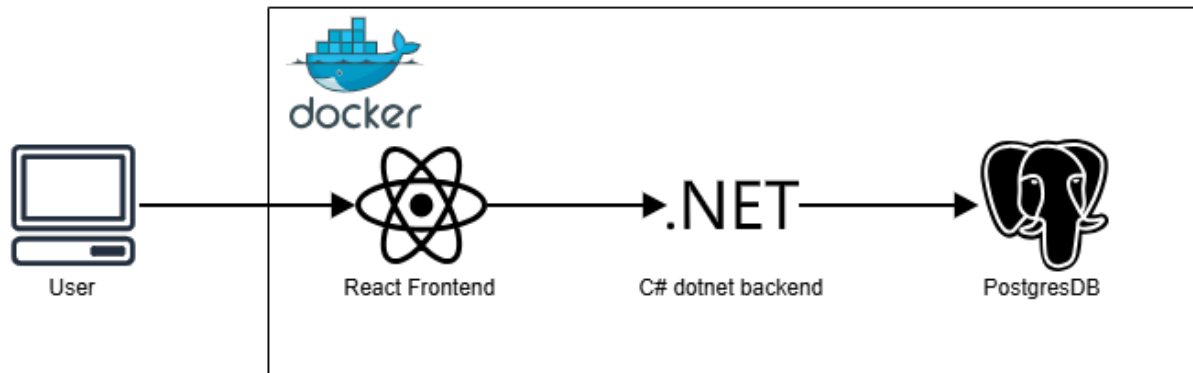
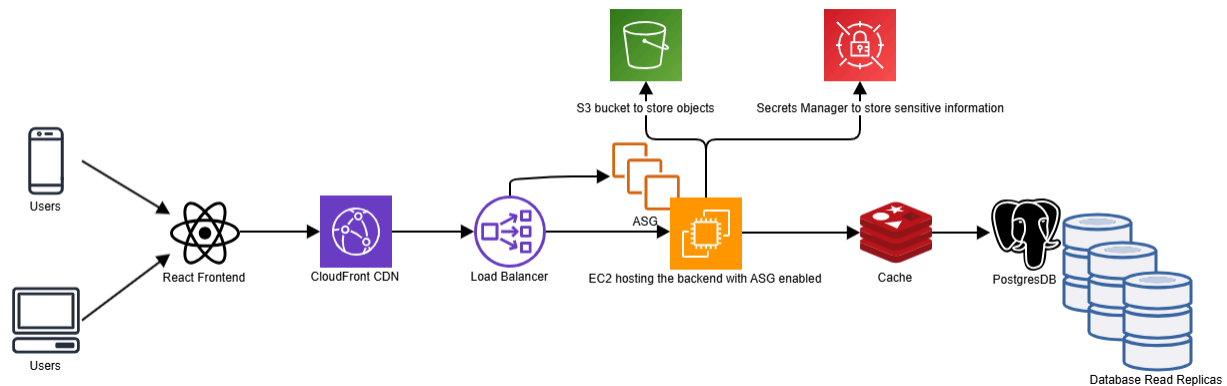


High-Level Architecture Diagram



Scalability Strategy



- The Application can be hosted on cloud (Either as server or serverless) to easily **scale based on triggers** (Usage demand, heavy usage hours, etc). A **load balancer** can also be used to limit/route traffic across multiple spawns. Cost wise, this approach will almost always be cheaper than having on-prem servers.
- Since this app handles images and videos, it's ideal to use **object storage** such as S3 to handle such tasks. It's also easy to integrate **CDN** to this for lower latency transfer and content caching.
- The Database should have a standalone **caching system** (Redis, Memcache, etc) to allow more configuration and options that does not require code change (I.e. Lazy-loading, Write-Through)
- As most users are going to use the app to view checklists, the reads will be more than the write. In that case, spawning **database replicas** should suffice. If we find that users are heavily using the writing operation, **database sharding** can also be

introduced but it adds extra complexity and data routing management. Having replicas also **improves Availability and Disaster recovery** especially if deployed **cross-zone, cross-region**

- Other strategies to help with speeding up development is to integrate these workflows: **IaC** (i.e. Terraform), **CI/CD** (i.e. Jenkins, Harness, etc), Remote Notifications (i.e. AWS SNS)

Component Choice

For the time frame given and to set a realistic goal, the below decision was made

- Decided to use **Docker** to orchestrate the core components frontend, backend, and database since it makes it easier for others to run and test in an isolated environment
- Entity Framework
 - Database operations are mainly handled by the backend in the app and makes it easy to get database work started. Ideally this should be offloaded as a separate responsibility entirely, having a data team to manage and create stored procedures for more complex and efficient queries.
 - EF also tracks context and leaves a **snapshot** which acts as a cache that reduces the number of reads/writes to the database. Although not as configurable and powerful as standalone cache like Redis, it is enough for a showcase within the scope of this project.
 - Context Pooling also increases the performance by taking advantage of **Singleton pattern**
- SignalR library allows multiple users to see live changes on the list, such as CRUD operations and also notifies users
- Integrated a very simple **JwtToken** flow. In this app, only username is used. Realistically, this should be handled by SSO or username with hashed password. All sensitive information such as DB connection string, username/password should be hashed/encrypted, and offloaded to a secure vault, such as AWS Secrets Manager

Data Models

- **user** - id::uuid, username::varchar
- **todo_list** - id::uuid, user_id(user->id)::uuid, title::varchar
- **todo_item** - id::uuid, list_id(todo_list->id), content::text, media_url::varchar, media_type::int

- **todo_list_share** id::uuid, list_id(todo_list->id)::uuid, user_id(user->id)::uuid, permission::int