My approach is to create an autocomplete service that receives queries through an API gateway and utilizes a load balancer and a cache for efficiency. The user input will be sent to the an instance of an API gateway through the load balancer. The gateway then forwards the query to my autocomplete service, along with metadata indicating if the query is complete or not. If the query is complete, the autocomplete service will cache that query. Otherwise, if the query is incomplete, the service first checks the cache for matching suggestions. If suggestions are found, the autocomplete service returns them immediately. If there are no matching suggestions found, the service queries the database for data matching the partial input, and returns that as suggestions. The autocomplete returns these suggestions to the API gateway, which forward them to the user.

The key goals for my approach for the autocomplete system are speed and efficiency. My approach must provide suggestions instantly as users type and delete letters. This speed will be achieved through a caching layer using Redis, which is a fast, in-memory data store that's commonly used for caching. Redis ensures low-latency access to frequently searched inputs and reduces the load on the backend database. Redis also supports eviction policies to manage memory usage. In my approach, after the user hits enter for a query, I send that query to the autocomplete service, too, even though it doesn't need autocomplete anymore. I do this so that the autocomplete service can cache this complete query in order to reduce the need for database lookups, because most queries in databases have temporal or spatial adjacency. This means that most queries are either repeated close together in time, or they are related and near each other in the database. This is relevant in autocomplete because the more times a query is searched, the more likely autocomplete will suggest that query, which is what my approach will do, too. And by caching recent/common queries, the system will reduce database lookups, which will improve the response time significantly. Response time is extremely important, because user's type very fast, so the suggestions have to match that speed. This means downtime is critical and periods of unavailability due to a failure can't happen. If the infrastructure can comfortably handle 100 or less users simultaneously, then a load balancer is not necessary, but to guarantee quick suggestions, horizontal scaling and a load balancer may be necessary to ensure that there is no delay in the autocomplete. I would utilize Nginx, as it can handle a variety of load balancing methods to distribute query traffic amongst multiple instances of the API gateway and autocomplete service.