GCTA ASE 2024-2025 Project Report

Group Name: GCTA

Members:

- Francesco Copelli (f.copelli@studenti.unipi.it)
- Simone Conti (s.conti27@studenti.unipi.it)
- Nicola Lepore (n.lepore@studenti.unipi.it)
- Purity Siantayo Nkaiwatei (p.nkaiwatei@studenti.unipi.it)

December 6, 2024

Contents

1 Gachas Overview

The GCTA project involves a gacha game backend system. The core functionality revolves around managing a collection of gacha items, categorized by rarity and tied to players' in-game progress. Below is an illustration of the gacha system:

Figure 1: Example of Gacha illustration

Gacha Collection:

- **Super Treasure 2024 (11 items):** *Epic (E)* rarity, offering exclusive and highly desirable items.
- **Treasure 2024 (13 items):** *Rare (R)* rarity, providing valuable items.
- **Boulevard 2024 (30 items):** *Rare (R)* rarity.
- **Mainlines 2024 (345 items):** *Common (C)* rarity, less valuable than the other gachas.

In-game Currency: The currency used in this gacha system is called *Octane*, which players use for rolling gachas, participating in auctions, and other transactions.

Player Gacha Collection: Each player has a personalized inventory of gacha items, linked to their profile. The items in the collection are classified on the basis of their rarity and acquired through rolls or auctions. The backend architecture ensures seamless management of the inventory, utilizing database relationships and microservices for efficient operations.

Figure 2: In-game Currency rappresentation

2 Architecture

The GCTA system is designed using a microservices architecture, leveraging Flask and Docker for modularity and scalability. Communication between services is managed through distinct networks and API gateways, ensuring isolation and secure data exchange. The updated architecture is illustrated in the diagram below:

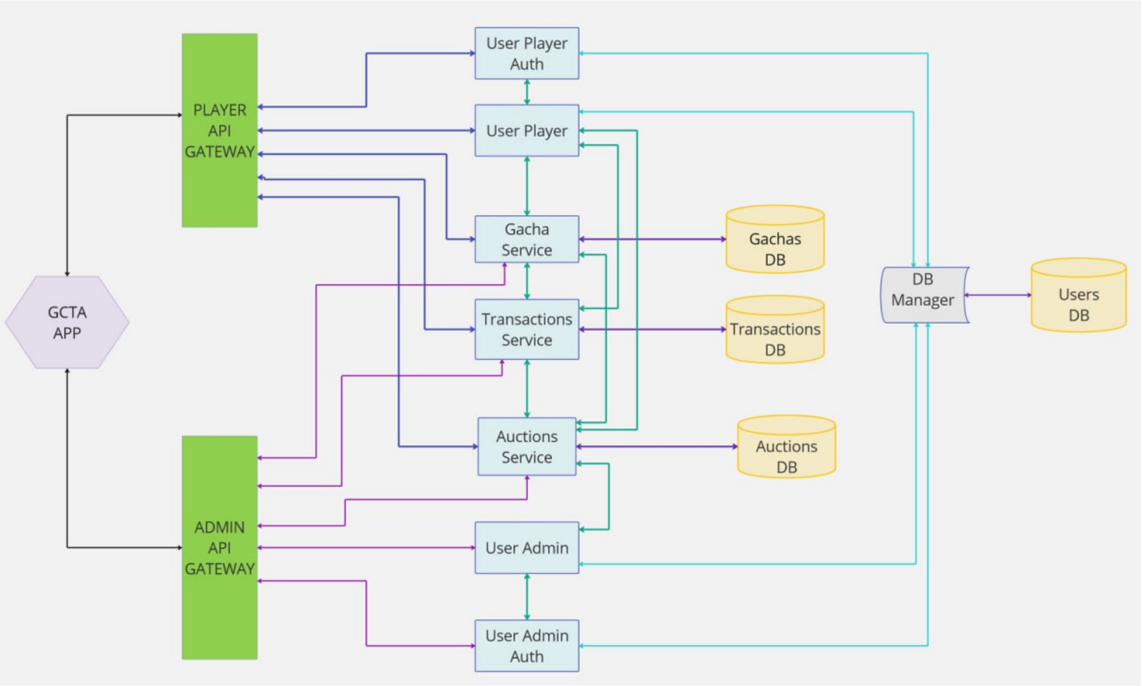


Figure 3: GCTA Architecture Diagram

2.1 API Gateways

The system uses two primary gateways to route requests to the appropriate microservices:

- **gateway_user:** Handles player-related functionalities, such as user profile management, gacha operations, and auction participation. This gateway communicates over the `users-network`.
- **gateway_admin:** Manages administrative functionalities, including user management and system oversight. This gateway operates within the `admin-network`.

