Distributed Systems for Data Engineering
Class Group Project: Kayak Simulation

Project Due Date (Presentation and Demo): 1st December 2025 and 8th December 2025

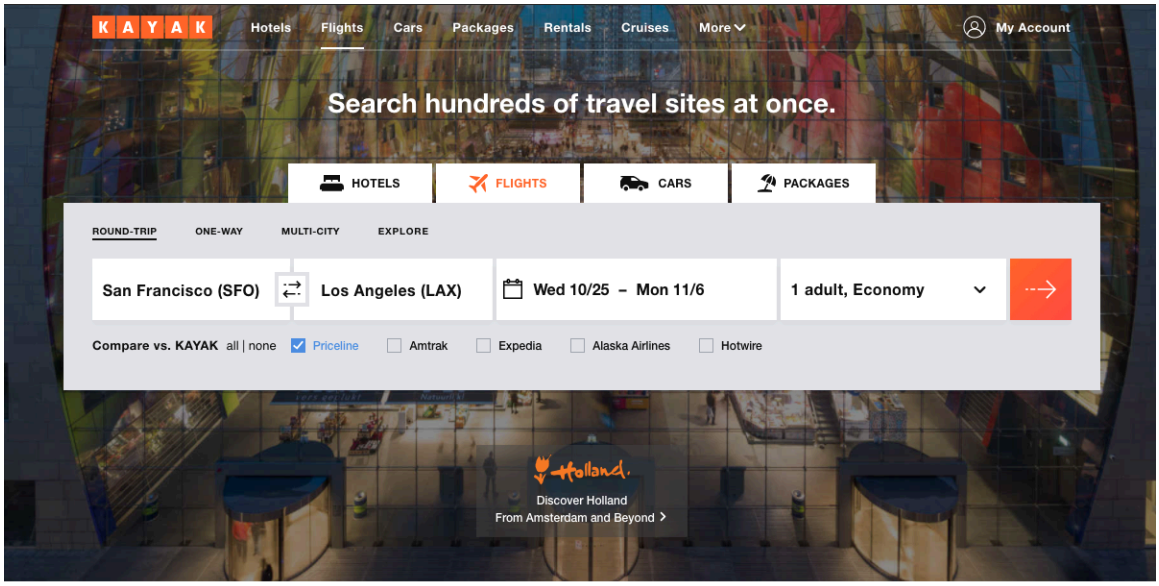
Turn in the following on or before November 3, 2025. No late submissions will be accepted!

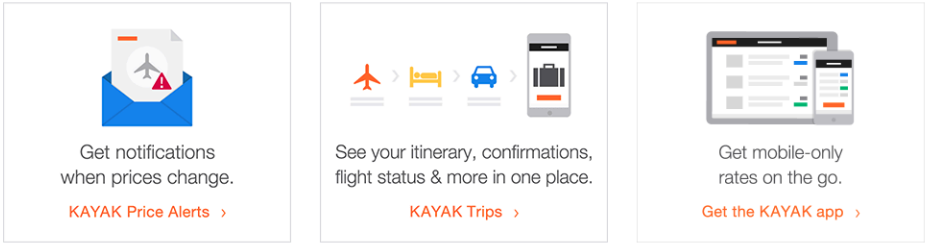
- An API design document of services provided by the application.

You are allowed to use GenAI tools for this group project only

History & About Kayak

KAYAK is a travel metasearch and booking platform. It aggregates options for flights, hotels, and rental cars from multiple providers, enabling travelers to search, compare, and choose the option that best fits their needs. Your simulation will emulate these core functions- search, filter, book, bill, and analyze usage within a distributed, service-oriented architecture.





Project Overview
In this project, you will design a 3-tier application that implements the functions of a Kayak-like travel system. You will build on the techniques used in your lab assignments to create the system.

