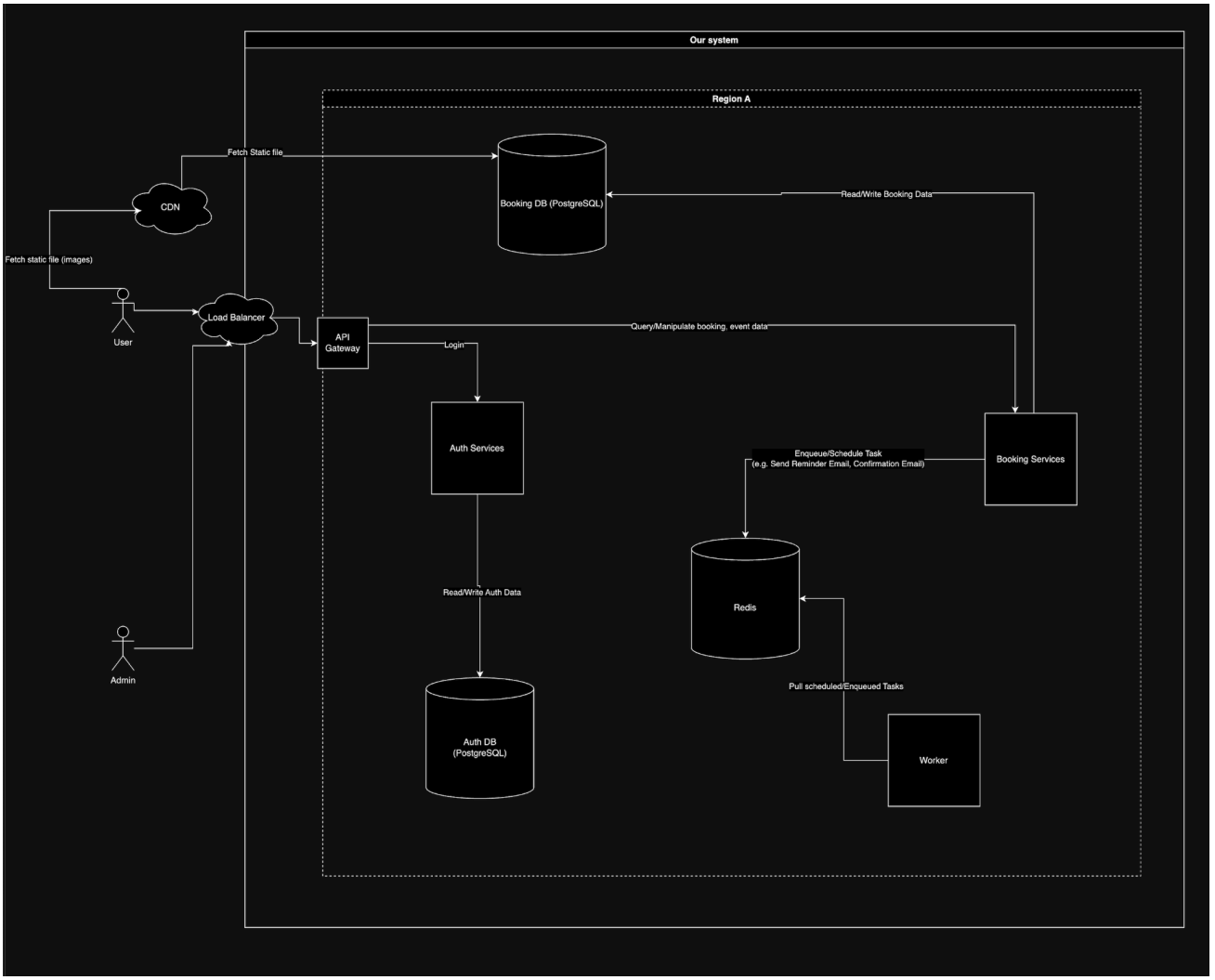
High-Level Architecture Documentation for the Event Booking System

Overview

This document provides a high-level architecture overview for the global event booking platform, integrating user bookings, admin operations, and payment processing with Stripe. The system is designed to operate in a multi-region environment, ensuring high availability, fault tolerance, and low latency for global users. It also addresses scalability challenges using Redis, API Gateway, and distributed microservices.