2.2 Core Services

The architecture comprises several core services, separated by functionality:

- **User Services:**
  - **user_admin:** Provides functionality for managing admin users.
  - **user_admin_auth:** Handles admin authentication processes.
  - **user_player:** Manages player-related functionalities, such as profile updates and inventory access.
  - **user_player_auth:** Ensures secure player authentication.
- **Gacha Functional Services:**
  - **transaction:** Processes transactions, such as top-ups, bids, and purchases. It communicates across `users-network`, `admin-network`, and `transaction_db-network`.
  - **auction:** Manages auction-related functionalities, interacting with `users-network`, `admin-network`, and `auction_db-network`.
  - **gacha:** Handles gacha mechanics, including item rolls and inventory management. It operates on `users-network`, `admin-network`, and `gacha_db-network`.
- **Database Management:**
  - **db-manager:** Acts as an intermediary for secure access to the users database, communicating via the respective network (`user_db-network`).

2.3 Network Distribution

The system isolates microservices and databases into distinct networks for enhanced security and scalability:

- **users-network:** Connects player-related services (`gateway_user`, `user_player`, and `user_player_auth`) with the other microservices that have endpoints only for a normal user (`gacha`, `auction`, `transactions`).
- **admin-network:** Connects admin-related services (`gateway_admin`, `user_admin`, and `user_admin_auth`) with the other microservices that have endpoints only for admin users (`gacha`, `auction`, `transactions`).
- **Database-Specific Networks:**
  - `user_db-network`: Isolates `users_db` allowing only requests from `db-manager`.
  - `gacha_db-network`: Isolates `gacha_db` allowing only requests from the `gacha` service.
  - `auction_db-network`: Isolates the `auctions_db` allowing only requests from the `auction` service.
  - `transaction_db-network`: Isolates `transactions_db`allowing only requests from the `transaction` service.

2.4 Secrets and Configuration

Sensitive data, such as certificates, database credentials, and encryption keys, are managed using Docker secrets, as reported in the following section.

Secrets and sensitive files

The following sensitive files are required for secure operations:

- **SSL/TLS Certificates and related keys**: Certificates used for secure communication, stored in the `certificates` directory. Specific files include:
  - `cert.pem`, `key.pem` (default SSL/TLS certificates used in the 2 gateways for the connection from the outside).
  - `ca-cert.pem` (Certificate Authority certificate for dbs connection).

  - `server-cert.pem`, `server-key.pem` (server-specific certificates for dbs).
  - `client-cert.pem`, `client-key.pem`, `client-req.pem` (client-specific certificates for connection to db).
  - Service-specific certificates and keys:
    - `auction_cert.pem`, `auction_key.pem` (Auction service).
    - `gacha_cert.pem`, `gacha_key.pem` (Gacha service).
    - `transactions_cert.pem`, `transactions_key.pem` (Transactions service).
    - `users_cert.pem`, `users_key.pem` (Users service).

- **Database Credentials**: Stored as plaintext files:
  - `db_user.txt` (database username).
  - `db_password.txt` (database password).

- **JWT Secrets**: The secret used by JWT is extracted from a file named `poetry.txt`, which contains the hidden secret in plaintext that will be extracted.

2.5 Data Flow and Relationships

- API gateways route requests to their respective services based on configurations in `nginx.conf`.

- Each service communicates with its corresponding database over its isolated network.

- The `db-manager` service ensures secure and efficient database access for the users service.

Networking: The use of multiple networks and strict access control ensures secure communication and minimizes the risk of unauthorized data access.

3 User Stories

User stories provide insight into the interactions users have with the GCTA system. These scenarios were pivotal in designing the microservices and database architecture, ensuring the system meets user expectations and needs.