In the Kayak System, you will manage and implement different types of objects:

- Users
- Listings (Flights , Hotels, Cars)
- Billing
- Admin

For each type of object, you will also need to implement an associated database schema that will be responsible for representing how a given object should be stored (relational and/or document, as appropriate).

Your system should be built upon a distributed architecture. You have to use REST-based web services and message queues (Kafka) as middleware technology. You will be given a great deal of "**artistic liberty**" in how you choose to implement the system.

System Entities

You must support the following entity types (minimum). You may add attributes/relations as needed and define primary/foreign keys across services.

Users - It represents information about an individual user registered on Kayak. You must manage the following information for this entity:

- User ID [SSN Format]
- First Name
- Last Name
- Address
- City
- State
- Zip Code
- Phone Number
- Email
- Profile Image
- Credit Card / Payment Details
- Booking History (Past / Current / Future Trips)
- Reviews submitted for flights, hotels, and cars

Flights - It represents information about available flights that users can search and book through Kayak. You must manage the following information for this entity:

- Flight ID (e.g., AA123)
- Airline / Operator Name
- Departure Airport
- Arrival Airport
- Departure Date and Time
- Arrival Date and Time
- Duration
- Flight Class (Economy / Business / First)
- Ticket Price
- Total Available Seats
- Flight Rating
- Passenger Reviews

Hotels - It represents information about hotels listed on Kayak. You must manage the following information for this entity:

- Hotel ID
- Hotel Name
- Address
- City
- State
- Zip Code
- Star Rating
- Number of Rooms
- Room Type (Single, Double, Suite, etc.)
- Price per Night
- Amenities (e.g., Wi-Fi, Breakfast, Parking)
- Hotel Rating
- Guest Reviews
- Images of Rooms and Property

Cars - It represents information about cars available for rent through Kayak. You must manage the following information for this entity:

- Car ID
- Car Type (SUV, Sedan, Compact, etc.)
- Company / Provider Name
- Model and Year
- Transmission Type

- Number of Seats
- Daily Rental Price
- Car Rating
- Customer Reviews
- Car Availability Status

Billing - It represents information related to payment and transaction details for bookings made by users. You must manage the following information for this entity:

- Billing ID
- User ID
- Booking Type (Flight / Hotel / Car)
- Booking ID
- Date of Transaction
- Total Amount Paid
- Payment Method (Credit Card, PayPal, etc.)
- Transaction Status
- Invoice / Receipt Details

Administrator - It represents information about the administrators responsible for managing the Kayak system. You must manage the following information for this entity:

- Admin ID
- First Name
- Last Name
- Address
- City
- State
- Zip Code
- Phone Number
- Email
- Role / Access Level
- Reports and Analytics Managed

Your project will consist of three tiers:

- The client tier, where the user will interact with your system
- The middle tier/middleware/messaging system, where the majority of the processing takes place
- The third tier, comprising databases to store the state of your entity objects

Tier 1 - Client Requirements

The client will be an application that allows a user to do the following:

User Module/Service

- Create a new User (use Kayak fields as guidance)
- Delete an existing User
- Change a user's information (name, address, profile image, etc.) - *This function must support the ability to change ALL attributes*
- Display information about a User
- Search listings for different categories (Flights, Hotels, Cars)
- Filter listings
    - Filter hotels by stars, price
    - Filter flights by departure/arrival times, price
    - Filter cars by car type, price
- Book a hotel/flight/car
- Make Payment
- View Past/Current/Future bookings

Admin Module/Service

- Allow only authorized (admin) users to access Admin Module
- Add listings (hotel/flight/car) to the system
- Search for a listing and edit it
- View/Modify user accounts
- Search for a Bill based on attributes (by date, by month)
- Display information about a Bill

Sample Admin Analysis Report
The Admin dashboard will present graphs/charts such as:

- **Top 10 properties** with revenue per year (bar/pie/any chart)
- **City-wise revenue** per year (bar/pie/any chart)
- **10 hosts/providers** with the maximum properties sold last month and associated revenue