Architecture Diagram



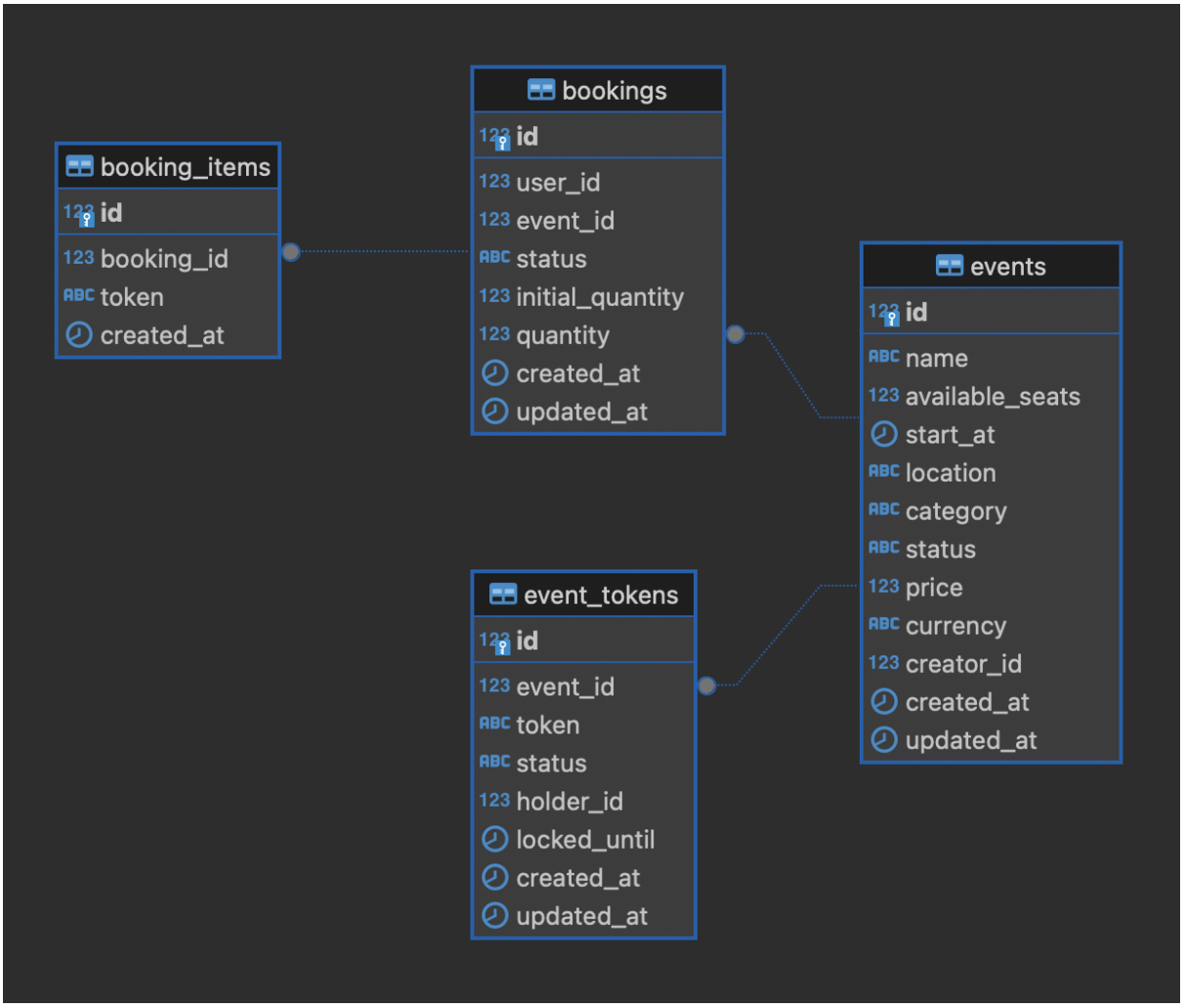
Components

These are the core components (for the current solution)

- Load Balancer: We need this to spread the load evenly to API Gateway
- API Gateway: The place to handle authenticate, authorize, rate limit, and forward the valid (already AuthZ, AuthN) incoming request to the correct services based on configured rules
- Auth Service: Service to handle authentication, authorization
- Booking Service: The service handles business logic (e.g. List events, retrieve event details, create an event, update events, create booking requests, confirm booking requests, etc,...)
- Worker: Handle all background tasks. In this case, we use Redis as the storage to store task's information
- CDN: To serve static files