3.1 Player Stories

The following user stories detail the functionalities required for players:

Account Management:

- **AS A player, I WANT TO create my game account/profile, SO THAT I can participate in the game.**
  - **Endpoint(s):** 4)`/register` (player_gateway, player_auth, dbm)
  - **Microservice(s) involved:** player_auth, dbm
- **AS A player, I WANT TO delete my game account/profile, SO THAT I can stop participating in the game.**
  - **Endpoint(s):** 5) `/user/delete` (player_gateway, player_auth, dbm)
  - **Microservice(s) involved:** player_auth, dbm
- **AS A player, I WANT TO modify my account/profile, SO THAT I can personalize my experience.**
  - **Endpoint(s):** 6) `/user/update` (player_gateway, player_auth, dbm)
  - **Microservice(s) involved:** player_auth, dbm

Gacha Interactions:

- **AS A player, I WANT TO roll a gacha, SO THAT I can acquire new items for my collection.**
  - **Endpoint(s):** 13) `/gacha/roll` (player_gateway, gacha, player)
  - **Microservice(s) involved:** Gacha, player
- **AS A player, I WANT TO view my gacha inventory, SO THAT I can track my acquired items.**
  - **Endpoint(s):** 9) `/gacha/inventory/{user_id}` (player_gateway, Gacha, UserInventory)
  - **Microservice(s) involved:** Gacha, player

Market Participation:

- **AS A player, I WANT TO participate in auctions, SO THAT I can bid for rare gacha items.**
  - **Endpoint(s):** 18) `/auctions/bids` (player_gateway, Auctions, player)
  - **Microservice(s) involved:** Auctions, player
- **AS A player, I WANT TO list my gacha items for auction, SO THAT I can trade or sell them to other players.**
  - **Endpoint(s):** 19) `/auctions/list` (player_gateway, Auctions, Gacha, player)
  - **Microservice(s) involved:** Auctions, Gacha, player

3.2 Admin Stories

The following user stories define the functionalities required for administrators:

Account and Profile Management:

- **AS AN administrator, I WANT TO log in and out of the system, SO THAT I can access and leave the game environment.**
  - **Endpoint(s):** 4) `/login` (admin_gateway, admin_auth, dbm)
  - **Microservice(s) involved:** admin_auth, dbm
- **AS AN administrator, I WANT TO check all user accounts/profiles, SO THAT I can monitor user activities.**
  - **Endpoint(s):** 5) `/admin/users` (admin_gateway, dbm)
  - **Microservice(s) involved:** dbm
- **AS AN administrator, I WANT TO check or modify a specific user account/profile, SO THAT I can update or rectify issues as needed.**
  - **Endpoint(s):** 6) `/admin/user/{user_id}` (admin_gateway, dbm)
  - **Microservice(s) involved:** dbm

Game Management:

- **AS AN administrator, I WANT TO manage gacha items (add, update, or remove), SO THAT the system remains up-to-date and balanced.**
  - **Endpoint(s):** 10) `/gacha/add`, `/gacha/update`, `/gacha/delete` (admin_gateway, Gacha, Admin)
  - **Microservice(s) involved:** Gacha, Admin
- **AS AN administrator, I WANT TO monitor auction activities, SO THAT I can ensure fair play and compliance.**
  - **Endpoint(s):** 15) `/auction/update` (admin_gateway, Auctions, Admin)
  - **Microservice(s) involved:** Auctions, Admin

3.3 Implementation Insights

The user stories influenced the system's design adopted during the implementation of GCTA system:

- **Microservices Design:** Specific services like `user_player`, `user_admin`, and `gacha` were developed to address player and admin needs.
- **Database Schema:** Relationships between users, gacha items, auctions, and transactions were modeled based on user interactions.
- **API Design:** Endpoints were created for each functionality, ensuring players and admins could seamlessly interact with the system.

Included in this documentation is only a subset of user stories implemented. A complete list of the API implementation for all the required user stories is included in the `openapi` documentation.

4 Market Rules

This section outlines the rules governing the in-game marketplace, including auction policies, real money transactions for purchasing game currency, and the mechanics for acquiring new gachas. All players are expected to adhere to the following guidelines to ensure a fair and enjoyable experience for everyone:

