

BDM Project 2 Report



Adrian Lim Patricio
Nishant Sushmakar
Oluwanifemi Favour Olajuyigbe

GitHub Repo Link
Universitat Politècnica de Catalunya

June 2025

Contents

1	Introduction	1
2	Architecture Design	2
3	Trusted Zone	3
3.1	Transports Data Preprocessing	3
3.2	LLM Itineraries	4
3.3	Events Preprocessing	4
3.4	User Data	5
3.5	Pipeline Implementation and Tools Used	5
4	Exploitation Zone	7
4.1	Structure of Exploitation Zone	7
4.1.1	Transport Fares	7
4.1.2	Events for Itinerary Generation	7
4.1.3	LLM Itinerary	7
4.1.4	Events for Analysis	8
4.1.5	User Data	8
4.2	Additional Data Assets Generation	8
4.2.1	Events for Analysis	8
4.3	Pipeline Implementation and Tools Used	8
4.3.1	Data Storage in the Exploitation Zone (Visuals)	9
5	Data Consumption Task	13
5.1	Events Analytics Dashboard	13
5.2	Transport Fare Recommendation	15
5.3	Personalised Itinerary Generation	16
5.4	Itinerary Network Analysis	17
5.5	Pipeline Implementation and Tools Used	17

1 Introduction

In Phase 1 (P1), we established the foundational data infrastructure for Tripify, an AI-powered travel planning platform that creates personalized itineraries by integrating transportation options, local events, and LLM-generated recommendations. The platform addresses the challenge of simplifying travel planning by reducing research time while providing optimized, cost-effective, and personalized travel experiences.

Our P1 implementation focused on building a robust data ingestion and storage pipeline that handles three distinct data sources: structured events data from city open databases (Paris, Barcelona, Madrid), semi-structured transport data from web scraping and APIs (flights, trains, buses), and unstructured LLM-generated itinerary content from the Gemini API. We implemented differentiated ingestion strategies tailored to each data type's characteristics, batch processing for relatively static events data and LLM data using Apache Airflow, and real-time streaming for dynamic transport pricing using Apache Kafka.

The P1 architecture established a comprehensive data flow from ingestion to organized storage within Google Cloud Platform. Raw data initially enters a Temporal Landing Zone, where automated lifecycle management is applied. It is then processed by Persistent Loaders orchestrated by Apache Airflow, which transform and move the data into the Persistent Landing Zone. This structured layer ensures data is cleanly organized and readily available for analysis and further processing.

Building on this data foundation, Phase 2 (P2) extends the architecture to implement the trusted and exploitation zones, enabling data transformation, enrichment, and consumption through specialized analytical and application interfaces that deliver Tripify's core value proposition to end users.

2 Architecture Design

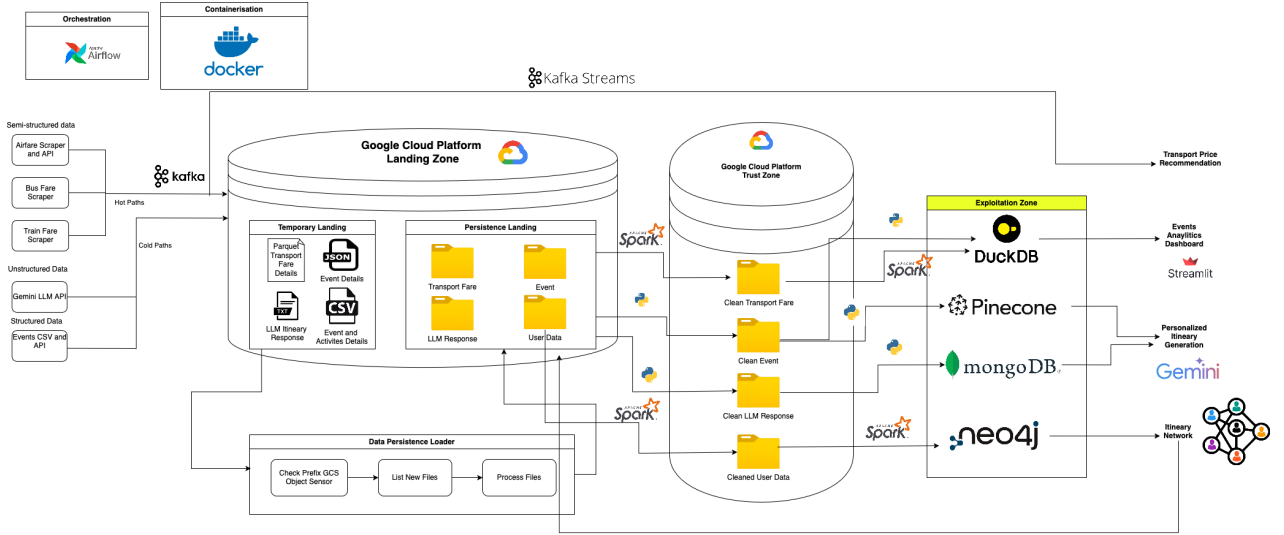


Figure 1: Architecture

The architecture follows a hybrid approach that combines an extended trust zone with specialized systems in the exploitation zone. The trust zone has been extended into Google Cloud, providing a reliable, secure, and flexible environment for rigorous data validation. This is particularly important given the diverse and often inconsistent data arriving in the landing zone from multiple sources. Validating data within the cloud-based trust zone ensures higher quality for downstream processing while maintaining strong security by keeping all data within the same cloud ecosystem. As a travel platform, our domain is well defined, focused on transportation, events, and itineraries. These datasets are typically structured and involve straightforward validation and enrichment tasks, which do not require complex query optimization. Therefore, we do not need specialized databases in the trust zone; cloud storage and general-purpose processing tools are sufficient for our needs. This approach simplifies the architecture without compromising performance, flexibility, or data integrity. To support data orchestration and streaming, we deploy both Apache Airflow and Kafka as self-managed Docker containers on Google Compute Engine instances. This setup provides full control over configuration, resource allocation, and scaling of these critical components. Running Airflow in a containerized environment enables us to customize workflows, manage dependencies efficiently, and maintain consistent deployments. Similarly, deploying Kafka as a self-managed container allows us to fine-tune performance and security settings, ensuring robust and responsive data streaming pipelines that integrate seamlessly with our cloud-based trust and exploitation zones. In addition, we have implemented a robust logging mechanism for auditing and monitoring purposes. All terminal output from Airflow tasks is captured and stored in a des-

