Platform for Personalized Itinerary Planning in Andalusian Villages

Master's Thesis Dissertation

September 2025

Author

Christian Berdejo Sánchez
cberdejo2205@gmail.com

Advisor

Antonio Benítez Hidalgo

Master's in Big Data, Artificial Intelligence and Data Engineering

Abstract

In today's data-driven tourism landscape, delivering personalized travel recommendations is key to enhancing user experience and preserving regional heritage. This project tackles that challenge by focusing on intelligent algorithms that combine semantic similarity and route optimization to create tailored itineraries across Andalusia.

The system collects rich cultural and touristic information through automated data extraction, transforms it into high-dimensional semantic embeddings, and applies ranking algorithms that adapt to each traveler's interests and accessibility constraints. By leveraging cosine similarity over embeddings generated with Sentence Transformers, the platform identifies the most relevant municipalities and cultural assets for each user profile. Route optimization algorithms then arrange these recommendations into efficient and engaging itineraries.

Cultural depth is enriched by integrating historical narratives, intangible heritage, and architectural landmarks into the recommendation logic, allowing the algorithm to go beyond simple geographic matching. This data-driven approach not only highlights the diversity of Andalusian culture but also demonstrates how natural language processing, geospatial computation, and optimization techniques can be combined into a single intelligent tourism recommendation engine.

Index Terms

Tourism, Itinerary, Embeddings, Semantic Search, Data Extraction Pipeline, Scraping, ORM, Database, Backend, Frontend, Database, Sentence Transformers, Cultural Heritage

I. Introduction

Andalusia offers a rich cultural and historical legacy that remains underrepresented in mainstream travel applications. Promoting this heritage through personalized, accessible and user-friendly digital tools can boost local tourism and generate social and economic value.The growing interest in tourism has made technology a fundamental tool for enhancing trip planning and destination discovery. In particular, personalizing tourist itineraries through intelligent digital tools can significantly enrich the travel experience, increase user satisfaction, and promote lesser-known regions [1]. This project proposes the development of an intelligent web platform capable of generating personalized travel plans based on cultural heritage, historical monuments, and user preferences.

A. Motivation

This work addresses the problem of discovering and planning visits to municipalities beyond mainstream destinations by leveraging semantic understanding of user preferences and local descriptions, and travel-time constraints.

The goal is to design and implement a comprehensive system capable of managing information across its entire life cycle, from initial acquisition to final presentation to end users. The system is designed to integrate heterogeneous data sources, ensuring proper organization, consistency, and effective use for analysis and decision support. Recommendations will be delivered as routes through an accessible, user-friendly interface. Beyond functional performance, the proposal highlights reproducibility, long-term maintainability, and the ability to adapt to emerging requirements. To achieve this, the system leverages open-source technologies, which ensure transparency, adaptability, and long-term sustainability.

B. Objectives

- **End-to-end stack:** Design and implement a modular scraper, back-end, and front-end.
- **Semantic ranking:** Use sentence embeddings and cosine similarity for town recommendation.
- **Routing:** Compute isochrones and optimal ordering subject to travel-time constraints.
- **Interactive UI:** Map-based exploration and PDF export of itineraries.
- **Automation & reproducibility:** Containerize services and orchestrate data pipelines.
- **Quality assurance:** Provide unit/integration tests for critical logic.

C. Document Organization

The document is structured into the following chapters:

- **Introduction**: This section outlines the motivation behind the project.
- **State of the Art**: A detailed review of existing technologies and solutions in the field of tourism recommendation systems.
- **Method**: Description of the computational techniques used, such as web scraping, embedding generation, and semantic similarity.
- **Experiments**: Implementation process, experiments performed, and evaluation metrics used to assess the system.
- **Results**: Analysis and discussion of the performance and functionality of the final platform.
- **Conclusions**: Summary of achievements, challenges encountered, and future lines of work.
- **Appendices**: Installation and user manuals.
- **References**: Cited literature and sources of data used throughout the project.

II. State of the Art

A. Semantic Understanding

Modern recommendation systems rely heavily on **semantic embeddings** technology to process natural language descriptions and transform them into dense vector representations that capture meaning beyond mere keywords. The field has seen significant advancement with transformer-based architectures like BERT, which have revolutionized how we understand and process textual content about destinations, cultural highlights, and historical significance.