1. **Auction Policies:**
    - Auctions are conducted through a designated marketplace interface. Players may list items for sale and set a starting bid.
    - Players can place bids on listed items, which is an amount higher than the current highest bid and the highest bidder at the end of the auction wins the item.
    - Players wishing to list an item for auction must ensure the item is unlocked and available for trading. If an item is locked, it cannot be listed until it is unlocked. The system will automatically lock the item once an auction is created, ensuring it cannot be sold or transferred until the auction ends.
    - Once an auction ends, the system will automatically transfer the item to the winning bidder and deduct the appropriate amount of in-game currency from the buyer's account. The seller will receive the bid amount, and the gacha will be transferred to the buyer's inventory.
    - When a player wants to place a gacha item in an auction, the gacha is first checked if it is in an active auction to prevent players from placing multiple auctions on the same gacha item
    - In the event of a higher bid during the auction, the money from the previous bid will be refunded to the previous bidder, ensuring fairness.
2. **Real Money Transactions for Buying Game Currency:**
    - Players can purchase the in-game currency, *Octane* through a specific endpoint.
    - The amount of *Octane* purchased will be credited to the player's account upon successful transaction.
    - Players can use *Octane* to participate in auctions or roll for new gachas

- All real money transactions are final. No refunds or exchanges will be provided.
- It is strictly prohibited to exchange *Octane* or in-game items for real-world money or other external benefits. Such activities will result in penalties or account suspension.

3. **Rolling for New Gachas:**

   - Players can spend *Octane* to roll for new gachas, which may include items of varying rarity.
   - The probability of receiving higher-rarity items (e.g., Epic, Rare) is determined by the system..
   - Only players can roll for new gachas at any time, provided they have sufficient *Octane*.
   - Each roll will be recorded in the player's inventory, and the player will receive the item corresponding to the roll outcome.

4. **Account Deletion and Market Continuity:**

   - If a player deletes their account, any active auctions or owned in-game currency are transferred to a management account.
   - The management account takes over ownership of the player's active auctions, as well as any *Octane* or other in-game currency. This ensures that the marketplace remains unaffected for other players.
   - Any bids placed on active auctions will remain valid, and the marketplace will continue to function normally for all other players, without disruption.
   - The management account acts solely as a mediator to ensure fairness and prevent the market from being influenced by the deletion of a user account.

5. **Initial Currency Grant:**

   - Each player is granted an initial amount of *Octane* upon account creation.
   - This initial currency can be used to participate in auctions and roll for gachas
   - The amount of initial currency provided will be predefined and may be subject to change at the discretion of the game administrators.

5 Testing

This section outlines the approaches used for unit and integration testing, as well as the continuous integration (CI) pipeline configuration.

5.1 Testing Methods

- **Unit Testing:** Individual APIs were tested using Postman to verify their functionality. Positive and negative test cases were executed to ensure error handling and boundary conditions are properly managed.
  - All APIs passed individual tests using Postman, confirming the correctness of responses and adherence to specifications.
- **Integration Testing:** Services were tested together using postman end-to-end tests to confirm that the entire system works as expected. Data flow and interactions between microservices were checked to ensure they work without any glitches.
  - Integration testing validated seamless communication between microservices, with no issues detected.
- **Isolation Testing:** Services were tested independently by mocking both database data and dependencies on other services. This approach ensured that the logic within each service was validated while mirroring the database data and other services as close as possible.
  - Isolation tests, when run on postman, were all successful which validated that each of the service's logic and functionality works when it is tested independently.
- **Performance and Load Testing:** Locust was used to test concurrent users interacting with the GCTA system to evaluate the system's performance under heavy usage. The test results demonstrate the system's scalability and robustness under high load conditions: These are the key results:
  - No failures were recorded during testing, when there was a peak of up to **300 concurrent users** making requests.
  - The tests used **3 player profiles** and **3 admin profiles**, each executing all possible operations, including authentication, gacha rolls, auction participation, and data retrieval.

- At a load of **300 concurrent users**, there were a few failures related to the response time since we had a limit of 20 seconds per response. This made the system to be unstable due to the many requests that are performed simultaneously. This triggers the **circuit breaker** which temporarily limits requests to the databases.

