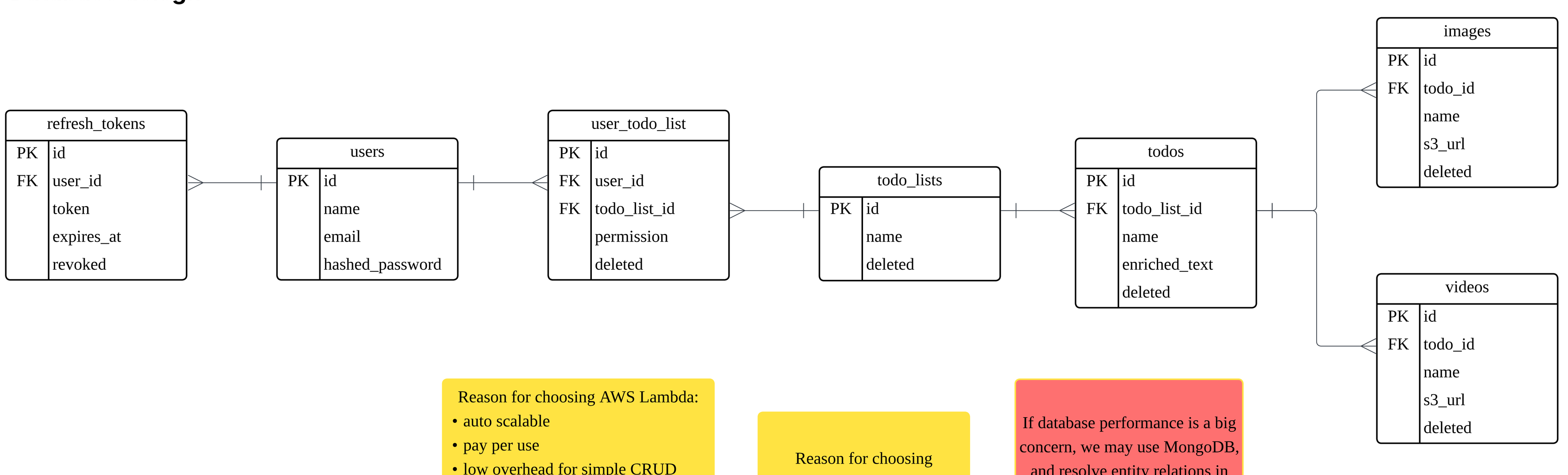
Database deisgn

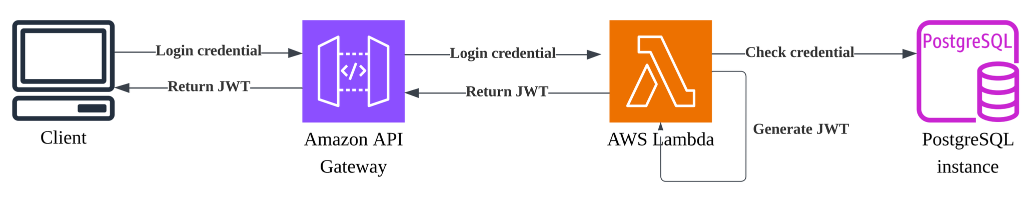


Reason for choosing AWS Lambda:
- auto scalable
- pay per use
- low overhead for simple CRUD functions comparing with EC2/ECS/EKS
- cost efficient for anticipating peak-valley fluctuation

Reason for choosing relational database:
- Strict entity relations to ensure robust authorization

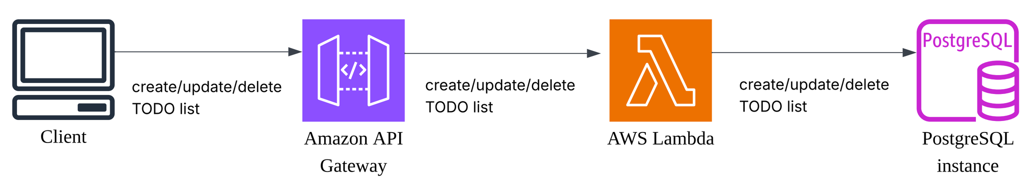
If database performance is a big concern, we may use MongoDB, and resolve entity relations in side lambda, so that database sharding is easier to be implemented