Sample Host (Provider) Analysis Report
(From logs and database as applicable)

- Graph for clicks per page (bar/pie/any chart)
- Graph for property/listing clicks (bar/pie/any chart)
- Capture the area/section which is least seen
- Graph for reviews on properties (database)
- Trace diagram for tracking one user or a cohort (e.g., users from San Jose, CA)
- Trace diagram for tracking bidding/limited offers for an item (if implemented)

You may add any extra functionality you want (optional, not required), but if you do so, you must document and explain the additional features. You still must implement all the required features, however.
The client should have a pleasing look and be simple to understand. Error conditions and feedback to the user should be displayed in a status area on the user interface. Ideally, you will use an IDE tool to create your GUI.

Tier 2 - Middleware

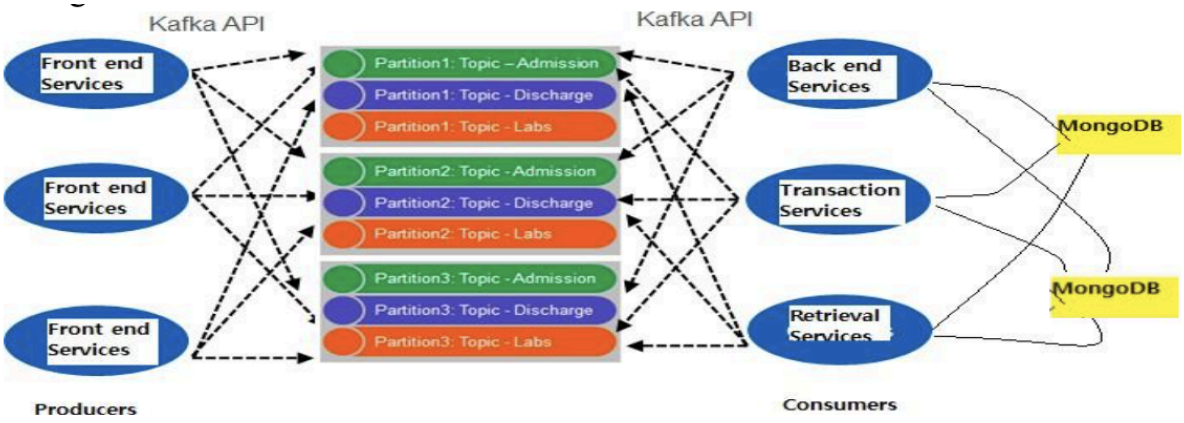
You will need to provide/design/enhance middleware using Kafka that can accomplish the above requirements. Implement it using REST-based Web Services as middleware technology. Next, decide on the interface your service will expose. Ensure you appropriately handle error conditions/exceptions/failure states and return meaningful information to your caller.

Handle the specific failure cases described below.

ON OR BEFORE the listed date, you must turn in a document describing the **API interface** your server will use (request–response descriptions).

Your project should also include a **model/data access layer** that uses standard modules to access your databases. Entity objects will use the data object to select/insert/update state.

Use **Kafka** as a messaging platform for communication between front-end channels and backend systems. Separate Node into two parts connected via message queues. Design "backend services" as consumer and "frontend services" as producer as shown below diagram.



Tier 3 - Database Schema and Database Creation

Design and create your databases. Choose column names and types that are appropriate for your data. Provide a simple program or script that creates your databases and tables/collections.

- Use **MySQL and MongoDB** appropriately (e.g., bookings/billing in MySQL; reviews, images, logs in MongoDB). Justify your choices when you are using which one

- Provide schema diagrams/samples and indexing where relevant.

