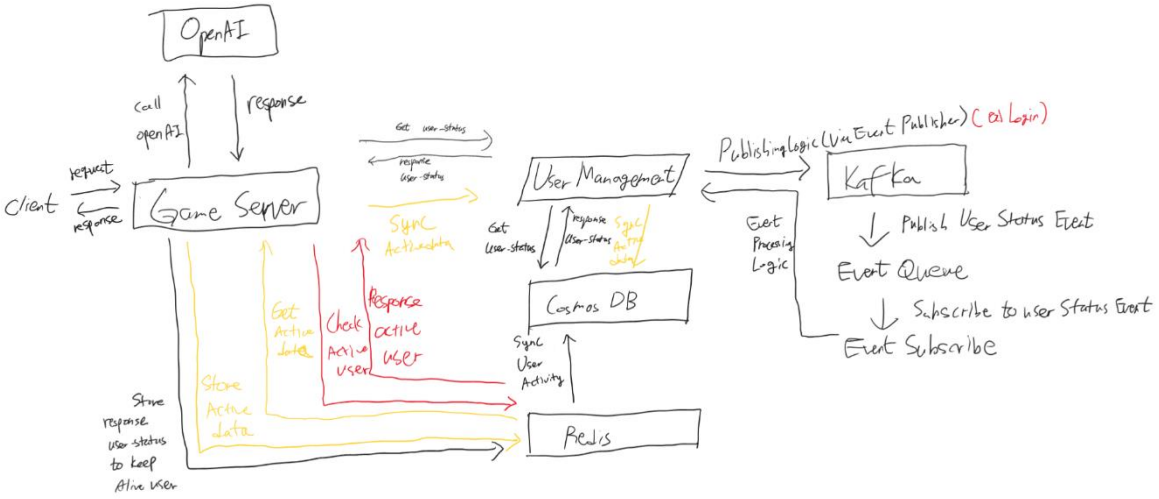
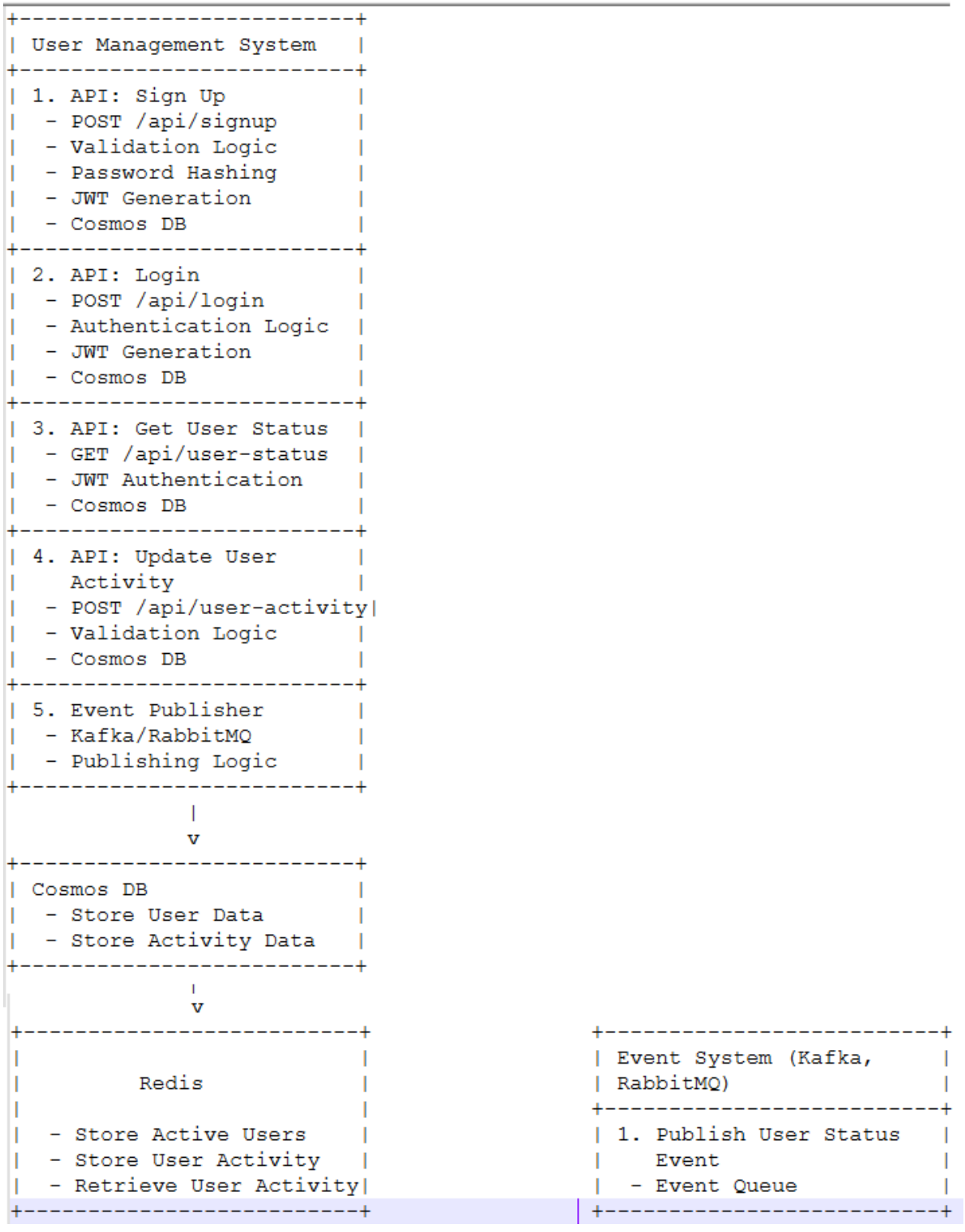
