select_related:

When utilizing select_related, Django enhances the retrieval process of related objects from a database. Rather than executing distinct queries for each related object (such as the author and post of a comment), it employs a JOIN within the database. This technique allows for the acquisition of all necessary information in a single database transaction.

annotate and Count:

Instead of performing value calculations within Python code, it was determined that these calculations should be executed directly in the database. This optimization prevents Python from expending resources on such computations.

subqueries:

Within views, instead of employing prefetch_related, the logic for retrieving the latest three comments has been incorporated into the serializer's to_representation method. This approach utilizes a subquery to fetch only the required comments at the moment of preparing data for transmission to the user.

Data by Endpoint:

Certain endpoints (specifically those with a detail endpoint) have been structured as follows:

For "get-object-list" endpoints (which in a real-world scenario are anticipated to contain substantial amounts of data, even with paging implemented), only a limited number of fields are returned. For instance, rather than returning identical objects in get_users and get_user_by_id, get_users provides the follower count but not the specific details. To obtain the follower details, it is necessary to access user_detail, thereby optimizing the time required to acquire basic data.

What else could be better?

We could avoid repeating Data structures (detail vs general) implementing different data return, filtering by context (user-list vs user-detail).