Agentic AI Recommendation Service in FastAPI

This service is a multi-agent travel concierge that **periodically** discovers great deals (scheduled scans), reasons about user needs in conversation, and turns those insights into actionable trip recommendations. It behaves less like a search box and more like a proactive teammate that plans, explains, and adapts. Use FastAPI for HTTP + **WebSockets** to push updates to the UI. (FastAPI)

Deals Agent (backend worker): ingests small CSV/mock feeds on a schedule, detects & tags deals, and emits updates via websocket connection(feel free to use Kafka as well)

Concierge Agent (chat-facing): understands user goals, composes simple flight+hotel bundles from cached deals, explains tradeoffs, and sets price/stock watches. Model requests/responses with Pydantic v2 and persist normalized entities with SQLModel (SQLite locally). (Pydantic)

What the service does (at a glance)

- Understands intent & constraints in natural language (e.g., "weekend in Tokyo under $900, pet-friendly, near public transit").
- Finds and tags **flight + hotel** deals only (cars out-of-scope for MVP) using simple rules (e.g., price drop >15%, scarce inventory, limited-time promo).
  Builds coherent itineraries (bundles) that optimize for cost, convenience, and essential preferences.
- Explains recommendations (tradeoffs, why this hotel over that one, key fees, cancellation rules) with short, template-based snippets.
  Watches for changes (price drops, sold-out risk) on a schedule and notifies the UI asynchronously over WebSockets. (FastAPI)
- Answers questions about policies, logistics, and alternatives using fields present in the datasets (refundability, pets, breakfast, neighborhood).

Primary user journeys
Tell me what I should book
- User: "I've got Oct 25–27, SFO to anywhere warm, total budget $1,000 for two."
- Outcome: The service proposes 2–3 complete bundles with clear, comparable summaries (price, travel time, hotel neighborhood, cancellation terms), and a short "why this pick works for you."

Refine without starting over
- User: "Make it pet-friendly and avoid red-eye flights."
- Outcome: The service preserves earlier context, regenerates options that respect new constraints, and highlights what changed (e.g., "+ $38, earlier departure, 20-minute longer connection").

Keep an eye on it
- User: "Track this Miami package; alert me if it dips below $850 or inventory drops under 5 rooms."
- Outcome:

Decide with confidence
- User: "Is the Marriott rate actually good?"

- Outcome: It explains: “This is 19% below its 60-day rolling average for these dates; similar 4 star options nearby are $25–$60 higher per night.”

Book or hand off cleanly
- Outcome: The service returns a complete, validated quote (fare class, baggage, fees, cancellation) and hands off to a partner flow. (Keep the “quote” purely from available dataset fields in MVP.)

The Multi-Agent Roster

Deals Agent (backend worker)

Feed Ingestion: use Kafka as the ingestion bus. Option 1: **Kafka Connect FileStreamSource** reads CSV slices into a topic (e.g., raw_supplier_feeds). Option 2: a **small producer job** (scheduled) publishes CSV rows to Kafka. The Deals Agent consumes, normalizes currency/dates, and writes curated records to a **normalized** topic (e.g., deals.normalized).

Deal Detector: a Kafka consumer (in a consumer group) reads deals.normalized, applies rules (≥15% below 30-day avg, limited inventory, promo end), computes a small integer Deal Score, and produces results to deals.scored. Consumer groups enable parallelism and fault tolerance. Confluent Docs

Offer Tagger : enrich scored records using only existing metadata (Refundable/Nonrefundable, Pet-friendly, Near transit/Breakfast). No geo/NLP. Publish tagged offers to deals.tagged.

Emit Updates: produce concise events to Kafka topics (e.g., deal.events) keyed by listing/route for stable partitioning; downstream agents/services consume via **consumer groups**. In Python services, use **aiokafka** for asyncio producers/consumers.

Concierge Agent (chat-facing)
Intent understanding (dates, budget, constraints) with a single clarifying question max.
Trip Planner: compose flight+hotel bundles from cached deals; compute a Fit Score (price vs budget/median + amenity/policy match + simple location flag).

Explanations: “Why this” (≤25 words) + “What to watch” (≤12 words) using facts.

Policy answers: quote from listing metadata (refund window, pets, parking) with a short snippet.