ignated folder within the same Google Cloud zone in Google Cloud Storage. This setup ensures that all execution logs are persistently archived for auditability, and access is restricted to maintain security. While these logs are currently retained primarily for auditing, this architecture allows for future integration with real-time monitoring services, enabling issue detection and operational transparency across our data pipelines.

3 Trusted Zone

The Trusted Zone represents the data quality and validation layer where raw data undergoes generic cleaning, normalization, and enrichment processes. This zone ensures data integrity and consistency across all sources through standardized transformation pipelines. Transport data is processed through city-to-country mapping, timezone standardization, and format normalization. Events data undergoes schema correction, text normalization, and multilingual encoding fixes. LLM-generated content is cleaned through text normalization and JSON cleaning. The trusted zone serves as the foundation for reliable downstream analytics and processing.

Transport data, LLM generated itineraries, and events data are stored in a dedicated bucket, each following a structured folder hierarchy. Events data is organized into separate folders by city, and transport data by transport type, with each type containing date-based subdirectories, where the final level holds the corresponding Parquet files. LLM itinerary data is organized first by city, then further divided into Parquet files based on specific days and user interests enabling efficient access and filtering.

3.1 Transports Data Preprocessing

The transformation pipeline applies a series of common and transport-specific transformations to normalize data across buses, trains, and airplanes. Common preprocessing steps include mapping cities to countries using a shared city to country dictionary for major cities such as Barcelona, Madrid, Rome, and Paris. City and station names are normalized to lowercase, and country information is added accordingly. All transport types undergo date and time standardization; departure dates are reformatted, arrival times are computed based on duration, and timestamps are annotated with timezone information (+02:00). Additionally, basic data cleaning is performed by filtering out empty records, exploding nested structures into individual rows, and standardizing station names.

For bus fare data, durations are cleaned by removing textual suffixes and converted into total minutes. Station names are processed to extract clean city names and remove special characters. The company name and price data are retained as provided, with no further transformations applied to the price.

In the case of train fare data, duration strings are cleaned and reformatted into a consistent time format. Extra metadata fields such as travel class, ticket type, and discount are added to enrich the dataset. Company and price fields are preserved from the source data.

For airplane fare data, the pipeline includes sophisticated parsing logic to handle various date and time formats, including AM/PM conversions and padding. Duration values are extracted using regex patterns that handle multiple format variations. Airline names are standardized, including cases involving multiple airlines. Price values are cleaned to retain only valid numeric components and are cast to a numeric type for consistency.

Overall, the transformation pipeline ensures that transport data from different sources is cleaned, enriched, and standardized into a uniform format, enabling seamless downstream processing and integration within the system.

3.2 LLM Itineraries

The text normalization and JSON cleaning pipeline is specifically designed to process and sanitize LLM generated itineraries, ensuring they are clean, readable, and structurally sound for downstream use. It includes robust text normalization, handling Unicode characters and diacritics, enforcing consistent lowercase casing, preserving essential punctuation for readability, and standardizing whitespace. For JSON specific content, such as nested itinerary structures, the pipeline removes markdown artifacts like code block markers, ensures UTF-8 encoding, and recursively cleans all text values without disrupting the original key-value relationships. To ensure reliability, the process includes strong error handling gracefully managing file reading issues, providing helpful debug messages, and returning None on failure to prevent pipeline crashes. Throughout, it preserves the full structure of the JSON, maintaining hierarchy and semantic relationships while delivering clean, normalized content. This makes the LLM generated itinerary data ready for display, analysis, or integration into travel application.

3.3 Events Preprocessing

In the events cleaning pipeline, general-purpose data quality operations were applied to harmonize and prepare the datasets collected for Paris, Madrid, and Barcelona. Common preprocessing steps included text normalization, standardized date formatting, coordinate validation and type conversion, and deduplication. Additionally, all datasets underwent event address standardization, phone number cleaning, and consistent field mapping to create uniform event records suitable for cross-city analysis.

The first step involved schema correction, including column renaming to harmonize naming conventions across cities and enable consistent referencing across all datasets. Data types were explicitly cast to appropriate formats, for example, date-related fields (such as `start_date` and `end_date`)

were converted to datetime objects, while geographical coordinates (latitude, longitude) were cast to float to support geospatial processing. Text fields such as `event_name`, `category`, and `description` were cast to string to ensure consistency.

Text normalization was the most complex task, involving comprehensive cleaning to address multilingual encoding issues. Character replacement dictionaries were created to fix encoding problems, replacing corrupted characters such as “Ã©”, “â€”, and “ÃfÂ©” with the correct accented letters. This process also included HTML tag removal, whitespace normalization using regular expressions, and selective case standardization for user-facing fields such as city names. The pricing field was converted by mapping binary indicators to the strings “free” or “paid”. Address construction was standardized by combining road types, names, and numbers into consistent formats.

Missing fields were systematically filled with NaN values to maintain schema consistency across datasets. The data was then deduplicated after all transformations to ensure data uniqueness and created a unified 31-column schema across all datasets. The cleaned data was stored using UTF-8 with BOM (utf-8-sig) encoding to preserve character integrity.