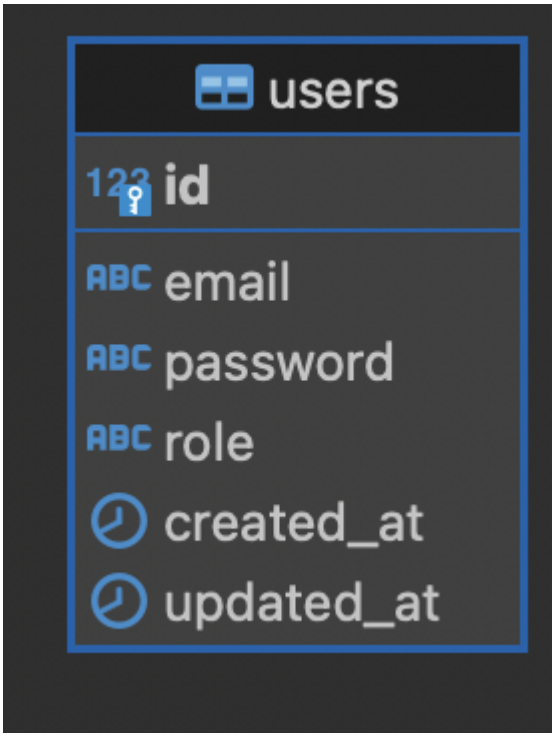
Database design

Booking DB



There is one thing special about my solution is that at the time the event created, we **also generated the tokens** for that event, one token stands for one available seat. This will help us reduce the lock when we create booking request, by apply **explicit locking** of **PostgreSQL**

Auth DB



Implementation

Code structure

- cmd/: Contains the main entry points for different services
  - background-tasks/: Background task service
  - central-auth/: Central authentication service
  - servers/: Main application server
- config/: Configuration files and related code
- internal/: Core application code
  - app/: Application-specific code
    - auth/: Authentication-related code
    - server/: Server-related code
    - worker/: Worker-related code
  - common/: Shared utilities and contexts
    - appcontext/: Application context-related code
    - util/: Utility functions
  - infra/: Infrastructure-related code
    - asynq/: Asynchronous task processing

        - emailsender/: Email sending functionality
        - paymentgateway/: Payment gateway integration
        - posgresql/: PostgreSQL database integration
        - redis/: Redis integration
    - middleware/: HTTP middleware
    - modules/: Core business logic modules
        - auth/: Authentication module
            - model/: Core models that are used for all layers
            - repository/: Data access layer
                - entity/: Entity for data accessing in the Data Access Layer
                - store/: Contain code to access the DB layer directly
                - client/: Contain client to connect to the external source via API, third-party SDKs
            - services/: Business logic layer
            - transport/: API layer (HTTP handlers, etc.)
        - booking/: Booking module
            - model/: Core models that are used for all layers
            - repository/: Data access layer
                - entity/: Entity for data accessing in the Data Access Layer
                - store/: Contain code to access the DB layer directly
                - client/: Contain client to connect to the external source via API, third-party SDKs
            - services/: Business logic layer
            - transport/: API layer (HTTP handlers, etc.)
- migrations/: Database migration files
- doc/: Documentation files

Note:

- *Importing across modules is not allowed. This ensures we split the modules correctly, so they can be deployed separately as microservices.*
- *Layers in each module are abstracted enough by using interfaces*

Use cases

In this part, I also instruct you how to use (test).

Before executing any API call, please login to get the token first. This is the pre-created user, so we can use it

```
curl --location 'http://localhost:5000/api/v1/login' \
--header 'Content-Type: application/json' \
--data-raw '{
  "email": "test@example.com",
  "password": "correctpassword"
}'
```

If it’s successful, the `access_token` will be returned. And we will use this **access_token** as the **Bearer token** (e.g Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJlbWFpbCI6InRlc3RAZXhhbXBsZS5jb20iLCJyb2xlIjoiIiwiZXhwIjoxNzI5ODU0Mzc1fQ.iKGqNm4-nz4oYGQDu7qJPw29x-iNjAFtXLIgQiGy0k4) in **Authorization** header

To retrieve event detail

```
curl --location 'http://localhost:5000/api/v1/events/{{event_id}}' \
--header 'Authorization: ••••••'
```

Replace event_id with the real id

To get events

```
curl -XPOST --location 'http://localhost:5000/api/v1/search/events' \
--header 'Content-Type: application/json' \
--header 'Authorization: ••••••' \
--data '{
  "location": "HCM",
  "name": "event1"
}'
```

Can add these params to body to filter

- category: string
- location: string
- start_from: string (e.g. 2024-11-01T10:00:00Z)
- start_to: string (e.g. 2024-11-01T10:00:00Z)
- name: string

To update event

```
curl --location --request PUT 'http://localhost:5000/api/v1/events/{{event_id}}' \
--header 'Content-Type: application/json' \
--header 'Authorization: ••••••' \
--data '{
  "status": "active"
}'
```

status can be `active` or `inactive`.