Authentication



Combining with the database entity relations, the JWT token will be used to determine whether the client 1. can access certain media files 2. can establish websocket for concurrently editing the a TODO list (or in view-only mode)

Create/Update/Delete TODO list & Invite/remove user



For creation:
- A new TODO list record will be created in the todo_lists table
- A new user todo list relation will be created in the user_todo_list table, with the permission of owner
- An owner is able to view, and edit todo lists and todos
- An owner is able to invite (remove) other users to (from) TODO list and assign role to them

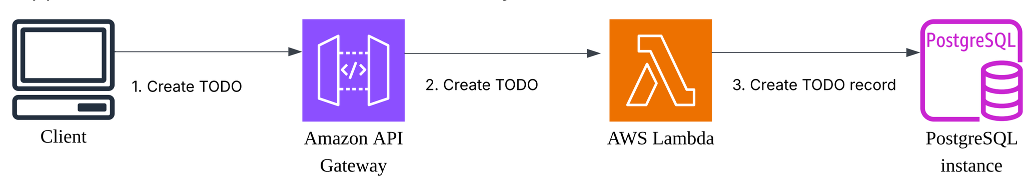
For Invitation:
- A new record will be insert to the user_todo_list with role assigned
- Edit permission can view and edit todo list and the todo items belonging to the todo list
- View-only permission can only view the todo list and the todo items

For removing user from a todo list:
- The record in the user_todo_list will be marked as deleted

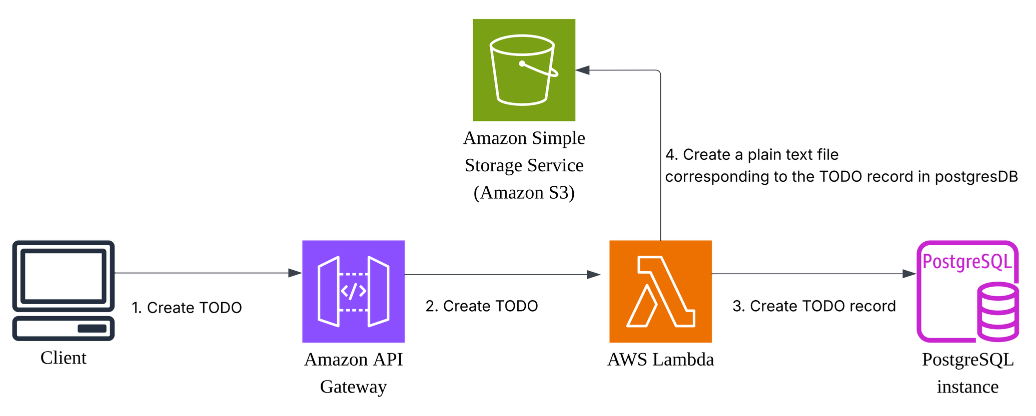
For deleting todo list:
- The record in the todo_lists table will be marked as deleted

Create TODO Item

Approach A: store the text in database directly



Approach B: store the text in S3



There are 100M daily active users, and assuming there will be on average 5 TODO list created per user, and the text size in a TODO list is on average 10 kb, then the total size of text we need to anticipate will be around 625Gb, which maybe a heavy loading for a transactional database.

(100M * 5 * 10 kb ~ 625Gb)

Also, extra cost will be incurred in AWS Lambda as its cost usage is based on api execution time and memory used per api call.