3.4 User Data

The node and edge data was generated for our graph data model which was implemented using Spark to efficiently process and transform the data at scale. *User* nodes were created from the `users_df`, extracting user-specific attributes such as `user_id`, `name`, and `email`. *City* nodes were generated from a predefined list, with each node representing a city identified by a `name` property. *Activity* nodes were parsed from the `daily_plans` field in the `posts_df`, where each structured activity entry was transformed into a node containing properties like `name` and `description`. *Post* nodes were derived from the `posts_df`, capturing post-related attributes such as `post_id`, `title`, `description`, `start_date`, and `end_date`.

Edges were also constructed in Spark to establish relationships between nodes: **FAVORITE** edges linked users to their favorite cities with a `created_at` timestamp, **CREATED** edges connected users to their posts along with the creation time, and **IN-CITY** edges mapped posts to their respective cities. Additionally, **INCLUDES** edges represented the relationship between posts and activities, annotated with properties such as `day`, `time`, `duration`, and `location`. Finally, **LIKED** edges were used to model the interactions between users and posts they liked, containing `like_id` and `created_at` metadata.

3.5 Pipeline Implementation and Tools Used

For the processing of event and itinerary data, we opted to use standard Python scripts. This decision was based on the relatively low volume and infrequent updates of these datasets, which

do not require the scalability or parallelization benefits provided by distributed frameworks. We leaned towards its simplicity and efficiency for lightweight transformations and validation tasks. In contrast, transport and user data involve substantially larger volumes and more frequent updating, making Apache Spark a more appropriate choice for those pipelines due to its ability to efficiently handle big data at scale. To maximize the efficiency and manageability of our Spark workloads, we utilized Databricks, which offers a fully managed environment for scalable, collaborative, and high-performance data processing with Spark.

The entire process is automated using Apache Airflow, which coordinates the detection, processing, and movement of files between the trusted and landing zones. Each dataset has a dedicated pipeline, ensuring modular, separate and maintainable workflows to process data. As soon as new data arrives, Airflow triggers the appropriate processing tasks to read raw data from the landing zone, perform cleaning, validation, and enrichment, and then write the processed outputs into the trusted zone. This architecture ensures that data is reliably and efficiently transformed and made available for downstream use with minimal manual intervention.

▼	trusted_zone/	⋮
▼	events/	⋮
▶	madrid/	⋮
▶	paris/	⋮
▼	itineraries/	⋮
▶	paris/	⋮
	logs/	⋮
▼	transport/	⋮
▶	busfare/	⋮
▼	users/	⋮
▶	2025-05-26/	⋮

Figure 2: Data Storage in Trusted Zone

4 Exploitation Zone

The Exploitation Zone transforms cleaned data from the trusted zone into optimized, analytics-ready formats stored in specialized databases. This zone implements dimensional modeling for events data using DuckDB, creates vector embeddings for semantic search capabilities via Pinecone, stores itineraries in MongoDB for flexible document-based queries, and maintains normalized transport fare tables for historical analysis. The exploitation zone generates additional analytical assets and serves as the primary data source for all consumption mechanisms.

4.1 Structure of Exploitation Zone

4.1.1 Transport Fares

Transport data has been normalized into multiple structured tables to maintain historical records while minimizing redundancy. This approach was chosen because fare prices fluctuate frequently, making a single, denormalized table inefficient and unsustainable. We use DuckDB for this purpose, as it is optimized for analytical workloads and performs exceptionally well on structured, relational data.

4.1.2 Events for Itinerary Generation

Event data from various cities is transformed into vector embeddings and stored in a Pinecone database. Each event is first converted into a clean, well-formatted text string that encapsulates all essential details. This text serves as input to the embedding model, which generates a corresponding vector representation. Each resulting embedding is assigned a unique identifier (UUID) and enriched with metadata such as the event's name, city, description, and other relevant attributes to provide contextual information. The embeddings, along with their metadata, are then indexed in Pinecone for efficient retrieval and similarity search. Pinecone provides a scalable, high-performance vector database that enables fast similarity search for embeddings ideal for use cases like semantic search, recommendations, and personalized experiences.

4.1.3 LLM Itinerary

We stored each itinerary as a document in a MongoDB collection, including details like the city, type of itinerary (e.g., sightseeing or food), number of days, and the full itinerary text. The city, type, and number of days were automatically extracted from each file's name using a filename parsing method. All files were then read from a directory and inserted into MongoDB in a structured format. To improve query performance, we also created a compound index on the city, type, and days fields. MongoDB was the ideal choice for this task due to its flexible document-based structure, which easily accommodates semi-structured data like itineraries and scales efficiently with growing datasets.

4.1.4 Events for Analysis

The exploitation zone transforms the event data from the different cities into a combined dimensional model designed to serve multiple analytical purposes across diverse business domains. Using a star schema architecture, the zone organizes data into five specialized dimension tables (*dim_location*, *dim_date*, *dim_category*, *dim_organization*, *dim_audience*) surrounding a central *fact_events* table, creating subject-oriented structures that support geographic analysis, temporal patterns, organizational insights, and audience segmentation simultaneously. The *dim_date* table, generated dynamically, enables rich time-based analysis by including attributes like day of week, season, and quarter. This model enables efficient slicing, aggregation, and time-series analysis while eliminating redundancy and improving maintainability. To ensure reliability, logging and error handling mechanisms were implemented, and all tables were stored in DuckDB for high-performance querying. DuckDB was selected as the technology due to its high-performance columnar storage, in-process architecture, and native Python compatibility, all of which make it excel at complex joins and aggregations typical of analytical workloads.