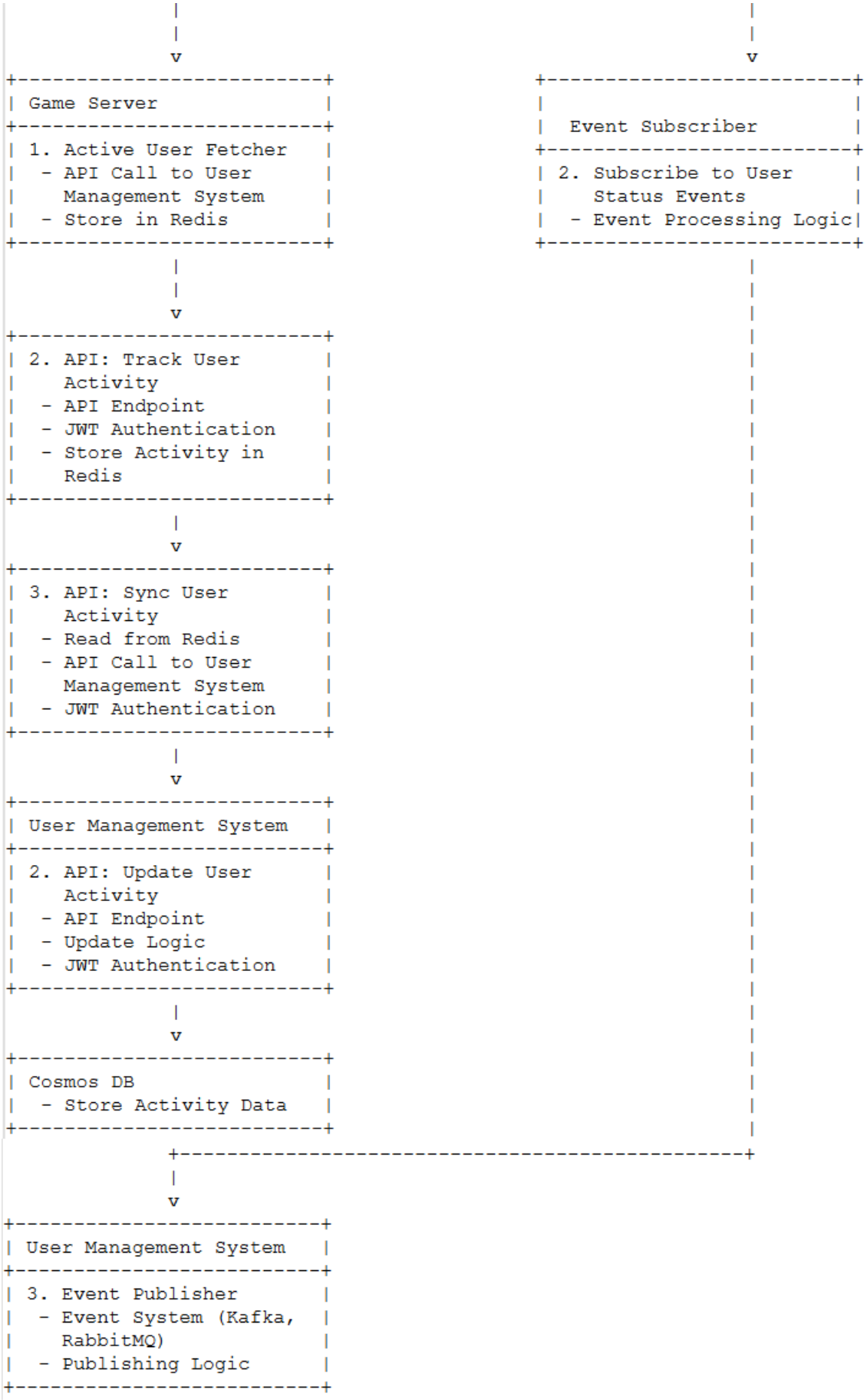

