Write a brief (<500 words) description of your approach, including answers to the following questions:

1) What are the most important goals of your autocomplete design and how are those goals achieved?
2) What technologies would your system leverage

The backend system follows the SOLID principle to ensure modularity, testability, flexibility, and scalability, especially for the autocomplete feature. In terms of efficiency and stability, it leverages Django libraries such as DjangoORM and the cache system.

As the UML diagram provides visual structure, the system consists of custom classes: CutomEntiry, PlayerSummaryResponse, CustomGame, and CustomShot, each with a single responsibility.

- CustomEntiry is an abstract class defining common methods add_sub_entity and to_dict.
- PlayerSummaryResponse manages player summary data, including the player's name and a list of CustomGame objects.
- CustomGame represents individual player statistics for a specific game and contains a list of CustomShot objects.
- CustomShot holds information about individual shots in a game.

PlayerSummaryResponse and CustomGame classes inherit from CustomEntity. This abstraction allows the system to be easily extended in the future without modifying existing code to align with the Open/Closed Principle in the SOLID. For example, adding a Team class that aggregates PlayerSummaryResponse objects can be done without impacting the current structure. The Dependency Inversion Principle is achieved by having PlayerSummaryResponce and CutomGame subclass depend on the CustomEntity abstraction rather than concrete implementations.

To achieve high decoupling and modularity, the system's architecture includes distinct layers—Controller, Handler, and DataLoader as the system architecture diagram projects. It separates the domain of tasks and defines clear responsibilities:

- The Controller handles API requests related to player suggestions and player summary retrieval.
- The Handler manages the logic for creating player summaries and converting data into PlayerSummaryResponse objects.
- The DataLoader directly interacts with the database, ensuring the reliable retrieval of player statistics, game data, and shot information as fundamental data representation.

This multi-layer architecture ensures decoupling and promotes modularity, making the system easy to extend and maintain with clear function boundaries for enhanced testing layers, system scalability, and stability.

One of the critical features of the system is the autocomplete suggestion functionality. The system shows potential players' names in the box and helps the user's player name search process as the frontend_example_1.jpg shows. When the system starts, it stores player names and corresponding IDs to the Trie(prefix tree) data structure for fast lookups. When a user types part of a name, the system efficiently retrieves potential matches from the Trie, reducing database access and improving performance. The AutocompleteController manages the Trie and API interaction to provide fast, accurate suggestions.

The system leverages Django ORM for seamless database interaction, mapping models such as Player, Game, PlayerStats, and Shot to the database. The model is a projection of the database tables visualized in the database schema. Moreover, Django's caching system stores frequently accessed player data, improving performance by minimizing redundant queries. It will provide better performance and stability than manually caching with a dictionary of plyaer_id and PlayerSummary instances key-value pair.

Although the current system focuses on player search and autocomplete, it is designed to be scalable and flexible for future growth with the SOLID principle. Its modular, multi-layered architecture ensures that components remain decoupled and easily testable, making the system adaptable to evolving requirements and long-term use.