To create event

```
curl -XPOST --location 'http://localhost:5000/api/v1/events' \

--header 'Content-Type: application/json' \

--header 'Authorization: ••••••' \

--data '{
```

```
  "name": "event13",
  "available_seats": 10,
  "start_at": "2024-11-01T10:00:00Z",
  "location": "HCM",
  "category": "MUSIC",
  "price": 10.5
}'
```

To create booking request

```
curl -XPOST --location 'http://localhost:5000/api/v1/bookings' \
--header 'Content-Type: application/json' \
--header 'Authorization: ••••••' \
--data '{
   "event_id": 31,
   "quantity": 11
}'
```

Can only create booking request on **active events**

To confirm booking

```
curl --location --request PUT 'http://localhost:8080/api/v1/bookings/3/confirm' \
--header 'Authorization: ••••••'
```

This is the checkout step, in real world usecase we will make payment

To cancel booking

```
curl --location --request PUT 'http://localhost:5000/api/v1/bookings/2/cancel' \
--header 'Authorization: ••••••'
```

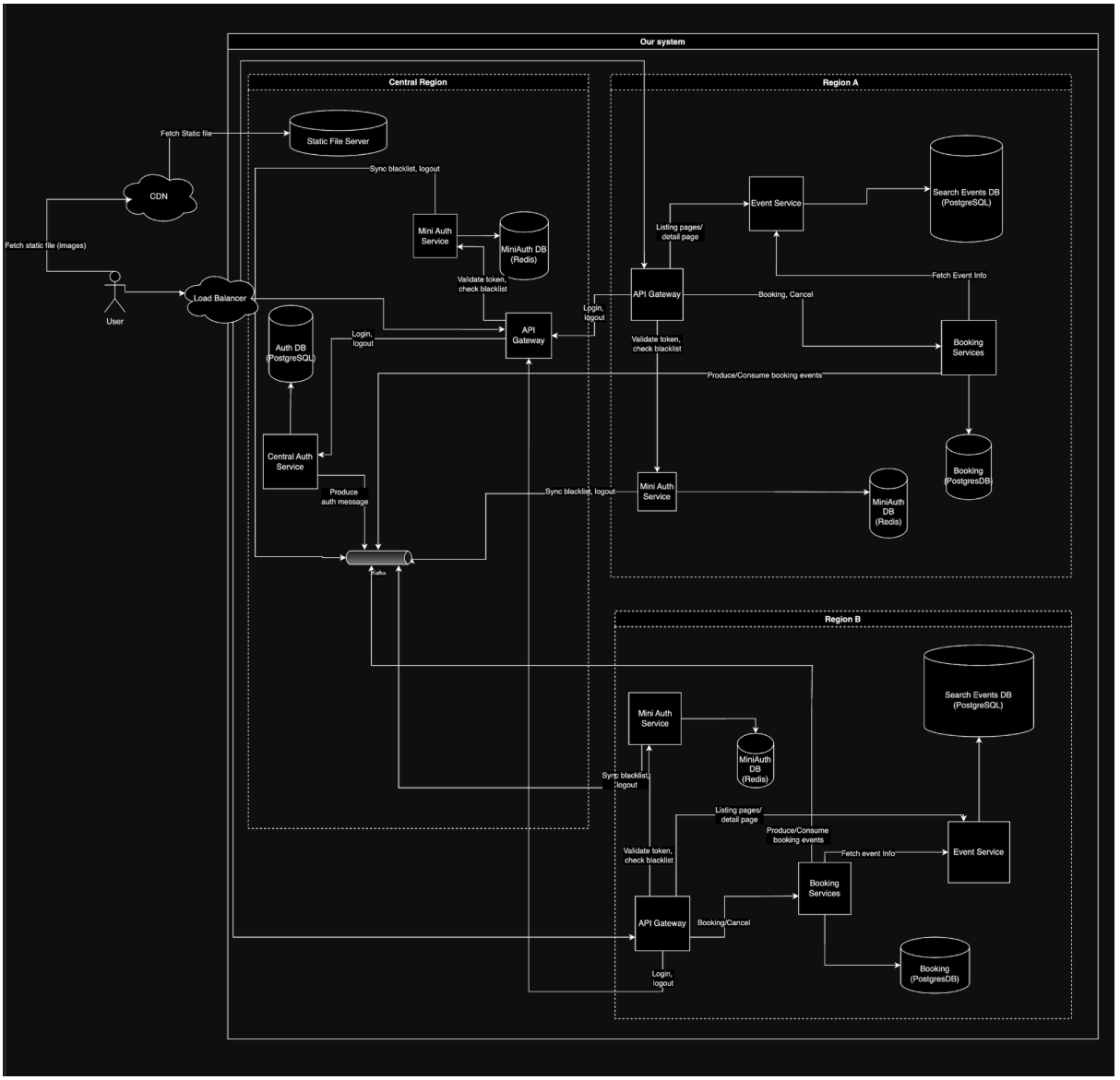
Can only cancel if the event is not started yet

