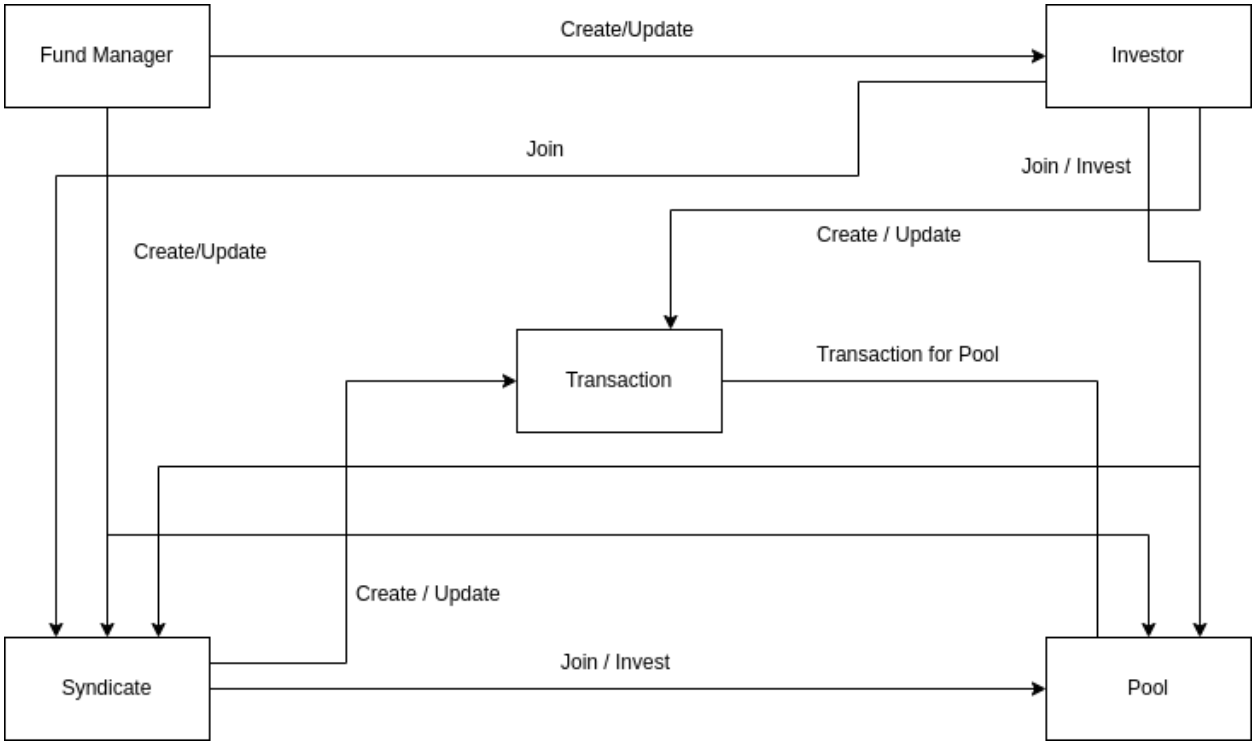
Task 3

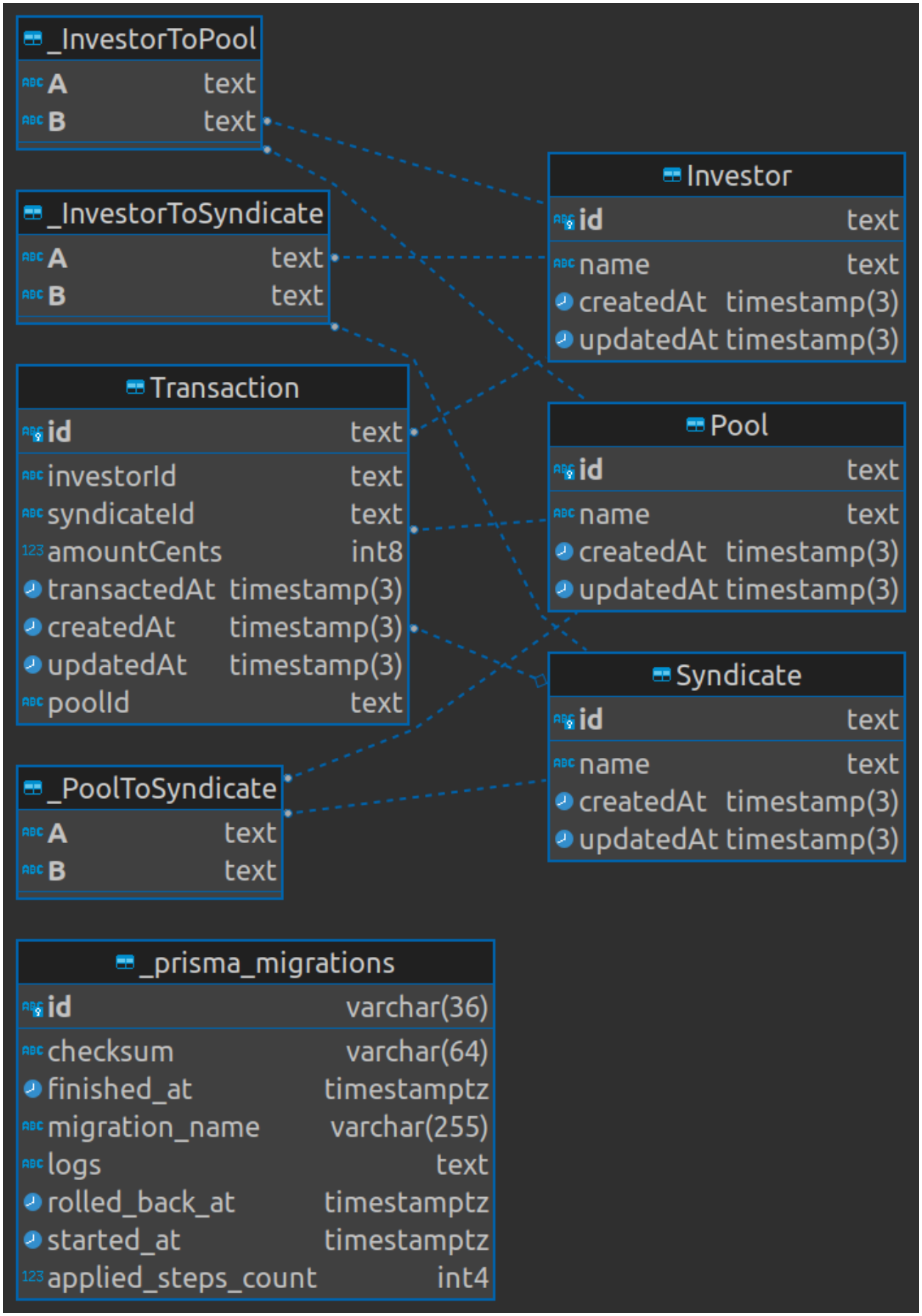
Let’s look at an overview diagram of the key components first.