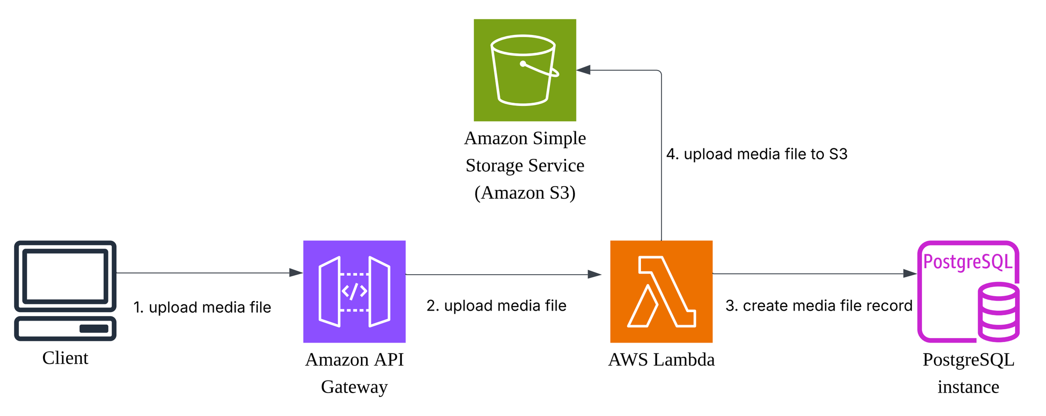
Therefore, we may want to store the text in S3 instead of postgresDB.

If we choose approach B, the enriched column in todos table will be altered to be "s3_url" corresponding to text file stored in the AWS S3.

	Approach A	Approach B
Pros	• Simple and straight forward • Easiser to maintain • No need to think about db record rollback in case of S3 file creation failed	• More cost efficient if the text size is big • segregating the job of document storage to S3
Cons	• Higher operational cost if the text size is big	• Over complicated if the text size is small

Upload media files

Approach A: upload file through server side

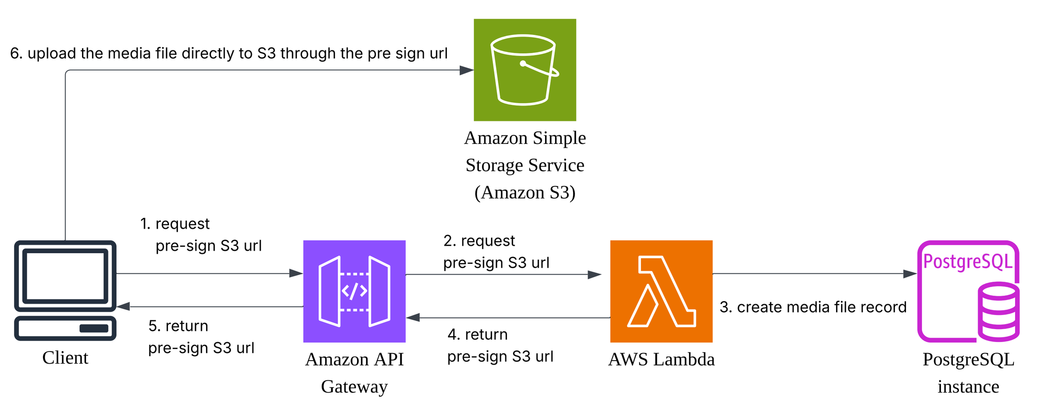


Again, we have many users, and they may upload a 4 minutes video attached to a TODO item. A 4 minutes mp4 video with 480p is around 50Mb in size.

This will create high cost and latency if we upload the media files via backend.

Uploading media fille directly in frontend is more preferrable. (Approach B)

Approach B: upload file directly in client side



	Approach A	Approach B
Pros	• No rollback action between client and server is required in case of failed upload	• lower server side cost • avoid API timeout if the file size is too big
Cons	• High operational cost • API timeout will happen if the file size is too big	• a more complicated solution

Editing TODO list concurrently

Approach A: Operational Transform with centralized servers

- Client will send an operation specifying 1. what text is inserted after which position locally; or 2. which characters between which position are deleted locally.
- The websocket server will transform the local operation positions from various client to global opertation positions by tracking what operations and at which positions are executed in all clients before.
- The websocket server will broadcast the transformed operations to client side and the client side will dynamically patch the text in local according to the transformed global operations

Example:

The initial text is "Hello World"

