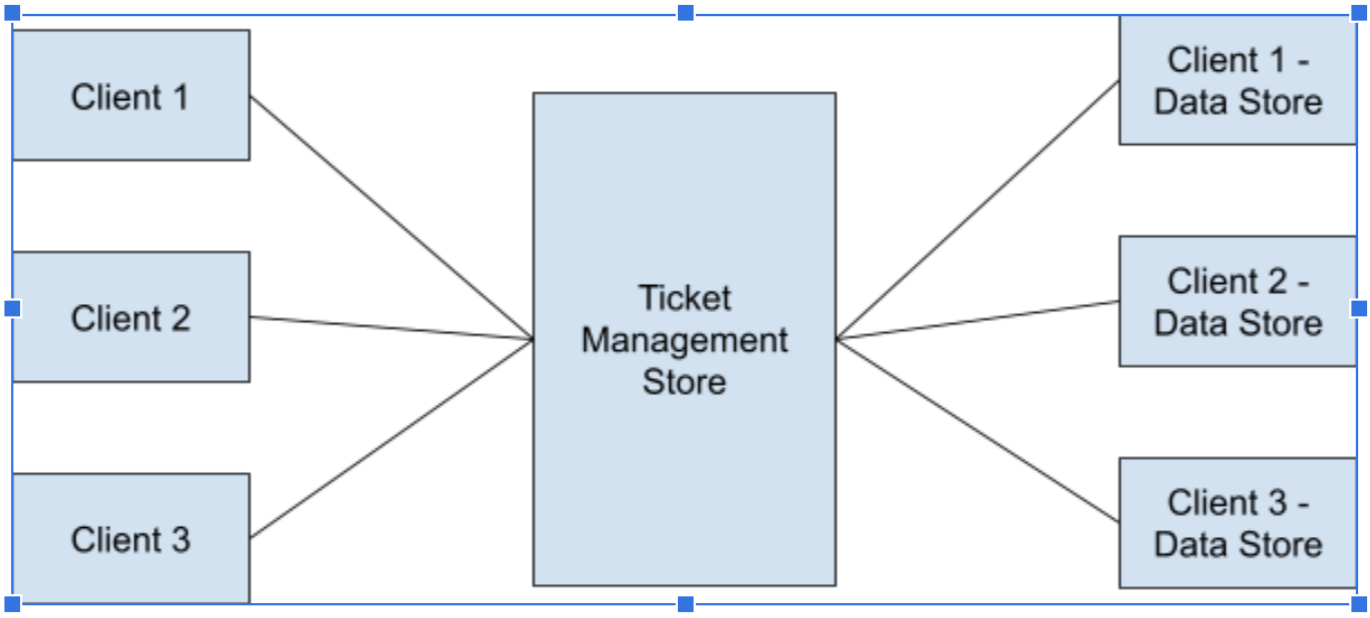
Ticket Management System Design

Functional Requirements

- Multiple clients will connect to a ticket management system.
- Ticket can be of different types based on SLAs: GOLD — within 2hrs, SILVER — within 4 hrs, BRONZE — within 8 hrs.
- Given, every ticket takes 1 minute to resolve.
- On receiving a ticket, create a ticket only if SLA can be guaranteed.
- Ticket must be created in client specific data store.

Non Functional Requirements

- Highly Available
- Consistent
- Ultra low latency
- Multi Tenant [Note: DB Management is choice of Developer]



Base Architecture

Public Endpoints

- /create_ticket
  - inputs: ticket_type, client_id
  - outputs: status, ticket_id
- /cancel_ticket
  - input: ticket_id
  - output: status
- /create_account
  - inputs: client_name, official_email, password
  - outputs: status, client_id
- /login
  - input: client_id
  - output: sub_domain
- /logout
  - input: client_id
  - output: status

Internal Endpoints [Within VPN]

- /validate_eta
  - inputs: client_id, ticket_type
  - outputs: status, slot_id
  - Note: slot is soft assigned [in pending state]
- /allocate_slot
  - inputs: client_id, ticket_type, slot_id
  - outputs: ticket_id

