CS 5200 – Phase 1 Project Proposal

1. Project Information

- **Project Title**: ChatBot Based Virtual Character Chat System
- **Group Name**: *Group #8*
- **Team Members**: Yulong Cao, Harshal Jorwekar, Chengxu Lan, Rohan Verma

---

2. Problem Statement

2.1 Project Overview

This project aims to develop a chatbot system integrated with Discord, allowing users to interact with virtual characters (VCs) that are user-generated and managed. By sending commands to a Discord bot, users can:

- **Chat** with an existing virtual character.
- **Create, customize, and publish** new characters.
- **Manage** conversation and character data, including resetting affinity and memory.

OpenAI's GPT API will be used for generating character responses. However, the **memory summary** is stored and managed separately in the database (not appended to the GPT conversation context) to retain long-term memory without inflating token usage.

2.1.1 How the Discord Bot Works

- **Bot Commands & Events**: The bot listens for specific commands (e.g., `.create`, `.chat`, `.reset`) and responds accordingly. Users can invoke these commands in public or private channels on a Discord server.
- **Backend Integration**: Upon receiving a command, the bot forwards the request to our backend service (likely in Python). The backend handles business logic—such as creating new characters, retrieving memory data, updating affinity scores, or calling GPT for a response.
- **Response Handling**: The bot then sends the result of that backend processing back to Discord, either as a text message or an embed.
- **Security & Authentication**: We leverage Discord's built-in authentication to identify which user is issuing commands. The backend uses that information to apply the correct permissions and retrieve the relevant data from the database.

2.1.2. System Architecture

- **Architectural Overview**

  We propose a **Discord Bot** as the main interface, interacting with a backend service hosted on a cloud VM or an application service. The **backend** connects to a **MySQL database on GCP** for persistent data storage. Our tech stack:

  Frontend (Discord): Users interact via text commands and messages.

  Backend: Possibly built using **Python** (Django or Flask) to handle business logic, and will utilize **Nonebot** project with community adapters to interact with discord API.

  GPT Integration: We will call **OpenAI's GPT API** for character responses.

  Database: **MySQL** on **GCP**.

- **System Flow**

  1. **User sends a command** in Discord (create character, chat, reset memory, etc.).
  2. **Discord Bot** receives the command and forwards it to the backend.
  3. **Backend** processes the request using relevant modules (User Manager, Character Manager, Points Manager) and updates/queries the MySQL database.
  4. **If needed**, the backend calls the OpenAI GPT API for a response.

5. **Response** is returned to the user through Discord.

2.2 Problem Definition

- **Motivation**: Traditional chatbots often have static conversation models and limited personalization. We aim to create a dynamic chatbot environment where each character's behavior and affinity evolve through user interactions.
- **Challenges**: Managing concurrency, ensuring database integrity, leveraging GPT effectively, and scaling to handle multiple characters interacting with multiple users.

2.3 Target Users

1. **General Users (Players)**: Anyone on Discord who wants to interact with or create virtual characters. They can:
   - Initiate chats with an existing character.
   - Create new characters with distinct personalities.
   - Track points usage or earn points from other users who engage with their characters.
2. **Admins**: Overseers of the system with full CRUD access to all data. They ensure the system remains moderated, stable, and well-managed.

2.4 Project Scope

- **Feature Highlights**:
  - User creation and customization of virtual characters.
  - Conversation tracking with memory summary and affinity scoring, with long-term memory stored separately from GPT’s request context.
  - Ability to reset memory or affinity on demand.
  - Implementation of a points system as a sort of currency.
  - Potential for multi-character scenario creation to allow group interactions

    (post-MVP feature)

2.5 Real-world Relevance

- **Entertainment and Social Interaction**: Fosters creativity and engagement.
- **Data Analysis**: Potentially explore user behavior, conversation patterns, or AI responses.
- **Micro-economy Insight**: The points system offers a limited simulation of in-app currency dynamics, though we currently do not consider transaction fees or daily limits.