Figure 4 shows an image of the results of locust tests

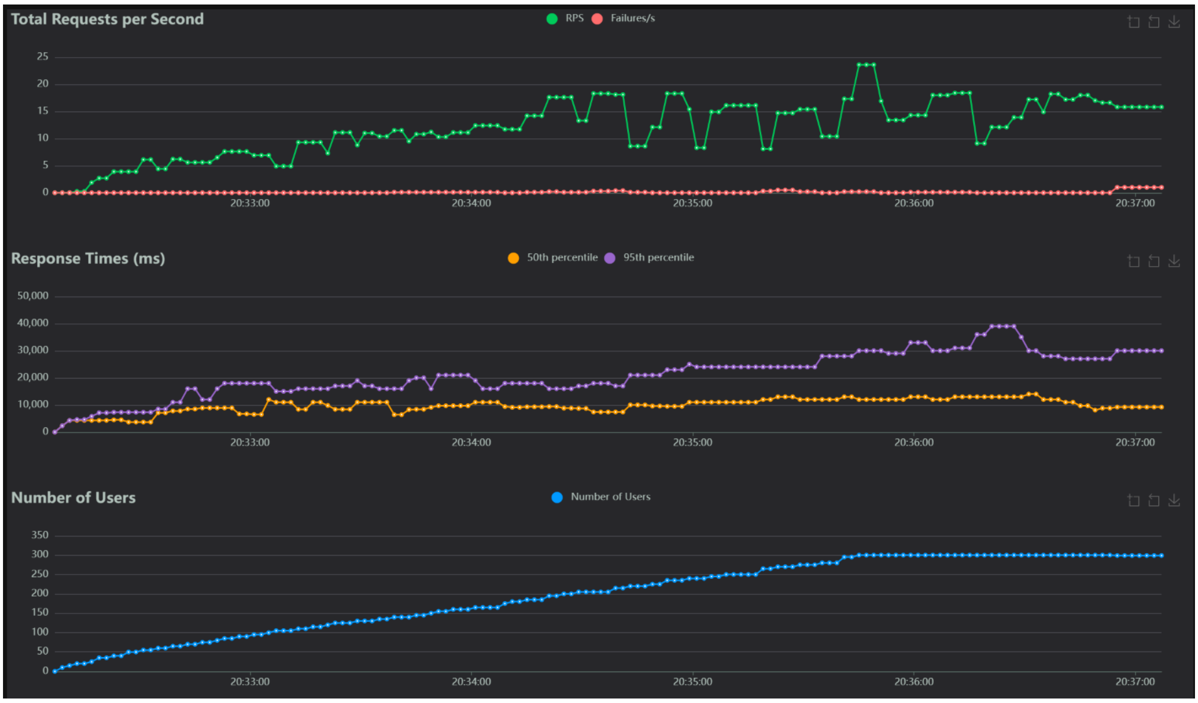


Figure 4: Test Result of Locust Test

5.2 Continuous Integration (CI) Pipeline

The Continuous Integration (CI) pipeline has been implemented using GitHub Actions to automate the building, testing and integration of the project. The CI process is outlined below:

- **Trigger:** The pipeline is triggered on every `push` event to the `main` branch.
- **Environment Variables:**
  - `REGISTRY`: Docker registry URL.

  - `IMAGE_NAME`: Docker image name derived from the GitHub repository.
  - `SHA`: Commit SHA from the pushed changes.

- **Jobs:**
  - **Build and Test Job:**
    - **Set Up Workspace:**
      - The codebase is checked out using `actions/checkout`.
      - For the `main` branch, a custom workspace is set up at `/tmp/main_branch_workspace`.
    - **Docker Configuration:**
      - Docker is configured using `docker/setup-buildx-action`.
      - Login to Docker registry using credentials stored as GitHub Secrets (`DOCKER_USER`, `DOCKER_PAT`).
      - Docker Compose is installed to manage multi-container applications.
      - Docker services are built and started using `docker-compose up`.
    - **Service Validation:**
      - Docker Scout is used to perform quick vulnerability scans and provide recommendations.
      - A waiting mechanism ensures that services are ready before tests are executed.
    - **API Testing:**
      - The Postman CLI is installed to run API tests.
      - The Postman API key is used to authenticate and execute the tests against the services.
    - **Artifacts and Cleanup:**
      - Docker services are stopped using `docker-compose down`.

The CI pipeline successfully executed on every push to the main branch, ensuring automated testing and consistent performance monitoring

6 Security – Data

In the GCTA system, several security measures were taken during implementation to ensure the system is secure while handling user and game data.

6.1 Input Sanitization

Some of the critical inputs sanitized in the system are the **user-provided email address** and **username** which are used during player registration and profile updates. These inputs:

- **Represents:** The email address used to uniquely identify users and communicate account-related information.
- **Used by Microservices:** `user_player` and `user_admin`.
- **Sanitization:** The email input and username are validated and sanitized to prevent injection attacks and ensure data consistency:
  - Checked against a regular expression to verify format compliance (e.g., `name@example.com`).
  - Stripped of extraneous whitespace or special characters.
  - Stored in a standardized, lowercase format in the database to prevent duplicates.