OpenAI
call openAI
response
client
request
response
Game Server
Get user_status
response user_status
Sync Activestate
User Management
Publishing Logic (Via Event Publisher) (at Login)
Kafka
Get user_status
response user_status
Sync for offline
Event Processing Logic
Publish User Status Event
Event Queue
Subscribe to user Status Event
Event Subscribe
Get Active data
Check Active user
Response active user
Cosmos DB
Sync User Activity
Store Active data
Store response user status to keep Alive User
Redis


User Management System
1. API: Sign Up
- POST /api/signup
- Validation Logic
- Password Hashing
- JWT Generation
- Cosmos DB
2. API: Login
- POST /api/login
- Authentication Logic
- JWT Generation
- Cosmos DB
3. API: Get User Status
- GET /api/user-status
- JWT Authentication
- Cosmos DB
4. API: Update User Activity
- POST /api/user-activity
- Validation Logic
- Cosmos DB
5. Event Publisher
- Kafka/RabbitMQ
- Publishing Logic
Cosmos DB
- Store User Data
- Store Activity Data
Redis
- Store Active Users
- Store User Activity
- Retrieve User Activity
Event System (Kafka, RabbitMQ)
1. Publish User Status Event
- Event Queue


Game Server
1. Active User Fetcher
- API Call to User Management System
- Store in Redis
2. API: Track User Activity
- API Endpoint
- JWT Authentication
- Store Activity in Redis
3. API: Sync User Activity
- Read from Redis
- API Call to User Management System
- JWT Authentication
User Management System
2. API: Update User Activity
- API Endpoint
- Update Logic
- JWT Authentication
Cosmos DB
- Store Activity Data
Event Subscriber
2. Subscribe to User Status Events
- Event Processing Logic
User Management System
3. Event Publisher
- Event System (Kafka, RabbitMQ)
- Publishing Logic

- **User Management System**

  - **API: Get Active Users**
    - **Interaction**: Gets the list of active users.
    - **Components**: Uses Cosmos DB, API Endpoint, and JWT for security.
  - **API: Update User Activity**
    - **Interaction**: Updates user activity data.
    - **Components**: Uses API Endpoint, logic for updating, JWT for security, and stores data in Cosmos DB.
  - **Event Publisher**
    - **Interaction**: Sends out events when user status changes.
    - **Components**: Uses an Event System like Kafka or RabbitMQ to handle events.