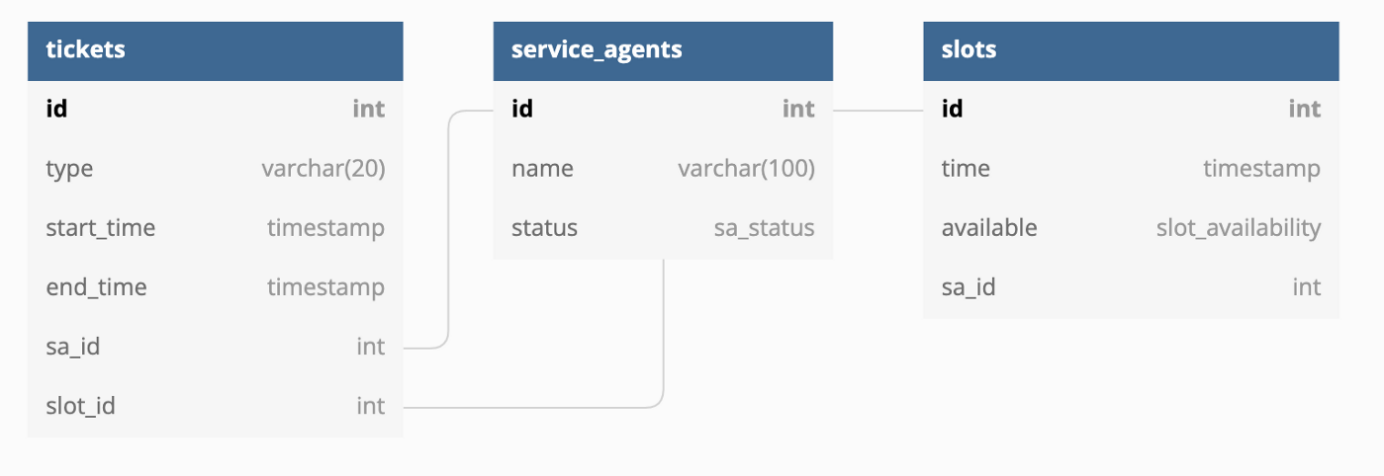
Choices

- DB type
  1. NoSQL — Scalable but not consistent
  2. SQL — Consistent but not scalable

- DB Designs
  1. Single DB for all clients: Only vertically scalable and also no isolation among client data
  2. Single DB but different schema: Isolated but not horizontally scalable
  3. Multiple DB: 1 or more DB/client

DB Schema

- Every client is going top have its own DB.
- The below schema is per client DB.




```
// Creating tables
Table tickets as t {
  id int [pk, increment] // auto-increment
  type varchar(20)
  start_time timestamp
  end_time timestamp
  sa_id int
  slot_id int
}


Table service_agents as sa {
  id int [pk, increment] // auto-increment
```

```
  name varchar(100)
  status sa_status
}


Table slots {
  id int [pk, increment] // auto-increment
  time timestamp
  available slot_availability
  sa_id int
 }

// Creating references
// You can also define relaionship separately
// > many-to-one; < one-to-many; - one-to-one
Ref: t.sa_id < sa.id
Ref: t.slot_id - slots.id

//// -- Enum
Enum slot_availability {
  available
  unavailable
}

Enum sa_status {
  available
  unavailable
}
```

How to validate ETA

- Before even creating a ticket, we have to check if the ticket can be resolved within requested ETA [based on ticket type: GOLD, SILVER and BRONZE]
- So at the moment plan is to validate the ETA using internal endpoint /validate_eta, then if can be accommodated put the request into a message bus like KAFKA or RabbitMQ as well as notify the Client.
- A simple query to slots table in the specific client DB to get the least available time that is available and within the requested ETA from the current time would work. If I get a timestamp back from query, then possible else not.
- Query: select min(time) from slots where available = 'available' and time > now() and time < now() + now() + eta_duration