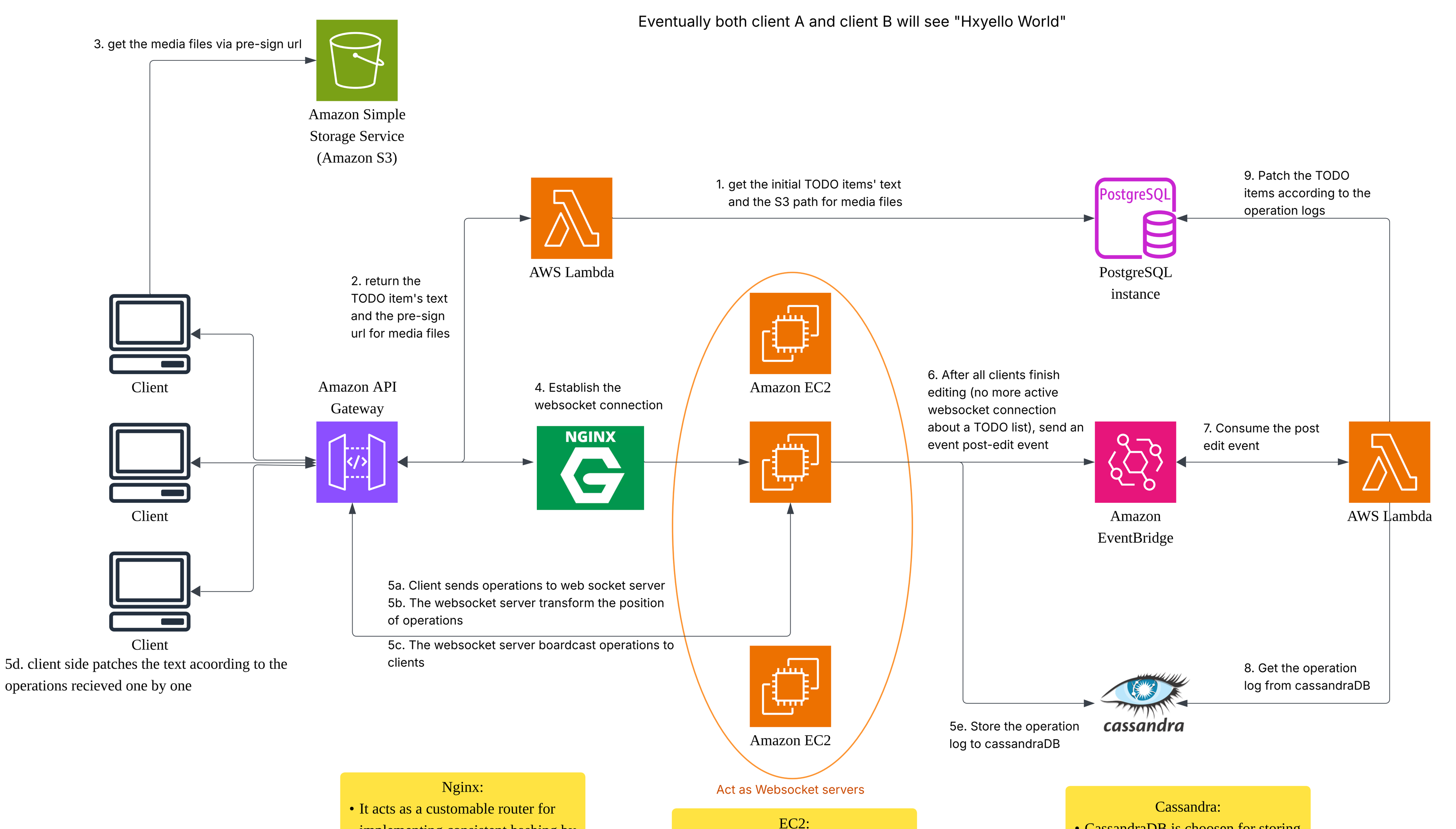
Client A insert "x" between "H" and "e", it sends {"operation": "insert", "text": "x", "position": 1} to the websocket server

Just shortly after, client B insert "y" between "H" and "e", it sends {"operation": "insert", "text": "y", "position": 1} to the websocket server

The websocket server will boardcast {"operation": "insert", "text": "x", "position": 1} to client B

The websocket server will boardcast {"operation": "insert", "text": "y", "position": 2} to client A

Eventually both client A and client B will see "Hxyello World"



Nginx:
- It acts as a customable router for implementing consistent hashing by todo_list_id
- To ensure all clients editing the same TODO list establish websocket connections to the same websocket server, we need to route the traffic to the same server according to the todo_list_id