4.1.5 User Data

In this stage, nodes and edges data are uploaded to **Neo4j Aura**, our managed graph database service. The import process creates nodes for *users*, *posts*, *cities*, and *activities*, and establishes relationships such as **POSTED** (user to post), **LIKED** (user to post), and **IN_CITY** (post to city). This graph structure enables complex queries and analytics that support application features like itinerary recommendations and social networking.

4.2 Additional Data Assets Generation

4.2.1 Events for Analysis

In addition to the dimensional model, fifteen specialized analytical assets were generated to extract actionable insights from the model. These include KPI summary tables (*event_summary*, *free_event_distribution*), temporal analysis tables (*seasonal_events*, *monthly_event_patterns*), geographic insights (*events_by_district*, *event_density_map*), and operational views (*upcoming_week_events*, *weekend_events*, *kid_friendly_events*). A flattened view (*event_details_view*) was also developed to simplify access for dashboards and analytics tools.

4.3 Pipeline Implementation and Tools Used

Similarly to the previous pipeline, the transfer of data from the trusted zone to the exploitation zone is orchestrated using Apache Airflow, which automates the detection, processing, and movement of datasets as they become available. In this stage, the processing tasks are often more complex,

as the data must be transformed, aggregated, or restructured to fit the requirements of specialized storage systems such as DuckDB, Pinecone, MongoDB, or Neo4j.

For high-volume datasets like transport and user data, we again leveraged Apache Spark on Databricks to efficiently handle the heavier processing and ensure seamless integration with these target databases. Each dataset is managed through a dedicated pipeline, maintaining modularity and clear separation of concerns. Once the necessary transformations and optimizations are applied, the processed data is loaded into the appropriate exploitation zone storage, making it readily accessible for downstream analytical and application use cases. This approach ensures that all data assets are not only reliably transferred but also optimally prepared for advanced querying, search, and analysis, while minimizing manual intervention throughout the process.

4.3.1 Data Storage in the Exploitation Zone (Visuals)

GCP

This only contains logs for exploitation data creation. The Users directory contains the Neo4J CSV files following the IMPORT format. Transport and events contain DuckDB files for analytics purposes.



Figure 3: GCP

DUCKDB

The data follows star schema for analytics purposes.

Filter by name prefix only ▼				Filter objects and folders	
<input type="checkbox"/>	Name	Size	Type		
<input type="checkbox"/>	events_exploitation.duckdb	12.3 KB	application/octet-stream		

Figure 4: DUCKDB (Events)


Filter by name prefix only ▼		Filter	Filter objects and folders
<input type="checkbox"/>	Name	Size	Type
<input type="checkbox"/>	 transport_data.duckdb	12.3 KB	application/octet-stream

Figure 5: DUCKDB (Transport)

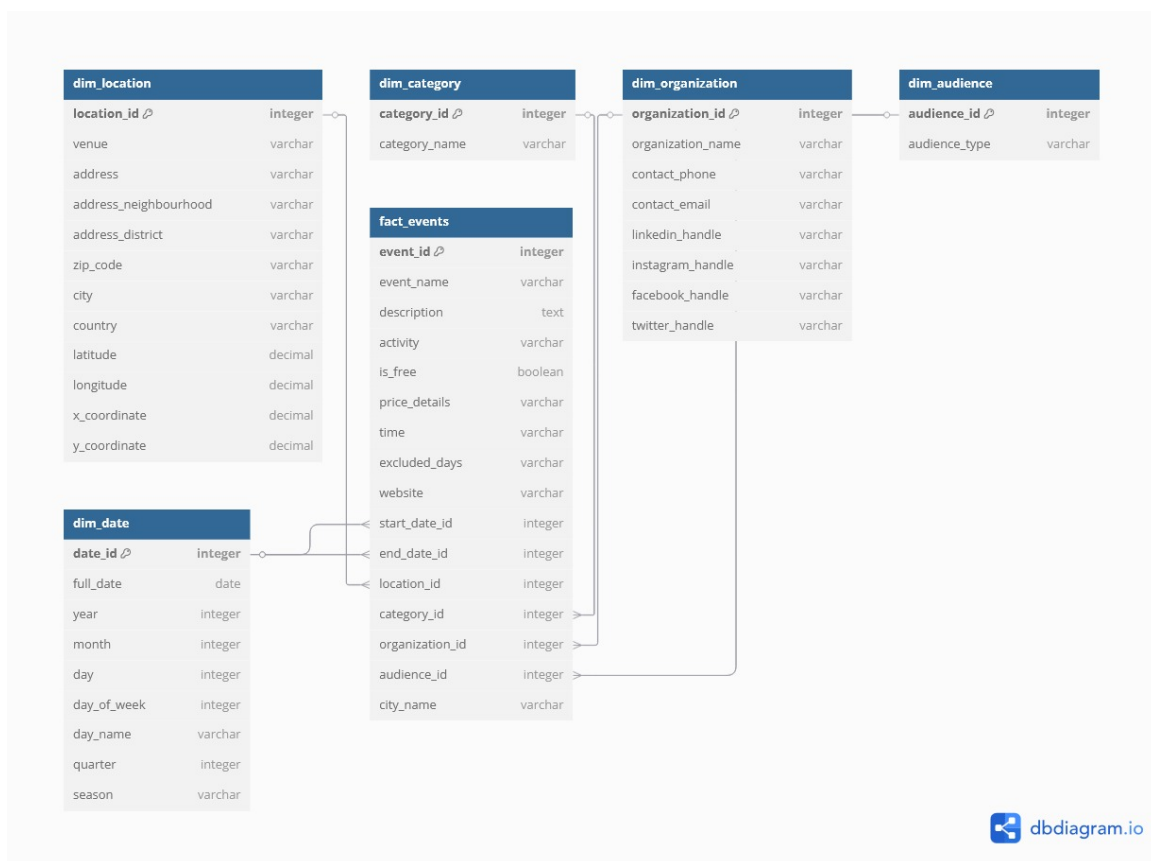


Figure 6: DUCKDB (Events Schema)