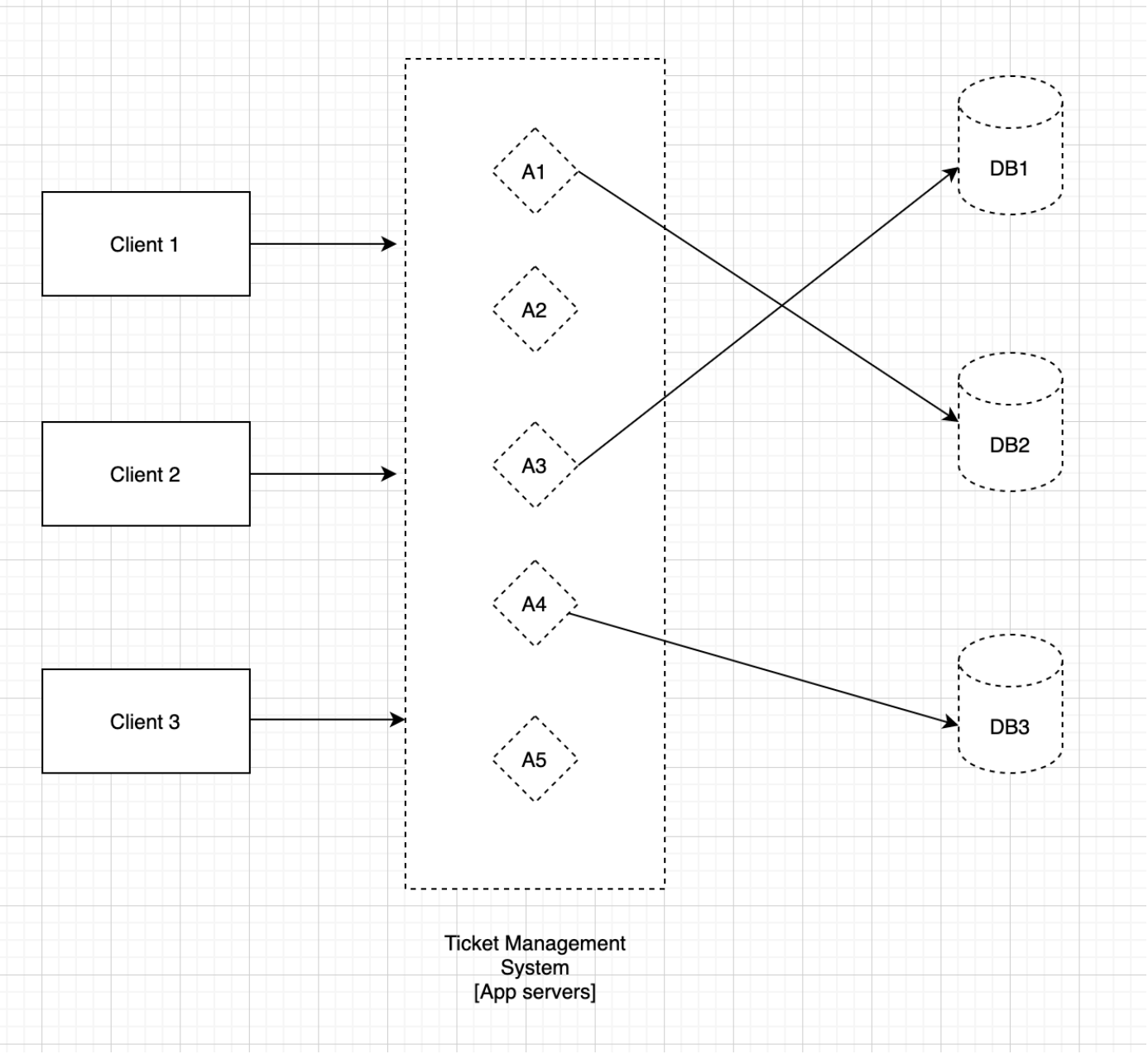
---

Why Multi-Tenant?

Advantages of multi-tenant model

- The client here itself is a big entity and not an individual customer. So the traffic they draw in will be significant. Single tenant model has scalability issues.
- Upgrades will be costlier and hard to manage. For about 1000 clients if each client has its own ticketing system. Once we plan to upgrade, we have to do it across clients which is costly.
- Outage protection.
- Stateless app servers.
- Horizontally scalable.