  Redis

  - **Store Active Users**
    - **Interaction**: Stores the list of active users from the Game Server.
  - **Store User Activity**
    - **Interaction**: Stores user activity data from the Game Server.
  - **Retrieve User Activity**
    - **Interaction**: Retrieves user activity data for the Game Server.

  Game Server

  - **Active User Fetcher**
    - **Interaction**: Calls the User Management System to get the list of active users and stores it in Redis.
    - **Components**: Makes API calls and stores data in Redis.
  - **API: Track User Activity**
    - **Interaction**: Collects user activity data and stores it in Redis.
    - **Components**: Uses API Endpoint, JWT for security, and stores data in Redis.
  - **API: Sync User Activity**
    - **Interaction**: Reads user activity data from Redis and updates it in the User Management System.
    - **Components**: Reads data from Redis, makes API calls to the User Management System, and uses JWT for security.

  Event System

  - **Publish User Status Event**
    - **Interaction**: Sends out events when user status changes.
    - **Components**: Uses an Event Queue to manage events.
  - **Subscribe to User Status Events**
    - **Interaction**: Receives and processes user status change events.
    - **Components**: Logic for processing events.

User Management System Detailed Breakdown

The User Management System (UMS) includes several essential features to manage users effectively. Here are the key components and their interactions:

Key Features

1. **Sign Up**
    - **Purpose**: To register a new user.
    - **Components**:
        - **API Endpoint**: POST /api/signup
        - **Validation Logic**: Validates the input user data.
        - **Password Hashing**: Hashes the password before storing it.
        - **JWT Generation**: Generates a JSON Web Token (JWT) upon successful sign-up.
        - **Cosmos DB**: Stores user data.
2. **Login**
    - **Purpose**: To authenticate a user.
    - **Components**:
        - **API Endpoint**: POST /api/login
        - **Authentication Logic**: Verifies user credentials.
        - **JWT Generation**: Generates a JWT upon successful login.
        - **Cosmos DB**: Retrieves user data for authentication.
3. **Get User Status**
    - **Purpose**: To retrieve the current status of a user.
    - **Components**:
        - **API Endpoint**: GET /api/user-status
        - **JWT Authentication**: Ensures the request is authenticated.
        - **Cosmos DB**: Retrieves user status data.
4. **Update User Activity**
    - **Purpose**: To update the activity data of a user.
    - **Components**:
        - **API Endpoint**: POST /api/user-activity
        - **Validation Logic**: Validates the input activity data.
        - **Cosmos DB**: Stores the updated activity data.
5. **Publish User Status Change**
    - **Purpose**: To publish events when user status changes.
    - **Components**:
        - **Event System (Kafka, RabbitMQ)**: Manages event publication.
        - **Publishing Logic**: Determines when and what events to publish.

Detailed Workflow

1. **Sign Up Workflow**
    - **Step 1**: The client sends a request to the POST /api/signup endpoint.
    - **Step 2**: The server validates the input data and hashes the password.
    - **Step 3**: User data is stored in Cosmos DB.
    - **Step 4**: A JWT is generated and returned to the client.
2. **Login Workflow**

    - **Step 1**: The client sends a request to the POST /api/login endpoint.
    - **Step 2**: The server verifies the user credentials.
    - **Step 3**: A JWT is generated and returned to the client.
3. **Get User Status Workflow**
    - **Step 1**: The client sends a request to the GET /api/user-status endpoint.
    - **Step 2**: The server verifies the JWT and retrieves user status from Cosmos DB.
    - **Step 3**: The user status is returned to the client.
4. **Update User Activity Workflow**
    - **Step 1**: The Game Server sends user activity data to the POST /api/user-activity endpoint.
    - **Step 2**: The server validates the activity data.
    - **Step 3**: Activity data is stored in Cosmos DB.
5. **Publish User Status Change Workflow**
    - **Step 1**: When a user's status changes, the server triggers the event publishing logic.
    - **Step 2**: The event is created and sent to Kafka or RabbitMQ.
    - **Step 3**: The Event System manages the event and delivers it to all subscribed systems.