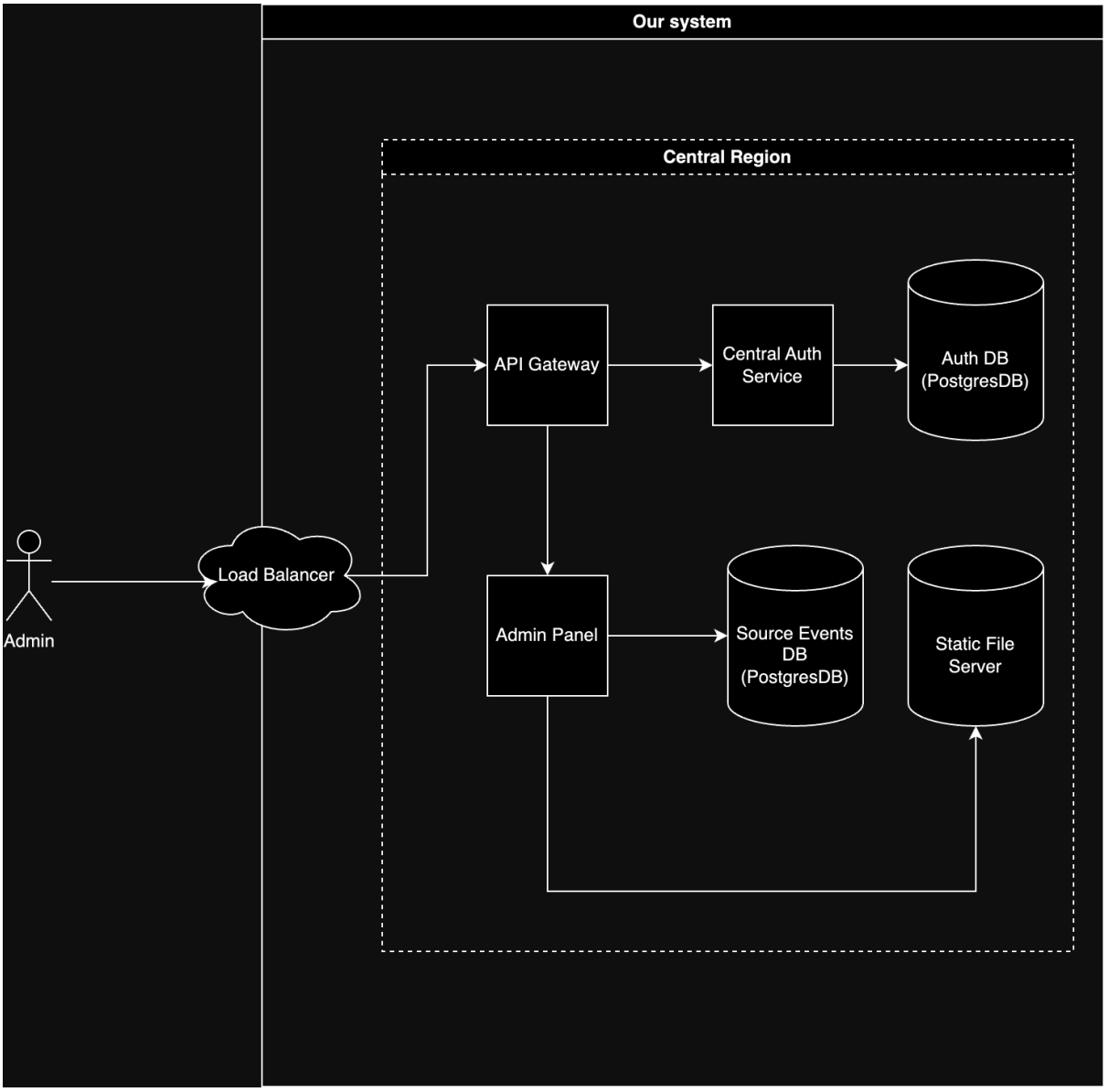
Retrive booking detail

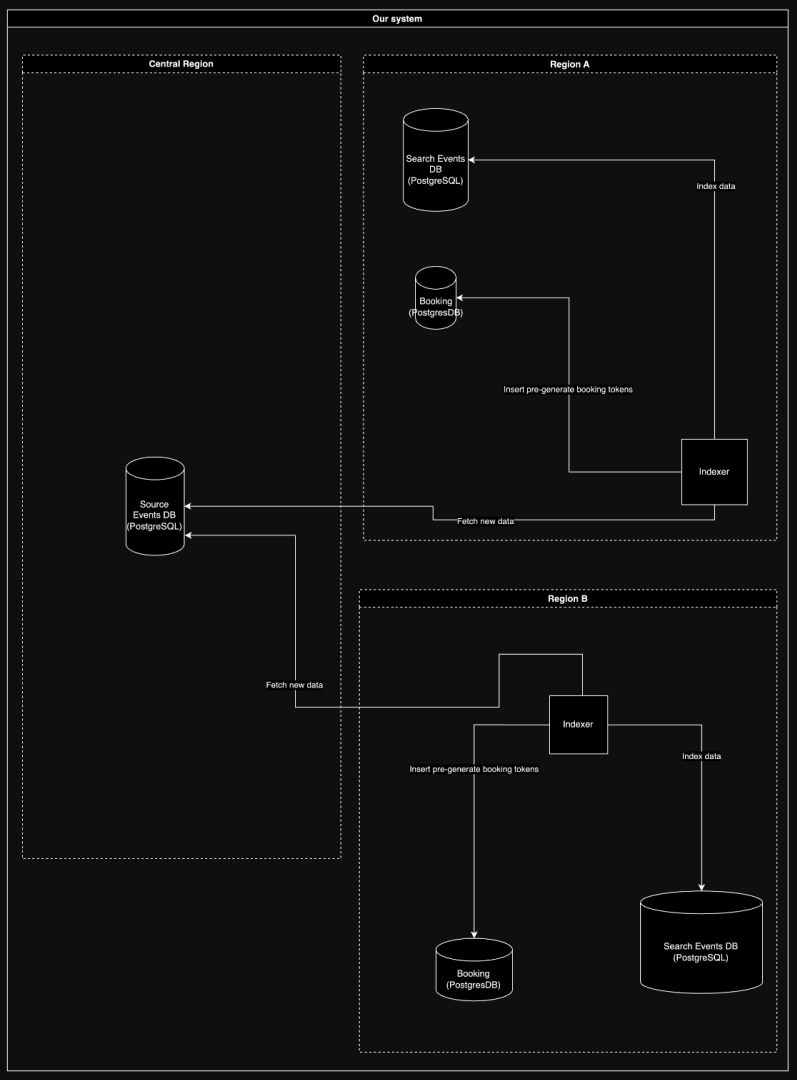
```
curl --location 'http://localhost:5000/api/v1/bookings/{{booking_id}}' \
--header 'Authorization: ••••••'
```