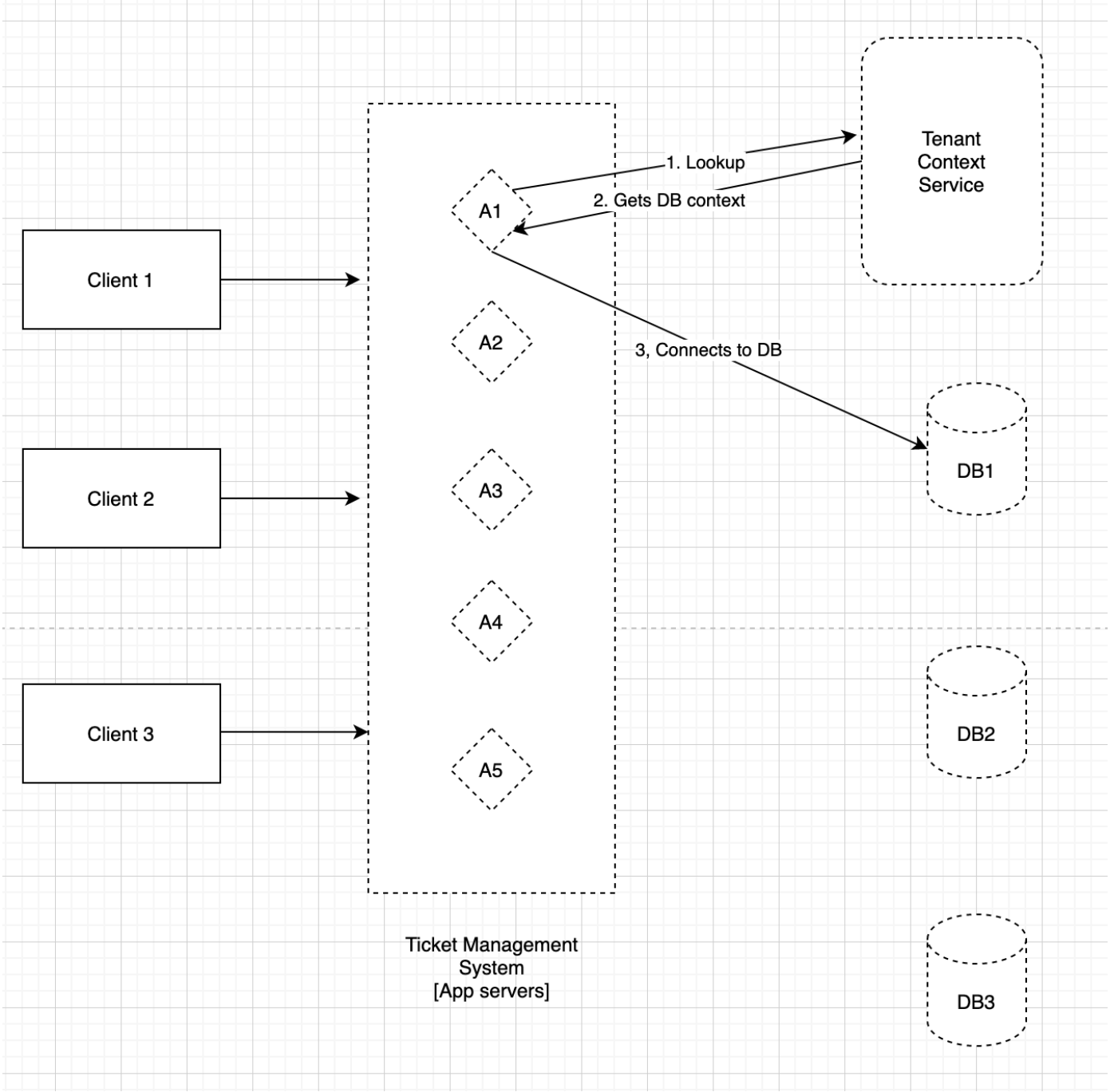
HLD — Attempt 1



Issues with this setup:

- App servers have to figure out which DB to connect per request.
- Handful of app servers might have to handle 1000s of DBs.

HLD — Attempt 2



Issues with this setup:

- Introduction of another service to store DB context adds another hub and as records of Tenant Context Service grows it will get slower and will add to latency.
- This service becomes a SPOF if not scaled.
- We require this to have:
  1. Ultra low latency.

2. Highly scalable.
3. Consistent.

PACELC and CQRS

PACELC [extension of GAP theorem] states that:

In case of PARTITION

- Choose either AVAILABILITY + CONSISTENCY
- or LATENCY + CONSISTENCY

Solution to PACELC limitation is CQRS.