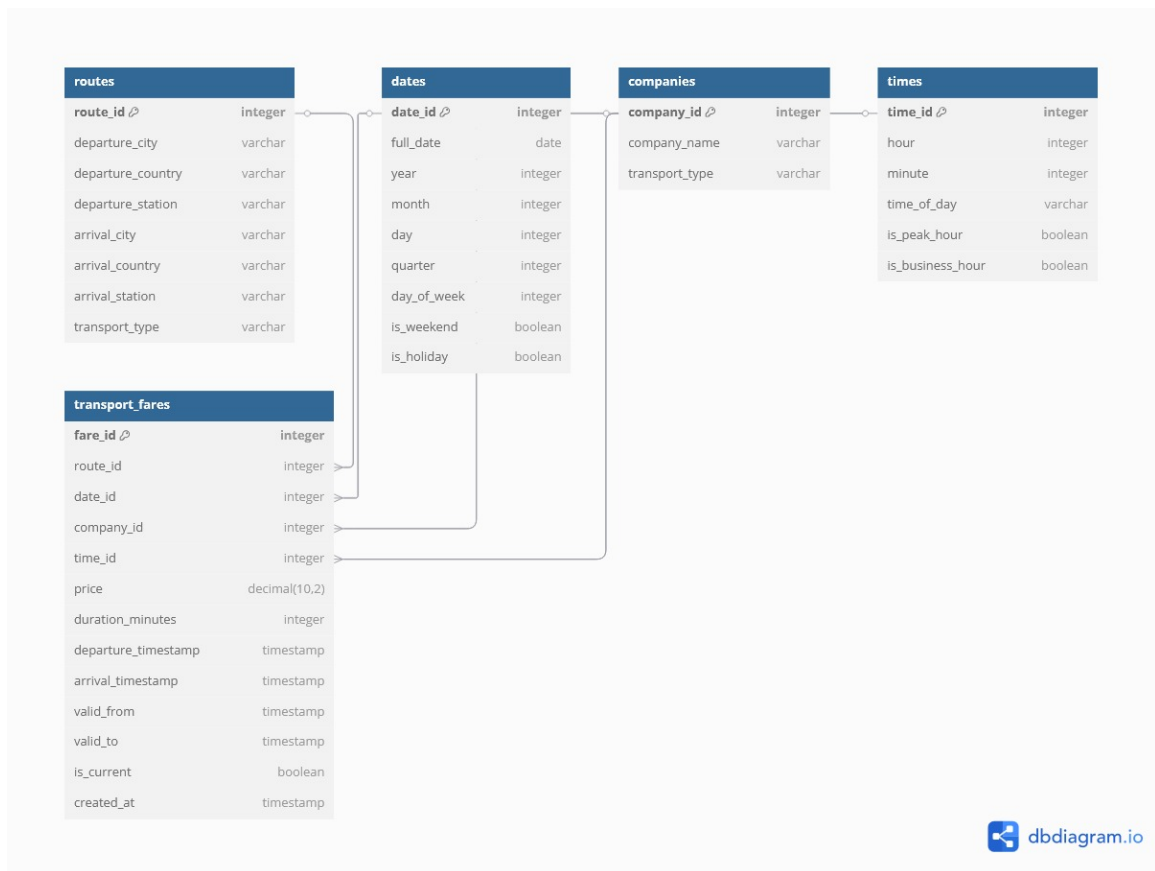


Figure 7: DUCKDB (Transport Schema)

PINECONE

The screenshot displays the Pinecone web interface for Tripify's Org. The left sidebar shows navigation options: Get started, Database, Indexes (1), Backups, Assistant, Inference, API keys, and Manage. The main panel shows search results for a specific ID.

Search Interface:

- Namespace: `__default__`
- Search by: ID
- ID: `00165377-3ec4-4336-98b9-3e4911124448`
- Top K: 10
- Buttons: Filter, Rerank, Search

Search Results (Showing 10 hits):

- ID:** 00165377-3ec4-4336-98b9-3e4911124448
activity: ""
address: "C Portillat 11"
audience: ""
category: ""
city: "barcelona"
description: ""
event_name: "Projecció 'Els Buits' + Col·loqui"
SCORE: 1.0000
[Show 6 more](#)
- ID:** 664c8fc4-8b38-482f-9307-0bd05f73ea1f
activity: ""
address: "Carrer del Comerç 36"
audience: ""
category: ""
SCORE: 0.9015

STARTER USAGE:

- WUs: 0 / 2M
- RUs: 12 / 1M
- Storage: 0.024 / 2GB
- Upgrade now

Figure 8: Pinecone

MONGODB

itinerary_db.itineraries

STORAGE SIZE: 208KB LOGICAL DATA SIZE: 162.82KB TOTAL DOCUMENTS: 46 INDEXES TOTAL SIZE: 72KB

[Find](#) [Indexes](#) [Schema Anti-Patterns](#) ⁰ [Aggregation](#) [Search Indexes](#)

[Generate queries from natural language in Compass](#)

[INSERT DOCUMENT](#)