---

3. Query Formulation

We anticipate at least **15** core queries to support the primary use cases:

1. **List all virtual characters created by a specific user**
2. **Retrieve a conversation history between a user and a character**
3. **List all virtual characters a user has interacted with**
4. **Display all characters with affinity above a certain threshold for a given user**
5. **Check if a character is currently set to private or public**
6. **Show the top 5 characters by total chat interactions**
7. **Search for characters by keyword in their descriptions**
8. **List users with points < 100**
9. **List all users who have not interacted with any characters in X days**
10. **List all active users in past 24 hours**
11. **Calculate total points transferred to a character’s creator**
12. **Retrieve all conversation timestamps for a user and character**

13. **Get total points a user has spent on a virtual character**
14. <Admin Only> **List all users that are active within last 10 minutes**
15. <Admin Only> **List all users that are have created more than 10 characters**

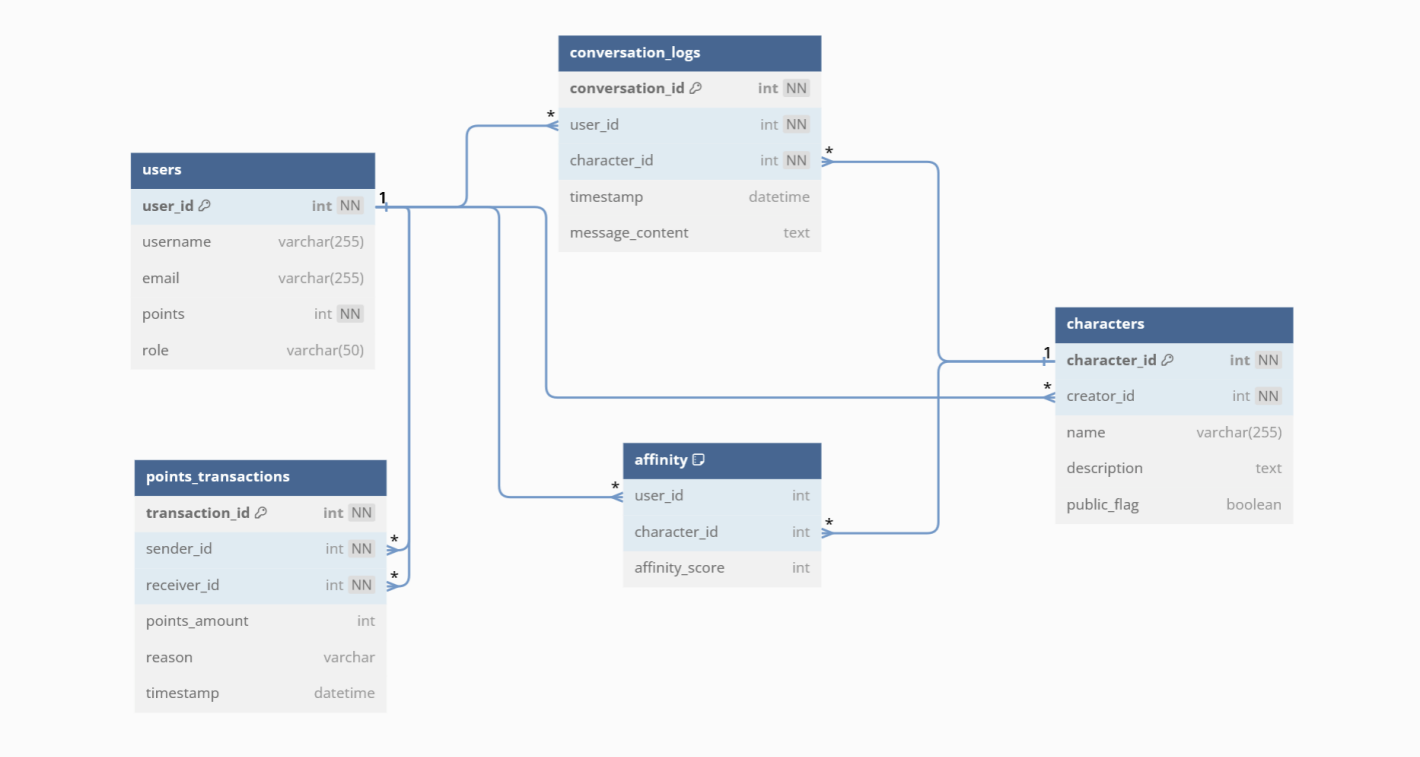
---

4. Database Assumptions

- **User Registration (Discord Authentication)**
  - All participants must have an authenticated Discord account to interact with the chatbot.
  - A new user record is created in our database the first time a Discord user interacts with the system (e.g., upon issuing a command).
- **Unique Identifiers**
  - Each record in our database (e.g., users, characters, transactions) is assigned a unique primary key (e.g., `user_id`, `character_id`, `transaction_id`) to **prevent duplicates** and ensure each entity is uniquely identifiable.
  - These identifiers may be auto-generated (e.g., auto-increment fields).
- **Transaction Handling**
  - Critical operations such as point transfers and new character creation will be **transactional** in the database. If any part of the operation fails, the entire transaction is rolled back to maintain data integrity.
  - This ensures, for example, that a user's points are not deducted without simultaneously recording the transaction in the `points_transactions` table.
- **Memory Summaries and Chat Logs**
  - Memory summaries (long-term user-character notes) are **stored separately** from the raw conversation logs to **avoid inflating GPT context** and to maintain performance.
  - Each conversation entry links to a valid user and character. A memory summary cannot exist for a non-existent user-character pair.
- **Referential Integrity & Foreign Keys**
