/market/<market_id>/offers`	none

Table 2: Admins' User Stories with corresponding Gachana endpoint.

Rarity	Assumed distribution	Actual distribution
Common	40%	$\sim$ 40%
Uncommon	20%	$\sim$ 20%
Rare	20%	$\sim$ 19%
Super Rare	15%	$\sim$ 15%
Legendary	5%	$\sim$ 5%

Table 3: Comparison of the distribution of gachas obtained with Locust.

- Gacha Roll `/player/gacha/roll`
- Gacha Collection `/player/gacha/collection`
- Gacha Available `/player/gacha/collection/available`
- Gacha Specific information `/player/gacha/collection/available/<gacha_id>`

The test also focused on the `/player/gacha/roll` endpoint, where the distribution of rarities, shown in Table 3, was verified to align with the assumptions made in Section 1.

The system demonstrated efficiency and scalability, handling a high number of concurrent requests without issues. The load test results are illustrated in Figure 4, which displays key metrics for the gacha service player endpoints. The number of requests per second managed by the system is shown, with the green line indicating stable throughput following an initial spike. No failures were observed, as represented by the absence of a red line.

6 Security

6.1 Data

For what concerns input sanitization, one example is the `new-auction` endpoint in the Market microservice. In fact it needs in input the data on the auction to start, that is the `gacha_id`, the `end_date` and

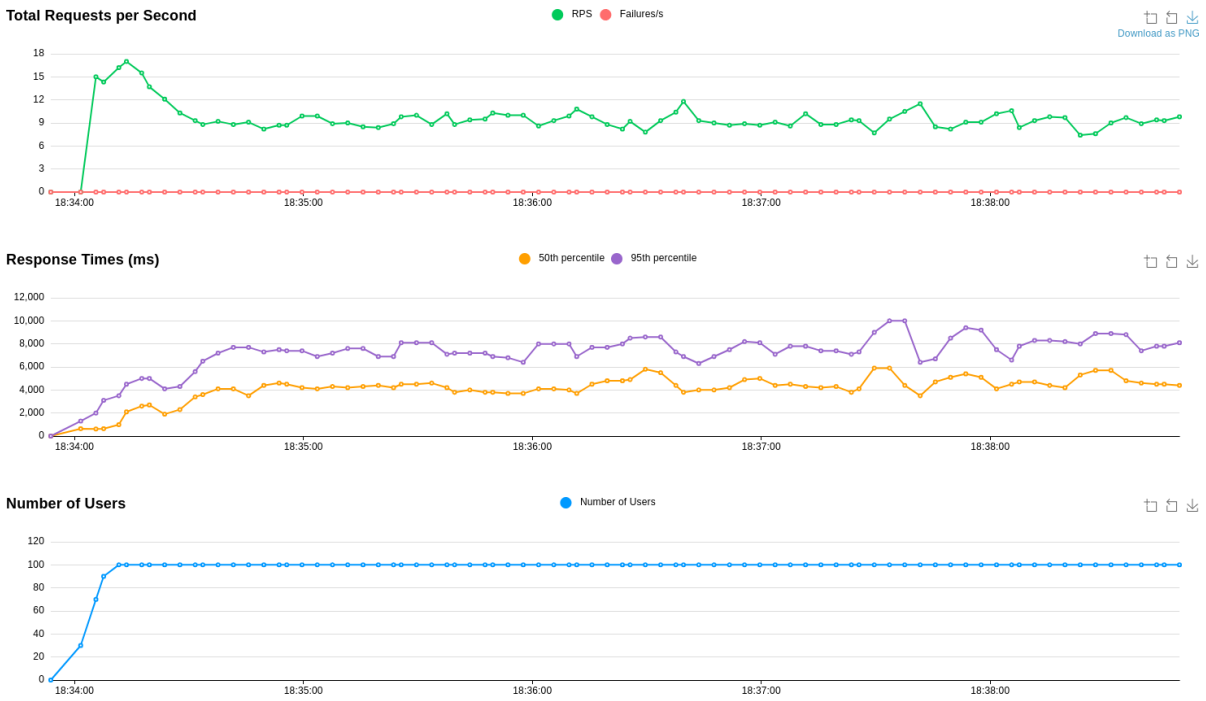


Figure 4: Locust performance test results.

the `init_value`. All these input data are sanitized using a `sanitize_input` method, which checks the type of each variable, along with some other minor checks.