Filter [🔗] Type a query: { field: 'value' } [Reset](#) [Apply](#) [Options](#) ▶

QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('68342c2c5fd87bc3634b92bf')
city: "paris"
type: "Food_Exploration"
days: 2
itinerary: Array (21)
  0: Object
    location: "le marais"
    time: "morning"
    text: "start your parisian food journey in the charming le marais district. b_"
    day_number: "2"
    user_persona: "Food_Exploration"
  1: Object
  2: Object
```

Figure 9: MongoDB

NEO4J

Nodes (19,674)

* [Activity](#) [City](#) [Post](#) [User](#)

Relationships (60,145)

* [CREATED](#) [FAVORITE](#) [IN](#) [INCLUDES](#)

[LIKED](#)

Property keys

[bio](#) [comments_count](#) [created_at](#) [day](#)

[description](#) [duration](#) [email](#) [end_date](#) [id](#)

[join_date](#) [like_id](#) [likes_count](#) [location](#)

[name](#) [post_id](#) [profile_picture](#) [start_date](#)

[time](#) [title](#) [total_cost](#)

[Show all \(2 more\)](#)

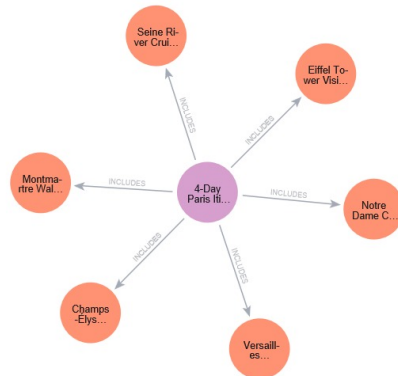


Figure 10: NEO4J Schema

5 Data Consumption Task

The Consumption Zone represents the final layer where processed data from the exploitation zone is transformed into user-facing applications and analytical interfaces. This zone includes interactive Streamlit dashboards for events analytics, real-time transport fare displays via Kafka streams, and personalized itinerary generation through RAG pipelines. The consumption zone ensures that all the processed data intelligence is accessible through intuitive interfaces that deliver value directly to end users.

5.1 Events Analytics Dashboard

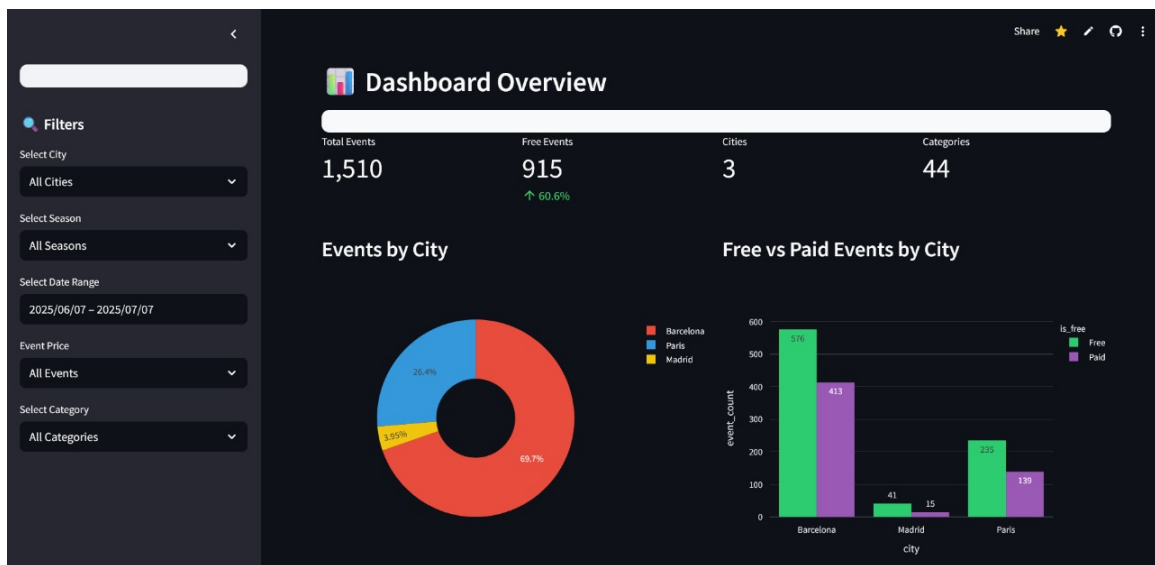


Figure 11: Screenshot of Dashboard

You can check out the dashboard here: <https://tripify-dashboard.streamlit.app/>.

Note: It might take a little while to load, so please be patient as the app can be a bit slow initially.

For Events, the consumption layer is implemented using Streamlit, a Python-based web application framework, which enables rapid development of interactive dashboards without requiring extensive frontend development.

Data is accessed from the exploitation zone and application's visualizations are developed using Plotly and Folium. Plotly is used to create interactive statistical graphics, including bar charts and pie charts, allowing users to explore KPIs, trends, and comparisons. For geospatial analysis, Folium is used to render interactive maps that display event locations with features like clustering and density mapping. Together, these tools create a comprehensive visual analytics environment that supports both statistical and geographic analysis.

To enhance performance, the application leverages Streamlit's caching mechanism. This allows frequently accessed or computationally expensive queries to be stored and reused, significantly reducing response times during user interactions such as filtering and navigation. Streamlit was selected not only for its performance and flexibility but also because of its strong compatibility with the Python data ecosystem, making it well-suited for quick iterations and real-time analytical applications.

The application features five distinct analytical interfaces: a comprehensive dashboard with KPI metrics and comparative visualizations, an interactive geospatial map showing event clustering and density patterns, an advanced search interface with pagination and multiple sorting options, a calendar view providing day-specific event scheduling, and documentation. Users can also filter data in all the interfaces by city, season, date range, price level, and event category, with all results updating in real time across the different views, allowing for highly personalized and focused data exploration.