Event System Detailed Breakdown with Required Events

The Event System will manage various events that are crucial for the seamless operation and synchronization of the User Management System (UMS) and Game Server. Below are the key events and their details:

Key Events

1. **User SignUp Event**
    - **Purpose**: To notify systems when a new user has signed up.
    - **Components**:
        - **Event Type**: userSignUp
        - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z", "userDetails": { "username": "john_doe", "email": "john@example.com" } }
2. **User Login Event**
    - **Purpose**: To notify systems when a user logs in.
    - **Components**:
        - **Event Type**: userLogin
        - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z" }
3. **User Status Change Event**
    - **Purpose**: To notify systems when a user's status changes (e.g., from inactive to active).
    - **Components**:
        - **Event Type**: userStatusChange
        - **Event Data**: { "userId": "123", "newStatus": "active", "timestamp": "2024-06-28T12:05:00Z" }
4. **User Activity Event**
    - **Purpose**: To notify systems of user activities (e.g., gameplay actions).
    - **Components**:
        - **Event Type**: userActivity
        - **Event Data**: { "userId": "123", "activity": "completedLevel", "details": { "level": 5 }, "timestamp": "2024-06-28T12:10:00Z" }
5. **User Logout Event**
    - **Purpose**: To notify systems when a user logs out.
    - **Components**:
        - **Event Type**: userLogout
        - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:15:00Z" }

Detailed Workflow with Events (TBD)

1. **Event Publishing Workflow**
    - **Step 1**: An event is triggered by a specific condition in the UMS (e.g., user status change).
    - **Step 2**: The Event Creation Logic creates an event message with relevant data.

    - **Step 3**: The Publishing Logic sends the event message to the Message Broker.
    - **Step 4**: The Message Broker stores the event message in the appropriate topic or queue.
2. **Event Subscription Workflow**
    - **Step 1**: The Event Subscriber subscribes to a specific topic or queue to receive events.
    - **Step 2**: When a new event is published, the Message Broker delivers the event message to all subscribers.
    - **Step 3**: The Event Processing Logic in the subscriber handles the event and performs necessary actions (e.g., updating user activity in the Game Server).

Interaction Details with Events

1. **User SignUp Event**
    - **Event Type**: userSignUp
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z", "userDetails": { "username": "john_doe", "email": "john@example.com" } }
    - **Workflow**: When a user signs up, this event is published to notify other systems.
2. **User Login Event**
    - **Event Type**: userLogin
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z" }
    - **Workflow**: When a user logs in, this event is published to notify other systems.
3. **User Status Change Event**
    - **Event Type**: userStatusChange
    - **Event Data**: { "userId": "123", "newStatus": "active", "timestamp": "2024-06-28T12:05:00Z" }
    - **Workflow**: When a user's status changes, this event is published to notify other systems.
4. **User Activity Event**
    - **Event Type**: userActivity
    - **Event Data**: { "userId": "123", "activity": "completedLevel", "details": { "level": 5 }, "timestamp": "2024-06-28T12:10:00Z" }
    - **Workflow**: When a user performs an activity, this event is published to notify other systems.
5. **User Logout Event**
    - **Event Type**: userLogout
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:15:00Z" }
    - **Workflow**: When a user logs out, this event is published to notify other systems.

Game Server Detailed Breakdown with Required Events

The Game Server is responsible for handling client requests, interacting with the User Management System (UMS), managing real-time data in Redis, and processing gameplay-related tasks using OpenAI's models.