Several approaches exist for generating embeddings: **word-level embeddings** (like *Word2Vec* and *GloVe*) focus on individual word representations, **sentence-level embeddings** (such as those generated by *Universal Sentence Encoder* and *Sentence-BERT*) capture semantic meaning at the sentence level, and **document-level embeddings** (like *Doc2Vec*) work with entire documents.

Traditional word embedding models such as Word2Vec [2] generate a single, fixed vector representation for each word in the vocabulary, regardless of the surrounding context. For example, the word gymnasium will always have the same vector, whether it appears in the sentence "I exercise in a gymnasium" or in "A gymnasium is a German school". This limitation fails to capture the subtle, context-dependent meanings of words in natural language (Figure 1).

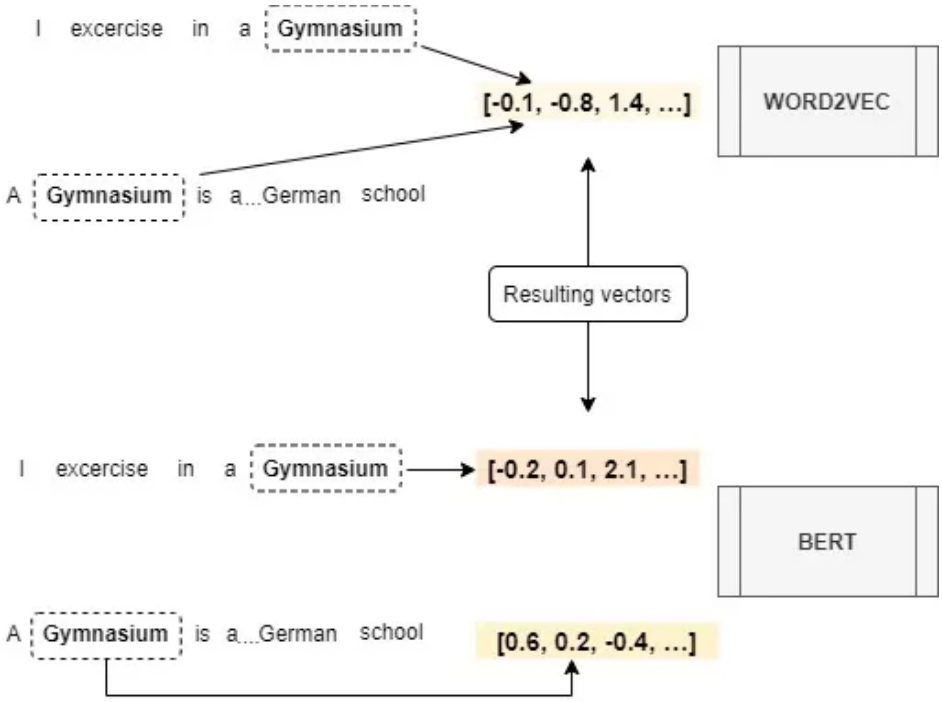

Fig. 1. Word2Vec vs BERT: same word, different vector.

To overcome this, contextualized language models such as Bidirectional Encoder Representations from Transformers (BERT) [3] have been developed. Unlike Word2Vec, BERT generates dynamic, context-sensitive embeddings for each word. For instance, the representation of the word bank will differ depending on whether the sentence refers to finance or to a riverbank. This is made possible by the Transformer architecture [4], which employs self-attention mechanisms to capture relationships between all words in a sequence simultaneously, independent of their position.

Since its introduction by the Google AI Language team in 2018 [3], BERT has become a cornerstone of modern NLP pipelines. Unlike embedding-only models, BERT not only produces embeddings but can also be fine-tuned for a variety of downstream tasks, such as question answering, text classification, and entity recognition.

B. Route Optimization in Tourism Systems

Intelligent travel systems require robust route optimization capabilities to generate efficient journeys. The field offers several approaches: **open-source routing engines** like *OSRM* [6] and *Valhalla* [7] provide comprehensive routing capabilities with different performance characteristics, while **commercial solutions** like *Google Maps API* [8] and *Mapbox* [9] offer high-quality data with usage-based pricing models.