  - All relationships between entities (e.g., `creator_id` in `characters` referencing `users.user_id`) must be **valid**; a record cannot exist if its parent record does not exist.
  - The system enforces **foreign key constraints** in MySQL to maintain consistent relationships and prevent orphan records (e.g., a conversation log referencing a non-existent user or character).
- **Points System Constraints**
  - Users cannot have negative points; all point allocations must pass a check ensuring no user's point balance falls below zero.
  - There are **no transaction fees or daily limits**; however, transactions failing the negative-balance check are rejected.
- **Data Consistency**
  - The system will ensure that any updates to character data, user data, or memory summaries will reflect accurately across related tables.
  - For example, if a character is deleted, all corresponding affinity entries, conversation logs, and points references tied to that character are handled according to cascading rules (either deleted or set to null, as appropriate).
- **Security and Roles**
  - **Admin** users have full access, including the ability to modify or delete any record.
  - **Regular** users can only modify or delete resources they own (e.g., their own characters).
  - Discord-based role checks or additional in-app logic ensures **only Admins** can perform critical system changes.
- **No Built-in Moderation**
  - Offensive or inappropriate content from users is not automatically filtered within the scope of this project. Any moderation or banning of users/characters must be handled manually by an **Admin**.

---

6. Database Schema



7. Conclusion

In this phase, we have:

1. Defined the **problem statement** and relevance of a Discord-based virtual character chatbot.
2. Outlined a **system architecture** leveraging **MySQL on GCP** and **OpenAI GPT**.
3. Proposed **15 queries** that represent core functionalities.
4. Identified key **assumptions** on entity relationships and constraints.
5. Integrated a **DBML diagram** to visually represent our database schema.
6. Noted that advanced features like multi-character scenarios are **post-MVP**.
7. Concluded that explicit privacy or moderation features are out of scope for this course project.

In subsequent phases, we plan to:

- Implement a **prototype** of the Discord bot.
- Create the **database schema** and ensure it supports all required queries.
- Integrate advanced features such as scenario-based multi-character interactions.
- Optimize and refine the points system based on user feedback