Beyond basic filtering, the application includes a range of advanced analytical capabilities. These include dynamic visual comparisons between cities using pie charts and stacked bar graphs, analysis of seasonal and weekday patterns, and neighborhood-level insights based on the frequency of events. The geographic intelligence layer uses latitude and longitude coordinates to enable precise event mapping, while radius-based clustering highlights areas with dense event activity, giving users an accurate picture of event distribution and hotspots within a city.

The dashboard can be found here: <https://tripify-dashboard.streamlit.app/>.

Note: It might take a little while to load, so please be patient as the app can be a bit slow initially.

5.2 Transport Fare Recommendation

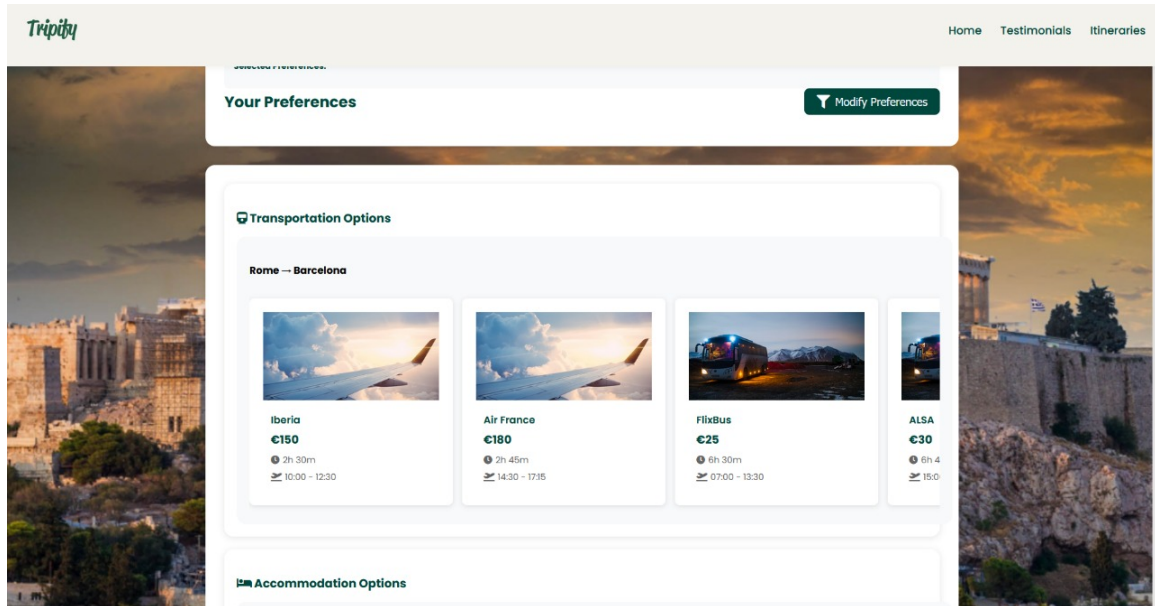


Figure 12: Screenshot of Transport Recommendation

Transport fare data is ingested in real time via a Kafka stream and is directly used to display up-to-date fares to users. Additionally, a fare recommendation system operates on this data, suggesting options based on key criteria such as the cheapest fare, the fastest route, and the most convenient mode of transport. This real time integration ensures that users receive timely and relevant suggestions tailored to their preferences.

Although the transport fares database which has been created in duckdb is not actively used at the moment, it has been designed with future scalability and analytical capabilities in mind. As the company evolves and begins capturing user interactions and purchase data, this schema will enable advanced use cases such as fare trend analysis, user behavior insights, and route popularity scoring. By investing early in a flexible and extensible data model, we ensure the foundation is in place to support future data-driven strategies and decision-making.

5.3 Personalised Itinerary Generation

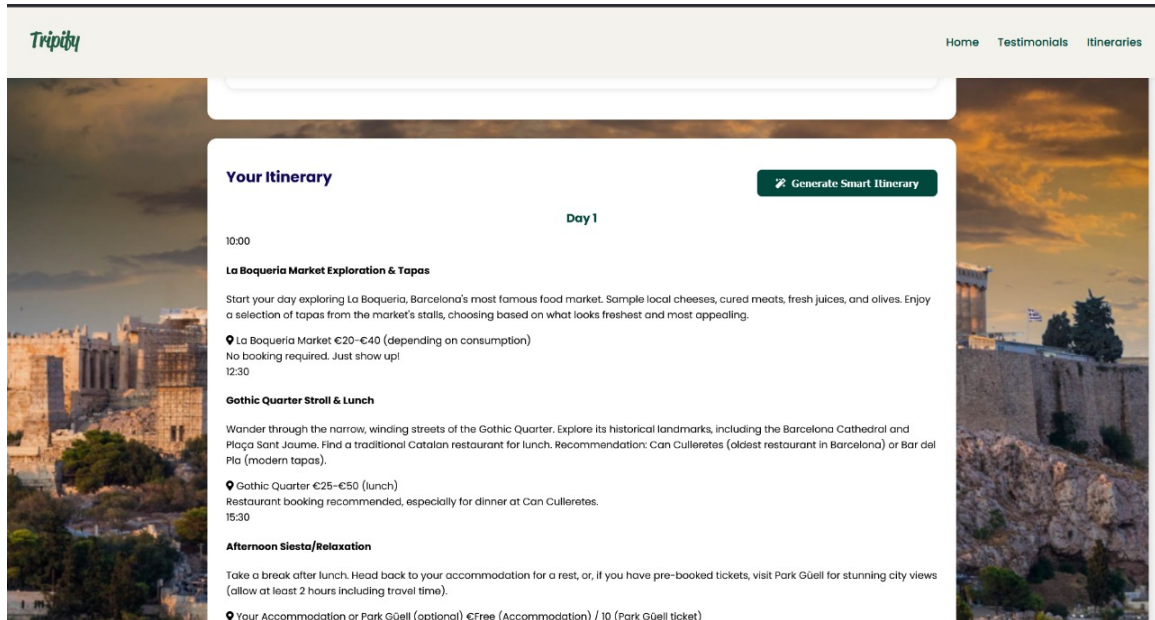


Figure 13: Screenshot of Personalised Itinerary

The events data embeddings are retrieved from Pinecone, while LLM-generated itineraries are fetched from MongoDB. Together, they enable the generation of personalized city itineraries tailored to multiple user interests. This integration is one of the core unique selling points of our product. The stored knowledge serves as contextual input in our Retrieval Augmented Generation pipeline, which queries the Gemini API to generate the final itinerary. Additionally, the pre-generated itineraries act as a fallback mechanism, ensuring continuity in case of a failure in the primary generation process.

In addition to the client-side features, we also leverage graph data on the backend to analyze user activity patterns at scale. By examining the interconnected relationships between users, itineraries, and interactions, we can uncover valuable insights that support strategic business decisions.

5.4 Itinerary Network Analysis

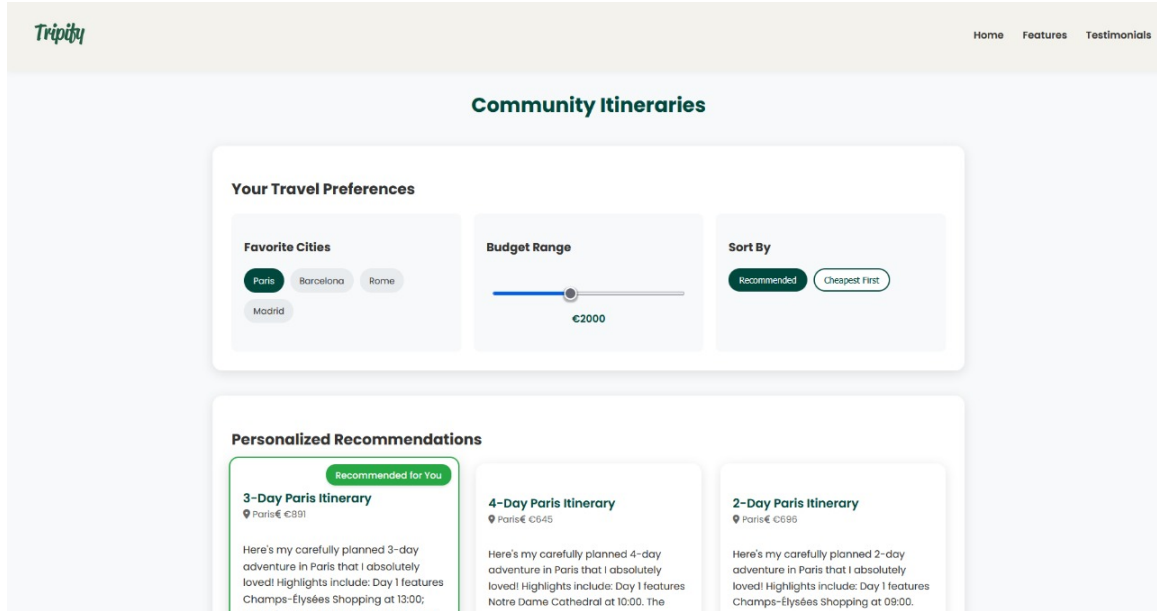


Figure 14: Screenshot of Social Network for Itinerary