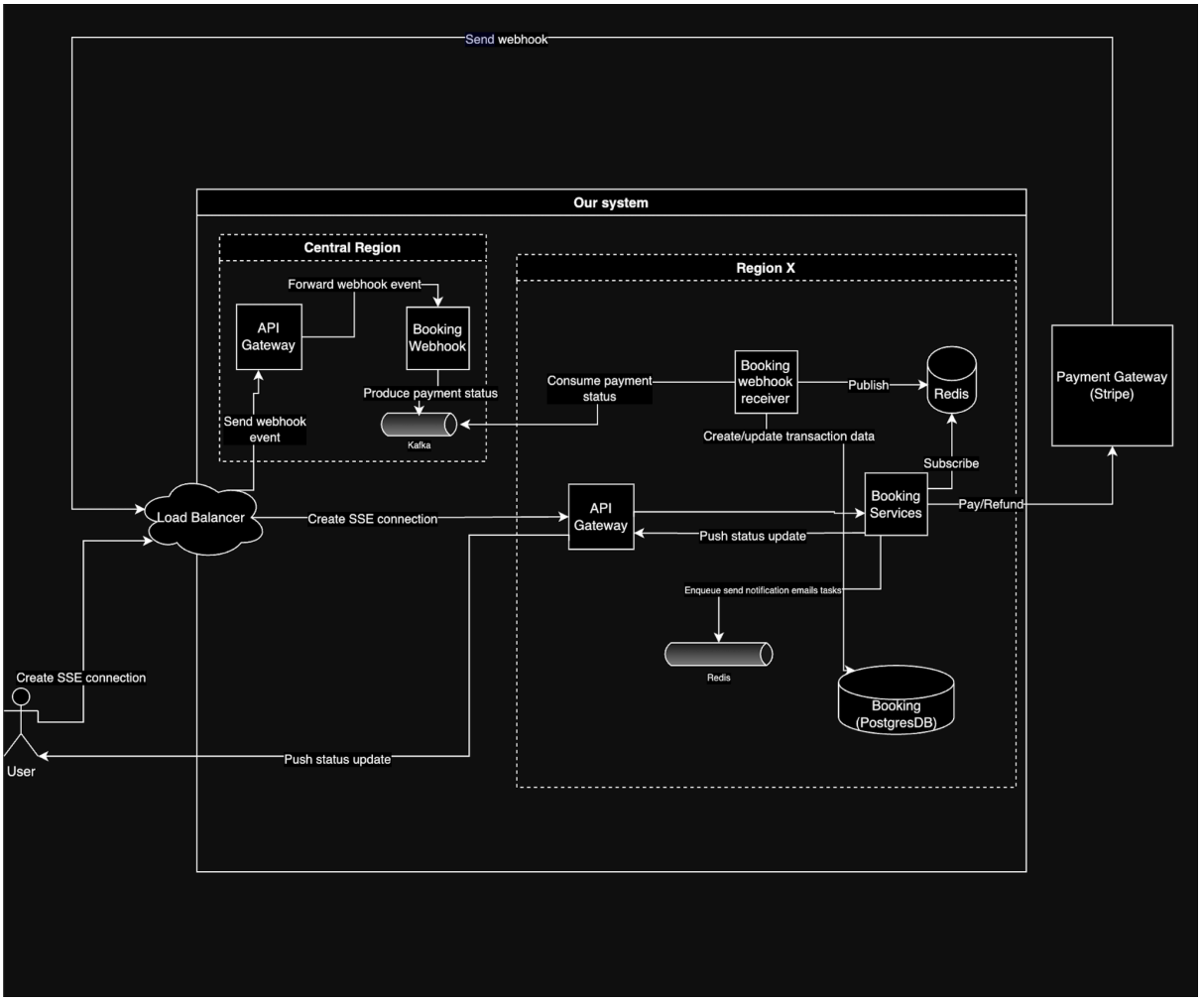
Future enhancements

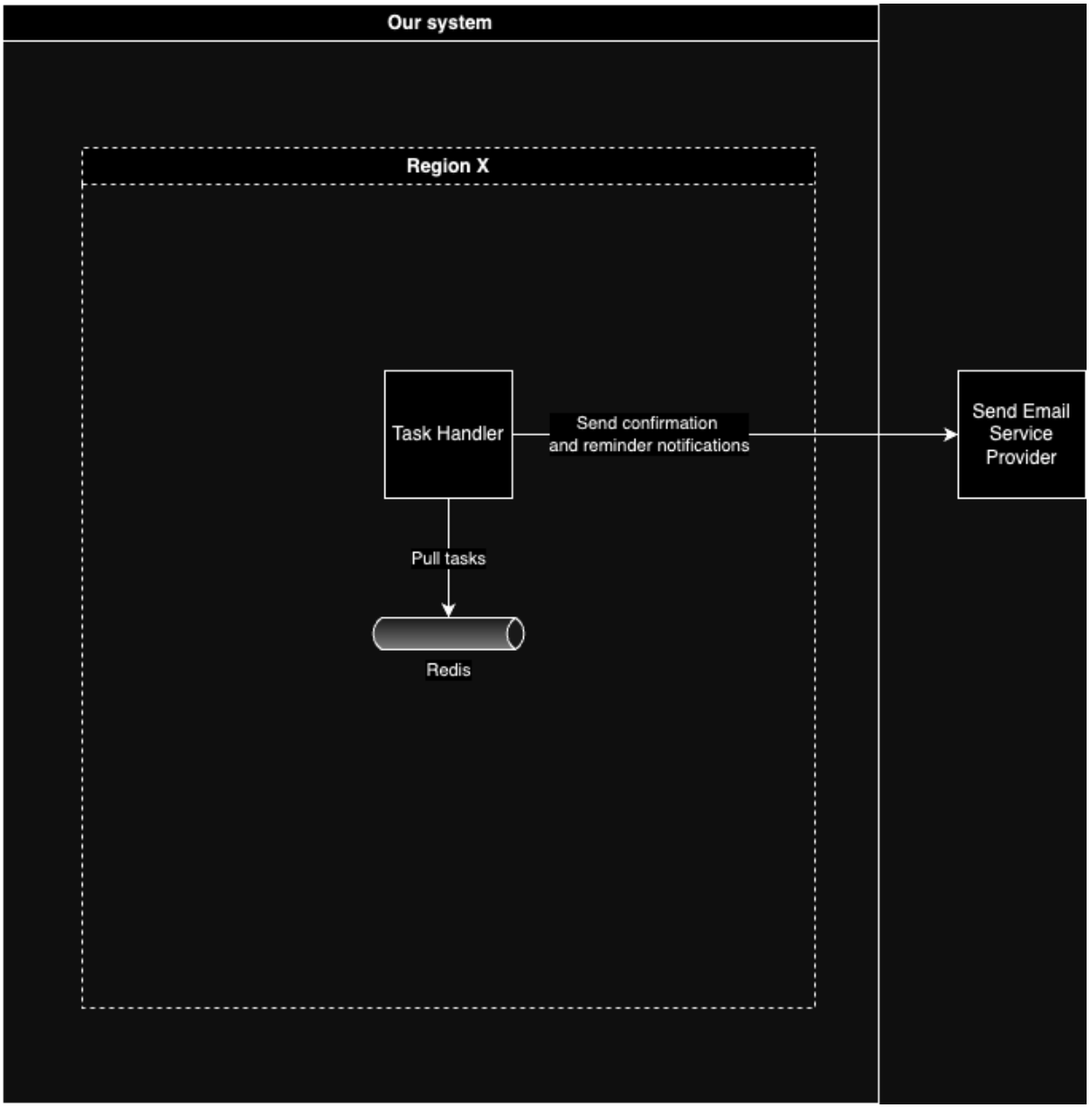
- Support more notification channels
- Add metrics, monitoring, alert
- Use Elastichsearch if the autocomplete, fuzzy search, and free-text search event name are critical features
- If we don’t want to use ES, can consider using pg_trgm plugin in Postgres with the GIN index, so we can free-text search the event name effieciently
- Consider using Kafka as the core distributed message system, to keep data consistency and have extremely good performance under high load so we can deploy a multi-regional system
- To make the system globally good, This is what I think it would look like




Our system
Central Region
API Gateway
Central Auth Service
Auth DB (PostgresDB)
Admin
Load Balancer
Admin Panel
Source Events DB (PostgresDB)
Static File Server


Our system
Central Region
Region A
Search Events DB (PostgreSQL)
Index data
Booking (PostgresDB)
Insert pre-generate booking tokens
Indexer
Source Events DB (PostgreSQL)
Fetch new data
Region B
Fetch new data
Indexer
Insert pre-generate booking tokens
Index data
Booking (PostgresDB)
Search Events DB (PostgreSQL)


Send webhook
Our system
Central Region
Forward webhook event
API Gateway
Booking Webhook
Produce payment status
Send webhook event
Kafka
Region X
Consume payment status
Booking webhook receiver
Publish
Redis
Create/update transaction data
Subscribe
Payment Gateway (Stripe)
Load Balancer
Create SSE connection
API Gateway
Booking Services
Pay/Refund
Push status update
Enqueue send notification emails tasks
Redis
Booking (PostgresDB)
Create SSE connection
User
Push status update



There are a few new introduced service

- MiniAuth: to validate the token in each region without calling to central auth db
- Indexer: sync the change from the source Booking (source event DB) to regional DB
- Kafka: Distributed message system
- At this moment, Load Balancer must be a global Load Balancer which has ability to know which region is the healthy and nearest region that it can forward the request to