Key Features and Workflows

1. **Receive and Process Client Packets**
   - **Purpose**: To handle incoming packets from clients, which may include user actions, game commands, or requests for AI processing.
   - **Components**:
     - **Packet Handler**: Receives and parses client packets.
     - **Request Router**: Routes the request to the appropriate handler based on the type of request.
2. **Fetch Active Users**
   - **Purpose**: To retrieve the list of active users from UMS and store it in Redis for quick access.
   - **Components**:
     - **API Call to UMS**: GET /api/active-users
     - **Store in Redis**: Saves active users in Redis for efficient access.
3. **Track User Activity**
   - **Purpose**: To track and store user activities in Redis for real-time data access and periodic syncing with UMS.
   - **Components**:
     - **API Endpoint**: POST /api/track-activity
     - **Store Activity in Redis**: Logs user activities in Redis.
4. **Sync User Activity with UMS**
   - **Purpose**: To periodically sync user activity data from Redis to UMS to ensure data consistency.
   - **Components**:
     - **Read from Redis**: Retrieves user activity data from Redis.
     - **API Call to UMS**: POST /api/user-activity
     - **JWT Authentication**: Secures API calls to UMS.
5. **Call OpenAI API**
   - **Purpose**: To handle gameplay-related tasks that require processing by OpenAI's models (e.g., GPT-3).
   - **Components**:
     - **Retrieve API Key**: Securely retrieves the OpenAI API key.
     - **Make API Call**: Sends a request to OpenAI’s API.
     - **Process Response**: Handles the response from OpenAI and integrates it into the game logic.

Interaction with User Management System (UMS)

1. **User Authentication and Initial Status Retrieval**
   - **Scenario**: When a user first logs in or connects to the game, the game server verifies the user's authentication and retrieves the initial status from UMS.
   - **Interaction**:
     - **API Call**: GET /api/user-status
     - **Reason**: To ensure the user is authenticated and to retrieve the current status for initial setup.
2. **User Profile and Activity Updates**
   - **Scenario**: When user activities occur or user profile updates are needed, the game server updates the UMS with the new information.
   - **Interaction**:

        - **API Calls**:
            - POST /api/user-activity: For activity updates.
            - POST /api/update-profile: For profile updates.
        - **Reason**: To keep user data consistent and updated across the system.

Event Handling and Messaging

1. **Event Publishing**
    - **Purpose**: To publish events when significant actions occur (e.g., user status change, new activity).
    - **Components**:
        - **Event System (Kafka, RabbitMQ)**: Manages event queues.
        - **Publishing Logic**: Sends events to the message broker.
2. **Event Subscription**
    - **Purpose**: To subscribe to relevant events and update the game state or user data accordingly.
    - **Components**:
        - **Subscription Logic**: Subscribes to specific event topics.
        - **Event Processing Logic**: Processes the events and performs necessary actions.

Required Events

1. **User SignUp Event**
    - **Purpose**: Notifies systems when a new user signs up.
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z", "userDetails": { "username": "john_doe", "email": "john@example.com" } }
2. **User Login Event**
    - **Purpose**: Notifies systems when a user logs in.
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:00:00Z" }
3. **User Status Change Event**
    - **Purpose**: Notifies systems when a user's status changes (e.g., from inactive to active).
    - **Event Data**: { "userId": "123", "newStatus": "active", "timestamp": "2024-06-28T12:05:00Z" }
4. **User Activity Event**
    - **Purpose**: Notifies systems of user activities (e.g., gameplay actions).
    - **Event Data**: { "userId": "123", "activity": "completedLevel", "details": { "level": 5 }, "timestamp": "2024-06-28T12:10:00Z" }
5. **User Logout Event**
    - **Purpose**: Notifies systems when a user logs out.
    - **Event Data**: { "userId": "123", "timestamp": "2024-06-28T12:15:00Z" }