Key functionalities in modern routing systems include: **time-distance matrix computation** for calculating travel times between multiple points, **isochrone generation** for visualizing areas reachable within specific time constraints, **real-time traffic integration** for dynamic route adjustments, and **multi-modal routing** supporting different transportation methods.

C. Front-End Technologies for Interactive Maps

Interactive mapping applications have become essential for tourism platforms. The landscape includes **lightweight libraries** like *Leaflet.js* [10] that prioritize simplicity and performance, **advanced visualization frameworks** like *Mapbox GL JS* and *OpenLayers* [11] that offer sophisticated rendering capabilities, and **comprehensive mapping suites** that integrate with commercial map providers.

Component-based front-end frameworks have gained prominence, with *React* [12] leading in market adoption, *Vue.js* [13] offering progressive enhancement capabilities, and *Svelte* [14] providing compile-time optimizations. These frameworks enable responsive, modular interfaces that adapt seamlessly across devices.

D. Data Acquisition Challenges

Modern tourism websites present unique challenges for data extraction, primarily due to their dynamic, JavaScript-heavy interfaces. Traditional static scraping approaches, based on HTML parsing tools such as BeautifulSoup [18] and Scrapy [19], are effective for simple, static pages but struggle with complex, interactive web applications.

This distinction gives rise to two categories: static scraping for traditional HTML content and dynamic scraping for JavaScript-rendered content. To address the latter, browser automation tools such as Selenium [15], Playwright [16], and Puppeteer [17] have been developed, enabling automated interaction with modern websites.

As data pipelines grow in scale and complexity, workflow orchestration becomes crucial. Solutions like Apache Airflow [20], Prefect [21], and Luigi [22] provide robust management for end-to-end ETL processes, ensuring reliability and scalability.

E. Back-end Infrastructure Patterns

Modern web applications favor **asynchronous architectures** that can handle high concurrency efficiently. The ecosystem includes **lightweight frameworks** like *FastAPI* [23] and *Flask* [24] for *Python* [25], **full-featured solutions** like *Django* [26] and *Spring Boot* [27] for comprehensive applications, and **high-performance alternatives** like *Express.js* [28] for *Node.js* [29] environments.

Database interaction patterns have evolved toward **Object-Relational Mapping (ORM)** solutions that provide abstraction layers between application code and database systems. These range from **lightweight ORMs** with minimal overhead to **comprehensive solutions** offering advanced features like automatic migrations and query optimization.

III. Method

The project was designed with two system actors in mind, each associated with a specific use case. The first actor is the **end user**, whose main use case is to generate a personalized travel itinerary by submitting preferences through a form. The second actor is the **data operator**, whose use case involves executing the data pipeline, monitoring its performance, and reviewing its output and reports.

1) Functional Requirements:

- **FR1:** End users shall be allowed to fill out a form specifying their travel preferences.
- **FR2:** The system shall generate a personalized travel itinerary based on the user's input.
- **FR3:** The system shall expose a REST API endpoint (`POST /api/v1/itinerary`) to receive form data and return ranked municipalities.
- **FR4:** The system shall calculate driving times and generate optimized travel routes between selected towns.
- **FR5:** End users shall be allowed to visualize suggested routes and towns on an interactive map.
- **FR6:** End users shall be allowed to download a PDF report containing itinerary details, maps, and descriptions.
- **FR7:** The system shall collect tourism and cultural data from official sources such as Junta de Andalucía, Wikipedia, IAPH, and Andalucia.org.
- **FR8:** The system shall store structured data in a persistent relational database.
- **FR9:** Data operators shall be allowed to run and monitor the data ingestion pipeline.
- **FR10:** The system shall display images, descriptions, history, monuments, and cultural events for each suggested municipality.

2) Non-Functional Requirements:

- **NFR1:** The system architecture shall follow a modular structure, separating concerns into `scraper`, `back-end`, `front-end`, and `utils-project` modules.
- **NFR2:** The front-end shall persist the generated itinerary locally so that it remains available after a page refresh.
- **NFR3:** The back-end shall be fully asynchronous to ensure efficient data handling and non-blocking operations.
- **NFR4:** All modules and services shall be containerized and deployable via *Docker* [30] with minimal setup.
- **NFR5:** The code base shall include unit tests to validate the correctness of critical functionality and ensure maintainability.