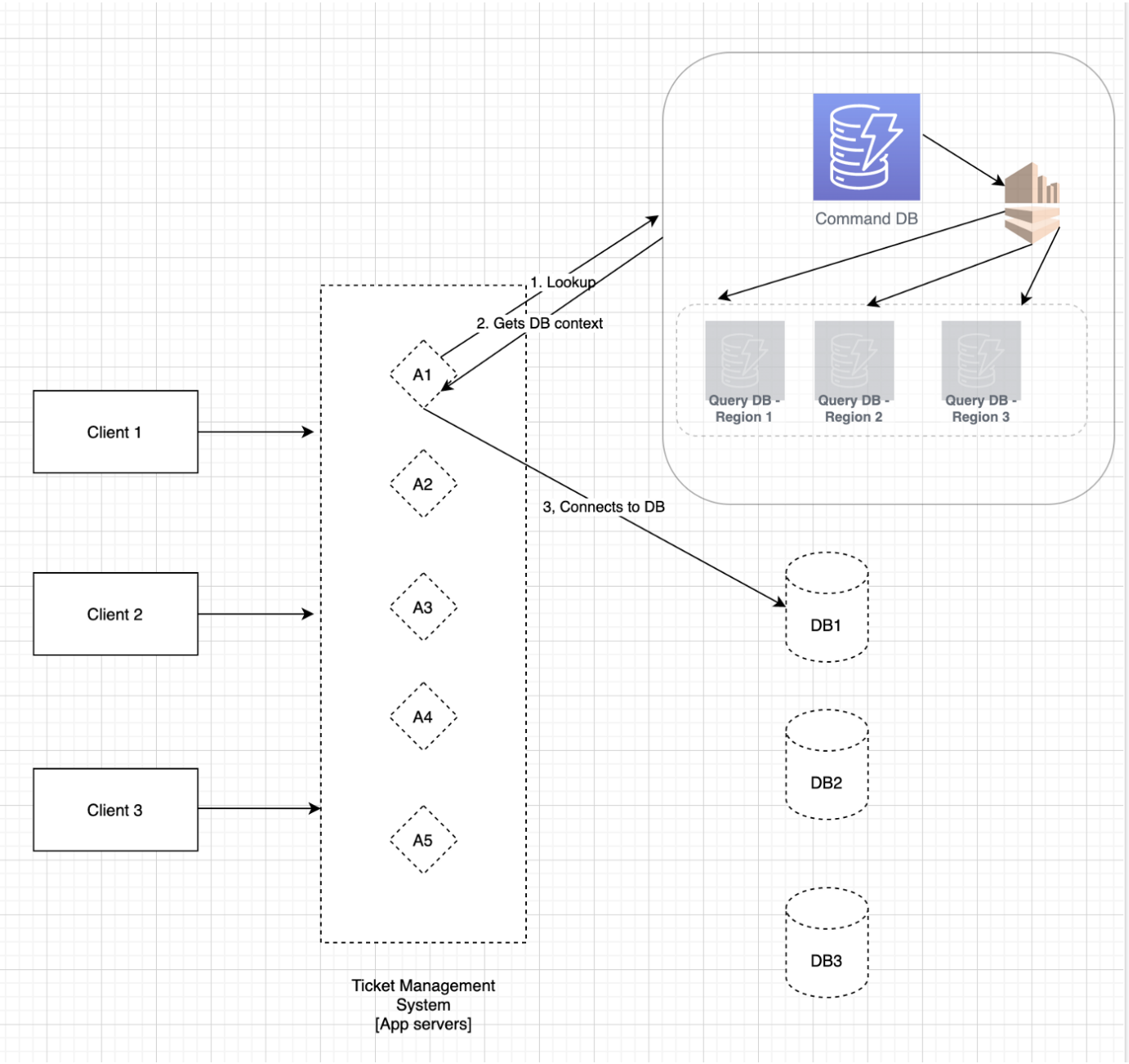
- Use a different models [in app or DB] to update info and read info.
- The idea is to do computation once at the time of writing and use it no matter how many times.
- So we can divide the DB of Tenant Context Service into Command DB and Query DB.
- With Command DB we will achieve consistency.
- With Query DB we will achieve scalability and HA.
- But the whole system is going to be eventually consistent.

Finalising the DB model

- At this moment we can finalise the DB/DB type for Command and Query models.
- I have chosen a NoSQL DB — DynamoDB to be specific as the data to be stored is going to be precisely key-value type. For both Command and Query models.
- Also the plan is to spread Query Model across regions of AWS (using AWS as the platform to deploy).
- Synchronisation between Command and Query DBs can be done via AWS Kinesis Streams.

- Using DynamoDB gives us 99.99% availability which translates to 52 min downtime per year. But since we duplicated it across region it gives us availability of 99.9999%.
- Also the individual client DBs are going to Relational, choosing Postgres.

HLD — Final



Folks, if you like my content, would you consider following me on linked in at: https://www.linkedin.com/in/hitesh-pattanayak-52290b160/