Assumptions:

1. We are not considering the Funds/Entities that will receive the investments in the end within the context of this scope. We will need to take those entities into account in the future eventually
2. Fund Manager will oversee all the interactions happening in the system

Now let’s look at the database schema to accommodate all the interactions mentioned in the diagram above.


_InvestorToPool
A text
B text
_InvestorToSyndicate
A text
B text
Transaction
id text
investorId text
syndicateId text
amountCents int8
transactedAt timestamp(3)
createdAt timestamp(3)
updatedAt timestamp(3)
poolId text
_PoolToSyndicate
A text
B text
_prisma_migrations
id varchar(36)
checksum varchar(64)
finished_at timestamptz
migration_name varchar(255)
logs text
rolled_back_at timestamptz
started_at timestamptz
applied_steps_count int4
Investor
id text
name text
createdAt timestamp(3)
updatedAt timestamp(3)
Pool
id text
name text
createdAt timestamp(3)
updatedAt timestamp(3)
Syndicate
id text
name text
createdAt timestamp(3)
updatedAt timestamp(3)

As you can see I have only extended the ER Diagram from Task 1 to accommodate the interactions mentioned in Task 3. A new entity, Pool is introduced which has many-to-many relations with Investor and Syndicate and a one-to-many relation with Transaction. This is under the following assumptions

- Each investor may invest in one or many pools
- Each pool may have one or many investors
- Each syndicate may invest in one or many pools
- Each pool may have one or many syndicates
- Each pool can have multiple transactions
- Each transaction may only be made to one pool

For the system I would consider the following tech stack

- PostgreSQL as the database
- NestJS in the backend
- NextJS in the frontend

I would consider a relational database for the system. As you can see, all of the components are interconnected with each other and a relational database serves better in these cases. PostgreSQL is the industry standard nowadays for relational databases with high scalability, performance and maintenance.

NestJS is highly scalable and performant for large systems with a good structural codebase. The development becomes faster and easier with all the commonly used features that come with it. Moreover, it is really popular in the developers community and easier to build a good team with this.

Let me be a little biased about NextJS. I believe it is the most cutting-edge platform for the frontend with all the good stuff in the frontend world. Scaling a large system has never been easier than this. NextJS is regularly maintained, highly performant and developer-friendly. Also, having a common programming language in both the backend and frontend adds the advantage of reduction of learning curve time.

A potential bottleneck could be the huge number of transactions made to the system. In that case, we need to optimize the database with proper indexes and partitioning if needed. We could also leverage the use of caching with Redis to serve the users with a faster system.

To develop the system, I would first consider a task management platform like Jira, ClickUp or Plane to collaborate with the team. The whole system should be splitted into multiple stories depending on the user journey. With proper planning, we need to divide the stories into sprints.

Each sprint will have an atomic task related to a feature, bug or improvement. The team will be assigned the tasks based on the priority of the deliverables. The system will be deployed in sprints. I would also consider 3 environments as Development, Staging and Production. A task will first be in the Development environment. After proper testing, it will be promoted to Staging. If no unexpected issues occur it will finally be deployed to production. The whole deployment cycle could also be automated with CI/CD and Docker containerization.