Graph Network is used for personalized travel recommendations and create a social community within the app. By modeling users, itineraries, and interactions in neo4j (such as likes, comments, and favorites) as interconnected nodes and edges in a graph, we enable dynamic filtering and recommendation features. This approach allows users to efficiently discover relevant itineraries filtered by user persona, city, starred, or commented on status and receive suggestions for similar trips based on their activity, recent bookings, and preferences. The project benefits by delivering highly relevant, community driven recommendations, increasing user engagement, and streamlining the trip planning process for our user base.

5.5 Pipeline Implementation and Tools Used

Once data is available in the exploitation zone, it is accessed by various consumption mechanisms tailored to specific application needs. DuckDB is stored as a single database file in Google Cloud Storage (GCS). By keeping the DuckDB file in GCS, we enable seamless, centralized access for our Streamlit dashboards and other consumption tools, without the need to manage dedicated database servers or complex infrastructure. This approach is particularly efficient because DuckDB is an in-process analytical database, where users and applications can directly attach to the database file in GCS (in read-only mode) and run SQL queries with minimal setup, leveraging DuckDB's high performance for analytical workloads. Compared to traditional client-server databases, this file-based model reduces operational overhead, simplifies deployment, and lowers costs, while still

providing fast, scalable analytics on cloud-hosted data.

Personalized itinerary generation services use MongoDB and Pinecone to deliver user-specific and vector-based recommendations. Both are deployed in dedicated, fully-managed, serverless database services on their own infrastructure (Pinecone Server and MongoDB Atlas), which aligns well with our needs for scalability, security, and simplicity. From the frontend, these services are accessed directly using secure API keys and connection strings provided by the database services. The serverless model is particularly advantageous for our use case because it automatically handles scaling, maintenance, and high availability, allowing our application to efficiently manage variable workloads without manual intervention or over-provisioning. This setup ensures that our backend remains responsive and cost-effective, even as user demand fluctuates, while also simplifying deployment and reducing the operational burden on our team.

For network-based analyses, itinerary data is accessed in Neo4j through graph queries. Dedicated APIs are developed for exploitation tasks, enabling backend services and user-facing applications to fetch the necessary data on demand. Similar to the other databases, the Neo4j fully-managed, serverless database service was utilized.

A Kafka stream is implemented to support the real-time processing and delivery of transport information, ensuring that any updates or changes can be immediately reflected within the application interface. An API is built on top of this Kafka stream, allowing our frontend application to subscribe and consume the latest transport data. This architecture enables consumption tools to query the exploitation databases in real-time or near-real-time without redundant data movement.

Apache Airflow is used to automate periodic refreshes or trigger updates in the exploitation databases, ensuring that all consumption layers have access to the most current and relevant data.