Watches: set price/inventory thresholds and push async updates to clients via **FastAPI WebSockets**. Define a **/events** endpoint that relays new watch/deal events to connected sessions. Use **Pydantic v2** for payload schemas. (FastAPI)

Use these Kaggle datasets:

Hotels / nightly prices
Inside Airbnb (pick a city):
NYC: https://www.kaggle.com/datasets/dominoweir/inside-airbnb-nyc
(Alt, general mirror): https://www.kaggle.com/datasets/ahmedmagdee/inside-airbnb (Kaggle)

Hotel behavior

Hotel Booking Demand (City + Resort hotels):
https://www.kaggle.com/datasets/mojtaba142/hotel-booking

Flight prices (structure + base fares) (Use any from below)
Flight Price Prediction (EaseMyTrip—India):
https://www.kaggle.com/datasets/shubhambathwal/flight-price-prediction

Flight Prices (Expedia, 2022 US routes):
https://www.kaggle.com/datasets/dilwong/flightprices

Schedules / ops (optional)
US Flight Delays & Cancellations (2015):
https://www.kaggle.com/datasets/usdot/flight-delays

Airports / routes reference
Global Airports (IATA/ICAO/coords/timezone):
https://www.kaggle.com/datasets/samvelkoch/global-airports-iata-icao-timezone-geo

OpenFlights (airlines, airports, routes):
https://www.kaggle.com/datasets/elmoallistair/airlines-airport-and-routes/

Behavioral (stretch only)
Expedia Hotel Recommendations (competition data):
https://www.kaggle.com/competitions/expedia-hotel-recommendations

How to use the datasets
Deals Agent (backend worker)
From Inside Airbnb pull: listing_id, date, price, availability, amenities, neighbourhood.
Compute avg_30d_price, then flag deals: price ≤ 0.85 × avg_30d; mark Limited availability (<5); add simple tags (e.g., Pet-friendly, Near transit, Breakfast).
From Flight Price Prediction / Expedia keep: origin, dest, airline, stops, duration, price.
Use these as baselines and simulate a **time series** (mean-reverting price + random promo dips −10% to −25% + seats_left scarcity).

From Airports/Routes, join IATA + coords for light location logic and to validate routes.
Persist normalized rows with **SQLModel**;

Concierge Agent (chat-facing)
Build flight+hotel bundles from cached deals.
Compute a small Fit Score (price vs budget/median + amenity/policy match + simple location tag).
Explanations: "Why this" (≤25 words) from facts like price_vs_median, tags, neighborhood; "What to watch" (refund cutoff, limited rooms).
Policy Q&A: quote cancellation/pet/parking snippets from the listing's fields (Inside Airbnb makes this easy).
Expose /bundles (HTTP) and /events (WebSocket). Keep any heavy work off the request path.

Scalability, Performance and Reliability

The hallmark of any good project is scalability. A highly scalable system will not experience degraded performance or excessive resource consumption when managing large numbers of objects or when processing large numbers of simultaneous requests. You need to make sure that your system can handle many listings, users, and incoming requests.

- Pay careful attention to how you manage "expensive" resources like database connections.
- Your system should easily be able to manage 10,000 listings, 10,000 users, and 100,000 reservation/billing records. Consider these numbers as minimum requirements.

For all operations defined above, ensure that if a particular operation fails (for example, a guard allocated two buildings at the same time), your system is left in a consistent state. Consider this requirement carefully as you design your project, as some operations may involve multiple database operations. You may need to make use of transactions to model these operations and roll back the database in case of failure.

Testing

To test the robustness of your system, design a test harness that exercises all the functions a regular client would use. This test harness is typically a command-line or scripted program.
You can use your test harness to evaluate scalability by creating thousands of listings and users. Use it to debug server-side issues before finalizing the GUI.