6.2 Data Encryption at Rest

Several critical pieces of data are encrypted to ensure confidentiality and integrity. These include:

- **Encrypted Data:**
  - **Passwords:** Used for authenticating players and admins.
  - **Session Tokens:** Used for maintaining user sessions across microservices.
- **Storage:** Encrypted data is stored in the `db-manager`'s relational database.
- **Encryption/Decryption:**
  - **Passwords:** Encrypted using a one-way hashing algorithm (e.g., bcrypt) before being stored. Hashing ensures that even if the database is compromised, passwords cannot be retrieved.

  - **Session Tokens:** Encoded when stored and decoded at runtime within the respective microservices.

These measures ensure the GCTA system maintains high standards of data protection and minimizes the risks of unauthorized access or data leakage.

7 Security – Authorization and Authentication

The GCTA system ensures secure access to its services through robust authentication and authorization mechanisms. Below are the key methods implemented:

7.1 Authentication

The system employs **JWT-based session management** for authenticating users and maintaining secure sessions. Each JWT contains:

- User-specific data such as `user_id` and `user_type`.
- An expiration timestamp to ensure tokens are only valid for a limited time.
- A signature generated using the system's `SECRET_KEY` to prevent tampering.
- Additional claims, such as `scope`, tailored to the user's role.

Token Generation: JWTs are created using the `generate_session_token` function, which includes:

- A **header** specifying the algorithm (`HS256`) and type (`JWT`).
- A **payload** containing claims such as:
  - `jti` (Unique Identifier): Ensures each token is unique.
  - `iss` (Issuer): Identifies the system generating the token.
  - `sub` (Subject): Indicates the type of access (e.g., `PLAYER_access` or `ADMIN_access`).
  - `iat`, `nbf`, `exp`: Define the token's creation time, validity start, and expiration.
  - `aud` (Audience): Specifies the intended audience of the token.
- A signature to verify the token's authenticity.

For system-level operations, the `generate_session_token_system` function creates tokens with `SYSTEM` as the user ID and type.

7.2 Authorization

Authorization is implemented using **role-based access control (RBAC)**, ensuring users have appropriate privileges based on their roles:

- **Player:** Tokens for players include:
  - `scope`: `game_access`
  - `user_type`: Player
- **Admin:** Tokens for admins include:
  - `scope`: `admin_panel`
  - `user_type`: Admin
- **System:** Tokens for system operations include:
  - `scope`: `system_operations`
  - `user_type`: System

Token Validation: Incoming tokens are validated using the `decode_session_token` function. This process ensures:

- The token was issued by the system and has not been tampered with.
- The token has not expired (`exp` claim).
- The audience (`aud`) matches the expected value (`ALL`).

7.3 Hybrid Architecture: Centralized with Distributed Elements

The GCTA system employs a hybrid architecture with a strong inclination towards centralization for handling authentication and authorization. Below is a detailed analysis:

7.3.1 Centralized System

JWT management and verification in the GCTA system are centralized due to the following reasons:

- **Dependence on the Auth microservice:** Each endpoint delegates the validation of JWT tokens and associated user details to the `Auth` microservice. This means that the validity of a token is not independently verified by the individual microservices but relies on an external centralized system.
- **Single Point of Verification:** The `Auth` microservice acts as a single point of failure. If `Auth` becomes unavailable, the system cannot authenticate users, as all token validation requests pass through it.

7.3.2 Distributed Elements

Despite its centralized nature, the GCTA system incorporates distributed elements:

- **Authorization Logic in Microservices:** Each microservice locally implements authorization logic based on user permissions. Once the token and user existence are confirmed by `Auth`, the individual microservices independently enforce access control by means of the decorator.
- **Potential for Independence:** If tokens were validated directly within each microservice, bypassing the need for `Auth` in every request, the system could function as a fully distributed architecture. However, this is not currently implemented.

7.3.3 Conclusion

The GCTA system's architecture is classified as **centralized** due to the following reasons:

- JWT verification always occurs through the `Auth` microservice, which acts as a single control point for all validation requests.
- Microservices are not fully autonomous in managing authorizations, as they rely on the responses from `Auth`.