The only data encrypted at rest are the users' passwords. They are stored in the User database and they are encrypted in the `signup` endpoint, when the users first provide them. They are only decrypted in the login method which needs to check whether the provided password is the same as the previously stored password for that user.

6.2 Authorization and Authentication

The chosen authorization and authentication model is distributed. In fact, there is no special microservice devoted to the authenitication/authorization process, but all microservices can decode the token (JWT). When a user successfully logs in, they receive their token, which must be inserted into the `Authorization` header of each subsequent request to the Gachana application. The endpoint which receive the request then decodes the token to get information on the user, like the `user_id` or the `role`, which can be player or admin. The token is not stored in any of the application databases, but is provided with each request. When a user successfully calls the `logout` endpoint, their token is inserted into a blacklist table. Subsequent requests using the blacklisted token are then rejected. An example of a token is the following one:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJyb2xlIjoiQWRta-
W4iLCJleHAiOjE3MzM0MTEyOTJ9.bCKZ2t4z1o-oaGkA6JiYQskgxHfHCeCEyiyCD_9x4LY
```

6.3 Analyses

The Gachana app was tested using both Bandit and the Docker Scout tools. The report of the Bandit test is shown in Figure 5, while the result of executing the Docker Scout analysis are shown in Figure 6. The link to the Docker Hub repository is the following https://hub.docker.com/r/ffra/gachana. There where 3 high severity vulnerabilities shown on the first run of Docker Scout. Two of them were due to using an old Python image (3.9) for the microservices, while the third was due to using an old version of the `mysql-connector-python` (8.1.0). Updating both respectively to their 3.12 and 9.1.0 versions completely resolved the vulnerabilities.

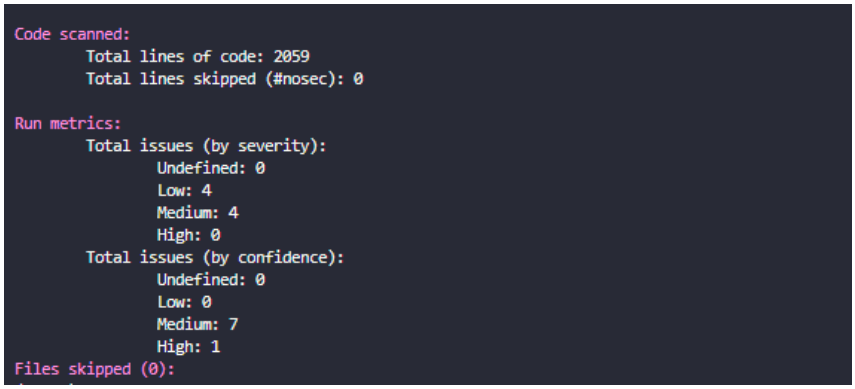


Figure 5: Bandit final table.

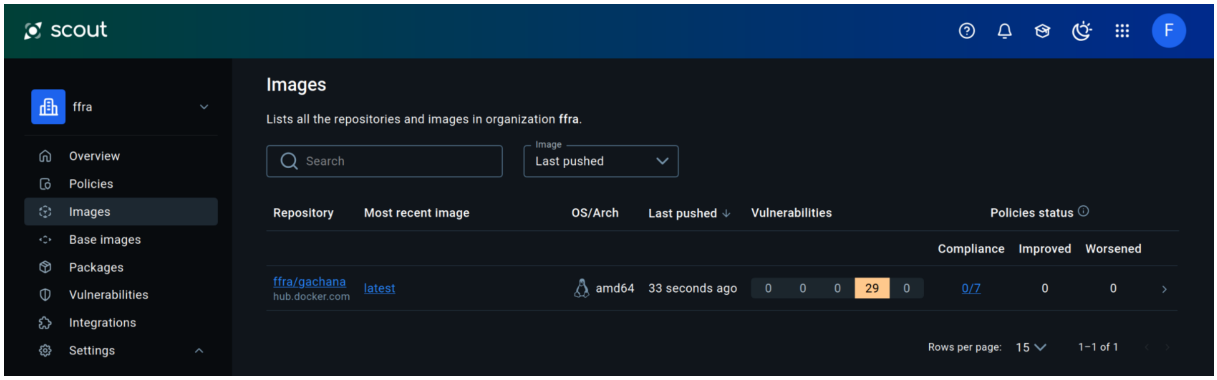


Figure 6: Docker Scout dashboard.