Other Project Details
Turn in the following on or before the due date. **No late submissions will be accepted!**

- A **title page** listing the members of your group

- A **contributions page** (one short paragraph per member)
- A short (**5 pages max**) write-up describing:
  a) Your **object management** policy
  b) How you handle **heavyweight** resources
  c) The policy you used to decide **when to write** data into the database (and cache invalidation)
- A **screen capture** of your client application showing actual data
- **Output** from your test class (if applicable)
- A **screen capture** showing your database schema
- **Observations and lessons learned** (1 page max)
- Invite **tanyayadavv5@gmail.com**, **smitsaurabh20@gmail.com** to your GitHub repository.

Grading Breakdown

- **40% Basic operation** – Your server implementation will be tested for proper operation of core features. Each passed test is worth some points; failed tests receive 0.
- **10% Scalability and robustness** – I will test with thousands of objects; it should not exhibit sluggish performance or crash. In addition to performance techniques covered in class, you are **required to implement SQL caching using Redis** and show performance analysis.
- **10% Distributed services** – Deploy with Docker into **AWS (Kubernetes/ECS)**. Divide client requirements into **distributed services**. Each service runs on a backend connected by **Kafka**. Divide your data into **MongoDB and MySQL** and provide performance data with justification of results.
- **15% Agentic AI Recommendation Service in FastAPI**
- **10% Analysis report, web/user/item tracking** – Devise your own tracking using logs and explain why it is effective to analyze a web site and user behavior.
- **5% Client** – While server-side is primary, you must provide a usable GUI; more points if your GUI resembles Kayak interactions.
- **10% Test class and project write-up**

Project Presentation

1. Group number and team details
2. **Database Schema**
3. **System Architecture Design Diagram**
4. **Scalability/Performance** comparison bar graphs that show differences by adding features: **B** (Base), **S** (SQL Caching), **K** (Kafka), and additional techniques for **100 simultaneous user threads** (4 total bar charts). You may use **Apache JMeter**.

Combinations:

a. B
b. B + S
c. B + S + K
d. B + S + K + other techniques you used

Note: Populate DB with at least **10,000** random data points and measure performance.

Explain in your report how your detection rules, caching policy, and messaging flows support timeliness and scalability.

Hints

- **Cache entity lookups:** Prevent costly DB trips by caching frequently read entities. Invalidate cache entries when state changes. You are **required** to implement **SQL caching using Redis** and show the impact.
- **Don't write unchanged data** back to the database.
- **Focus first on a complete implementation** — end-to-end correctness is as valuable as performance/scalability.
- **Do not over-optimize.** Complex optimizations often derail project completion.

Exceptions/Failure Modes

Your project **MUST** properly handle the following failure conditions:

- Creating **Duplicate User**
- **Addresses** (for Person) that are not formed properly (see below)
- Any multi-step booking/billing action that partially fails must leave the system in a **consistent** state (use transactions/rollbacks).

Other Notes

State abbreviation parameters

- Limit to valid US state abbreviations or full names. Methods accepting state as a parameter must raise a **malformed_state** exception (or equivalent) if the parameter does not match accepted values. You do not need to handle US territories.

Zip code parameters

- Valid patterns:
  - #####
  - #####-####

- Reject malformed ZIP codes.

Examples of valid Zip codes:

- 95123
- 12
- 95192
- 10293
- 90086-1929

Examples of invalid Zip codes:

- 1247
- 1829A
- 37849-392
- 2374-2384

User IDs are required to match the pattern for US social security numbers:

\[[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9] \]

You may assume that any SSN matching the above pattern is valid (there are some reserve social security numbers starting with 0 that are not valid in real life, but you may consider them to be valid for the purposes of the project). Any method accepting a driver ID is required to raise the 'invalid_driver_id' exception (or equivalent) if the supplied parameter does not match the pattern above.

(If you choose to use an internal “UserID format validation,” document your pattern and validation rule clearly.)

Peer Review

Each team member should write a review about other team members except himself/herself. It is confidential, which means that you should not share your comments with other members. Peer review is submitted separately on Canvas.