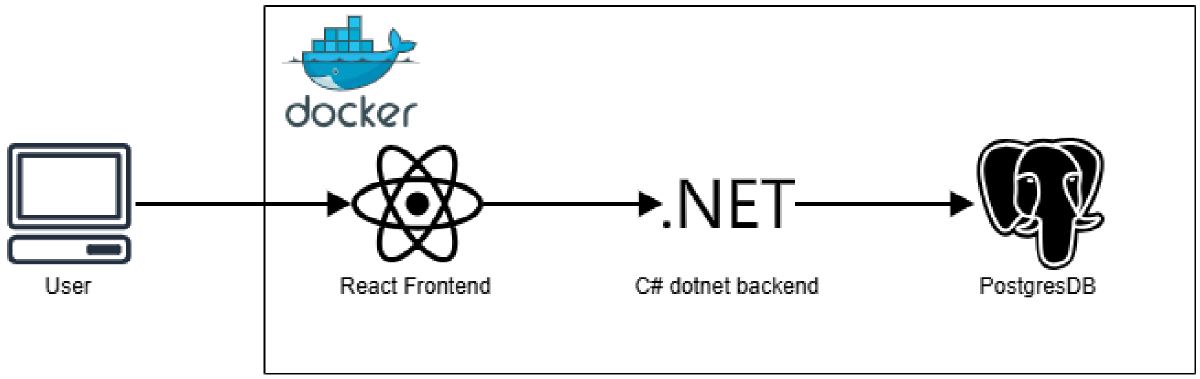
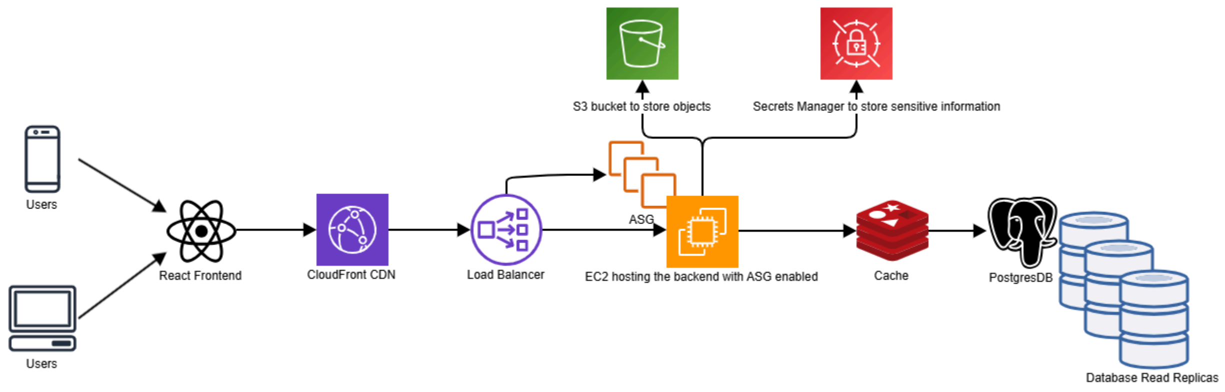
High-Level Architecture Diagram



Scalability Strategy


S3 bucket to store objects
Secrets Manager to store sensitive information
Users
Users
React Frontend
CloudFront CDN
Load Balancer
ASG
EC2 hosting the backend with ASG enabled
Cache
PostgresDB
Database Read Replicas

Note: Updated section here to elaborate more details on this approach (2025-05-27 EST)

- The application is hosted on the cloud, leveraging either **server-based infrastructure** (i.e. EC2) or **serverless compute** (i.e. AWS Fargate), enabling seamless horizontal scalability based on traffic. A **load balancer** (i.e. Application Load Balancer) distributes incoming traffic across multiple containers or instances to avoid overloading any single server.
    - To handle traffic spikes, such as 100 million daily active users collaborating in real-time, the system uses Auto Scaling Groups (ASG), typically triggered by CPU usage, memory consumption, or custom metrics (i.e. active SignalR connections). This elastic scaling ensures that during peak hours, new instances are spawned automatically and decommissioned when traffic subsides, optimizing both performance and cost.

    - Note that with this approach, it is possible for users to be routed to different instances. Users connected to different servers might not receive real-time updates. To resolve this problem, I can leverage SignalR with a Redis Backplane, specifically AWS ElastiCache for Redis, which acts as a distributed pub/sub layer that synchronizes across all sessions and servers.
    - Deploying applications in different zones/regions to prevent single point of failure.
- Since this app handles images and videos, it's ideal to use **object storage** such as S3 to decouple file storage from the application. With a large user base, it is crucial to integrate a **CDN** service like CloudFront.
    - Storing objects on S3 allows all instances to access and upload media in a centralized and durable location.
    - Usage of CDN service like CloudFront have multiple advantages:
        1. Content caching offloads traffic from the core application, reducing load
        2. Caching based on proximity closest to user reduces latency, especially for large objects like a 4-minute video
        3. Reduce huge amount of cost from GET request on S3
- The Database should have a standalone **caching system** (Redis, Memcache, etc). This caching layer enables configuration options like lazy loading, write-through, and cache invalidation, without requiring code changes.
    - Usage of cache on this layer is also to improve performance and reduce cost. Having a cache offloads the request to the database, allowing a large amount of users to read with minimal latency.
- As most users are going to use the app to view todo lists, the reads will be more than the write. In that case, **read replicas** are deployed to horizontally scale database reads. If we find that users are heavily using the write operation, there are two options: vertically scaling by increasing resources or horizontal scaling with **database sharding**. Although database sharding introduces complexity, it becomes necessary at extreme load.
    - To ensure **high availability** and **fault tolerance**, the databases are deployed in multi-AZ and multi-region configurations. This ensures that if one zone or region goes down, the system can failover automatically, preserving uptime and data integrity. This is critical for compliance and enterprise reliability.
- Other strategies to help with speeding up development is to integrate these workflows: **IaC** (i.e. Terraform, CloudFormation) to automate environment provisioning and configuration, **CI/CD** (i.e. Jenkins, Harness) to automate deployment, Notifications System (i.e. AWS SNS) for remote notification

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