EC2:
- EC2 is choosen for websocket server because we need to persist the websocket connection
- More EC2 instance will be added horizontally on demand

Cassandra:
- CassandraDB is choosen for storing operation logs because it is optimized for high write throughput

Approach B: Conflict-free Replicated Data Types with peer-to-peer connection

- "Conflict-free" means commutative, associative, and idempotent
- Commutative: a + b = b + a → order of merging does not matter
- Associative: (a + b) + c = a + (b + c) → batch operations can be merged arbitrarily
- Idempotent: f(f(x)) = f(x) → duplicated operations will produce the same output
- By have a data structure fulfilling these 3 properties, we can ensure consistent merging ouput among distributed nodes irrespective of merging sequence.

- "Replicated" means all data describing the local operation (insertiong and deletion) performed in a node will be replicated to all other distributed nodes.
- Each node needs to replicate all operations to ensure commutation and association.
- New text will be rendered in a client node after operations from other nodes are received.

- At the begining, each client node will receive the initial text and assign a unique id to each character to track the position (e.g.: Hello World → ["initial-1", "initial-2" ... "initial-11"])
- To fulfill idempotency, each insertion operation data has a unqiue id (e.g.: "clientA-1")
- To identify the position of insertion, each insertion operation data has "prev_id" and "next_id" referencing other node's operation
- The insertion operation data should look like: {"operation": "insert", "id": "clientA-1", "prev_id": "initial-1", "next_id": "initial-2", "value": "x"} or {"operation": "insert", "id": "clientB-1", "prev_id": "clientA-1", "next_id": "initial-2"], "value": "y"}
- In the above examples, after initial text and operation data are broadcasted, each client node should have a copy of:

  ["initial-1", "clientA-1", "clientB-1", "initial-2" ... "initial-11"]