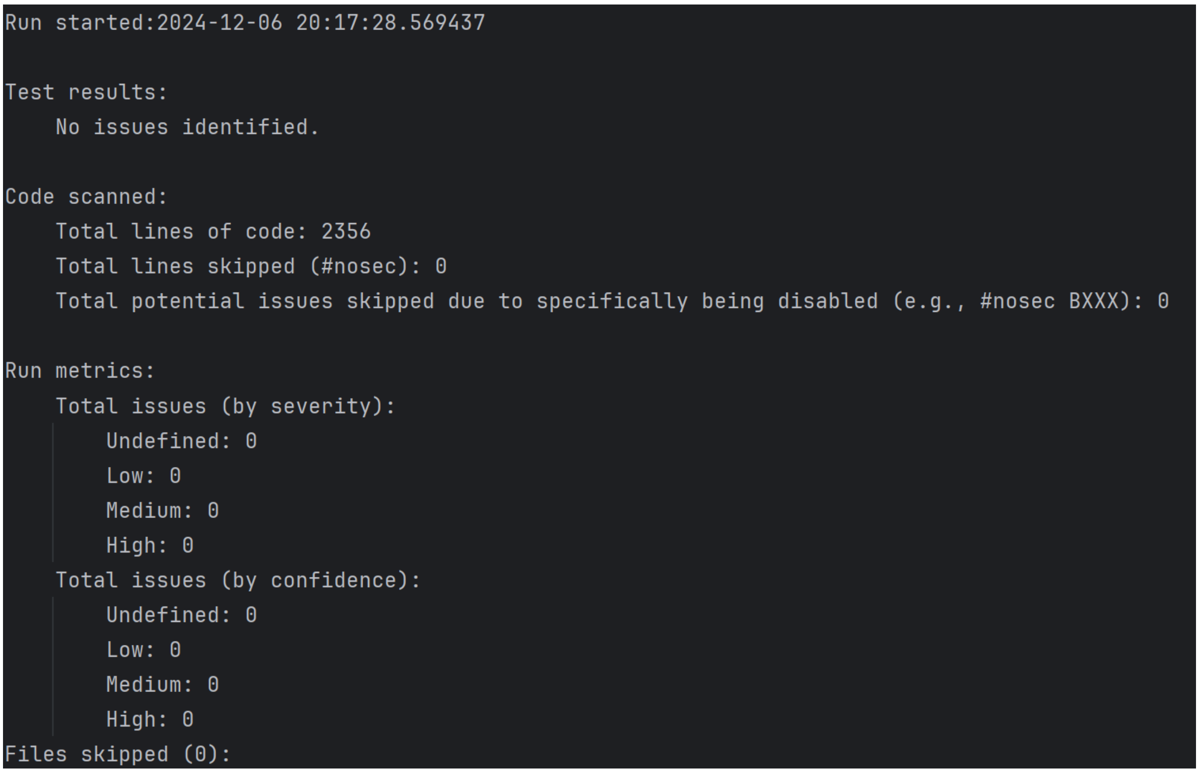
7.4 Security Measures

- Tokens are scoped for specific roles, reducing the risk of privilege escalation.
- Expired or invalid tokens are immediately rejected.
- Role-based claims (`scope`) ensure granular access control.

8 Security – Analyses

8.1 Bandit

To check any possible vulnerabilities in the code, we used the bandit tool for python.Figure 5 shows an image of the results of bandit.



```
Run started:2024-12-06 20:17:28.569437

Test results:
    No issues identified.

Code scanned:
    Total lines of code: 2356
    Total lines skipped (#nosec): 0
    Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX): 0

Run metrics:
    Total issues (by severity):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 0
    Total issues (by confidence):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 0
Files skipped (0):
```

Figure 5: Test Result of Bandit

As we can see from the image we have 0 high, medium, and low vulnerabilities. The command used to run the tool is the following (used in the root directory of the project):

```
bandit -r .  -x ./.venv/,./test_locust/,./mockup_test/,.locustfile.py-s=B501
```

We removed from the checks the B501 test which is related to certificate verification that we do not do because we are using self-signed certificates (we added the verify=False parameter in each request made by the microservices) and that would show vulnerabilities that are not related to the code itself but that are related to the requirement of the project. We also excluded .venv, mockup and locust test folder which would also give warning about the asserts that are not related to the application code itself but to the test code used.

8.2 Docker scout

The result of the docker scout is shown in the figure below.

Docker Scout Screenshot Link

As we can see from the image the services itself have only low vulnerabilities and none high or critical vulnerabilites. The dbs have medium and low vulnerabilities but no high or critical ones. These images can be seen in the public docker hub repository using the link below:

fcopelli28/gcta_ase_2024-2025