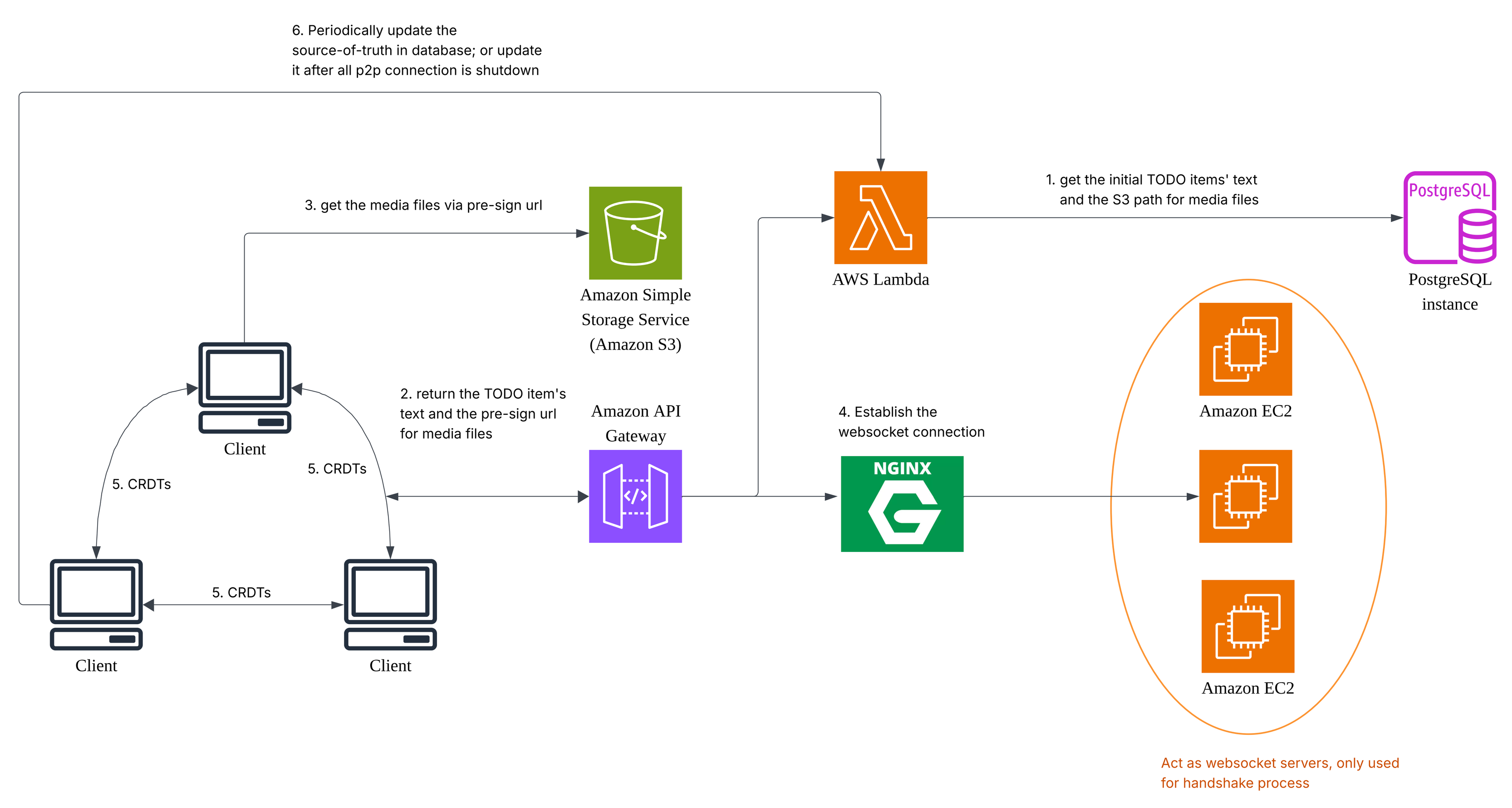
  and

  a key-value map linking the unique id and the character value: {"initial-1": "H", "clientA-1": "x", "clientB-1": "y"... "initial-11": "d"}

- "Hxyello World" will be rendered in each client node

- Since we need to keep the whole insert operation log, therefore for deletion operation, we will only store which insertion operation should be deleted, instead of physically delete a character or an insert operation log
- The delete operation data should look like: {"operation": "delete", "delete_id": "clientA-1"}
- After that, each client node should have a copy of:
  insertion log: ["initial-1", "clientA-1", "clientB-1", "initial-2" ... "initial-11"]
  deletion log: ["clientA-1"]
  key-value map: {"initial-1": "H", "clientA-1": "x", "clientB-1": "y"... "initial-11": "d"}

- "Hyello World" will be rendered in each client node



	Operational Transform with cerntralized servers	Conflict-free Replicated Data Types with peer-to-peer connection
Pros	• No complicated data strcuture in client side • More bandwidth efficient as there will be only one communication channel between a client and the websocket server • More efficienct client side rendering when there is large number of user editing the same TODO list, or the amount of operation is large	• Lower server side resource required • Robust operation during internet interruption as the operation log will be replicated to each client eventually • Seamlessly supporting offline editing as the global operation sequence does not matter
Cons	• Complicated scaling logic • Graceful shutdown must be implemented carefully between nginx and websocket servers during scaling down • Higher server side cost	• High memory consumption in client side • Heavier bandwidth loading as each client node needs to keep sending and receiving operation log to (from) all other client nodes • Extra consideration on updating the source-of-truth in the database in edge cases

Scaling strategy

- Global deployment:
  100M daily users mean the traffic coming across the globe.
  Sever clusters should be deployed in different geographic location to reduce latency

- Relational database:
  Command Query Responsibility Segregation (CQRS):
  For database mutation (create,update,or hard delete), the action should be performed on the database master node. For any read (get) query, the traffic should be routed to replica node

  Database sharding:
  If database performance is a big concern, we may use MongoDB, and resolve entity relations in the server, so that database sharding for replica node is easier to be implemented.

  Auto-scaling:
  Replica node can be horizontally scale up/down according to memory or cpu usage. (costly operation, not preferred)

- Media files:
  Cache in CDN (AWS Cloudfront) to reduce latency and S3 cost

- Websocket servers:
  Horizontally scale up/down according to memory usage and cpu usage
  Customized traffic routing logic should be implemented in nginx to ensure all clients editing the same TODO list establish websocket connections to the same websocket server
  Consistent hashing according to TODO list id is an option.
  Graceful shutdown should be implemented to ensure there is no active websocket connection before shutdown