3) Sequence Diagram: To facilitate understanding of how the application's functions interact with system components, two sequence diagrams have been created. These diagrams highlight the critical requirements related to user–system interactions, as shown in Figures 2 and 3.

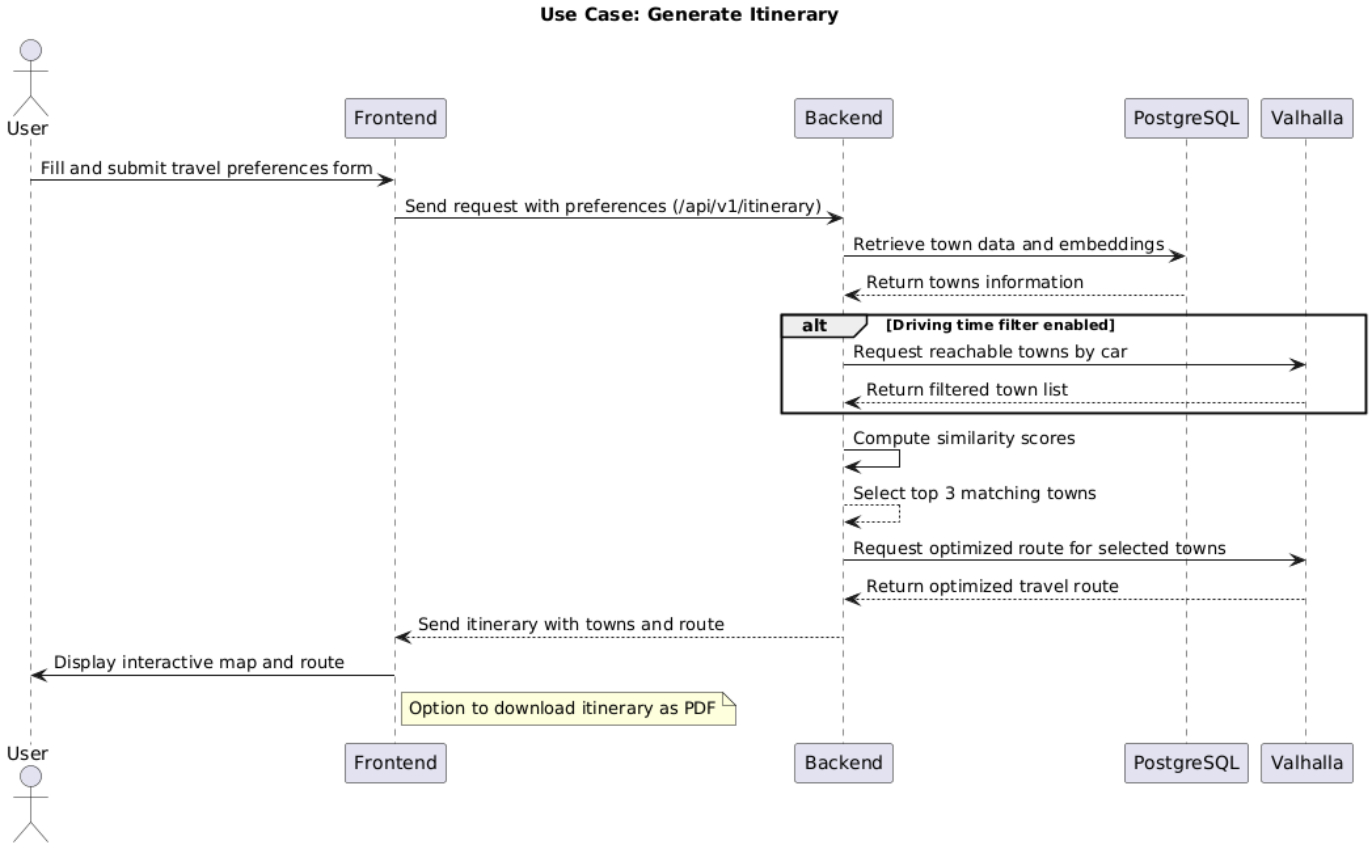

Fig. 2. Sequence diagram illustrating the workflow for generating a travel itinerary.

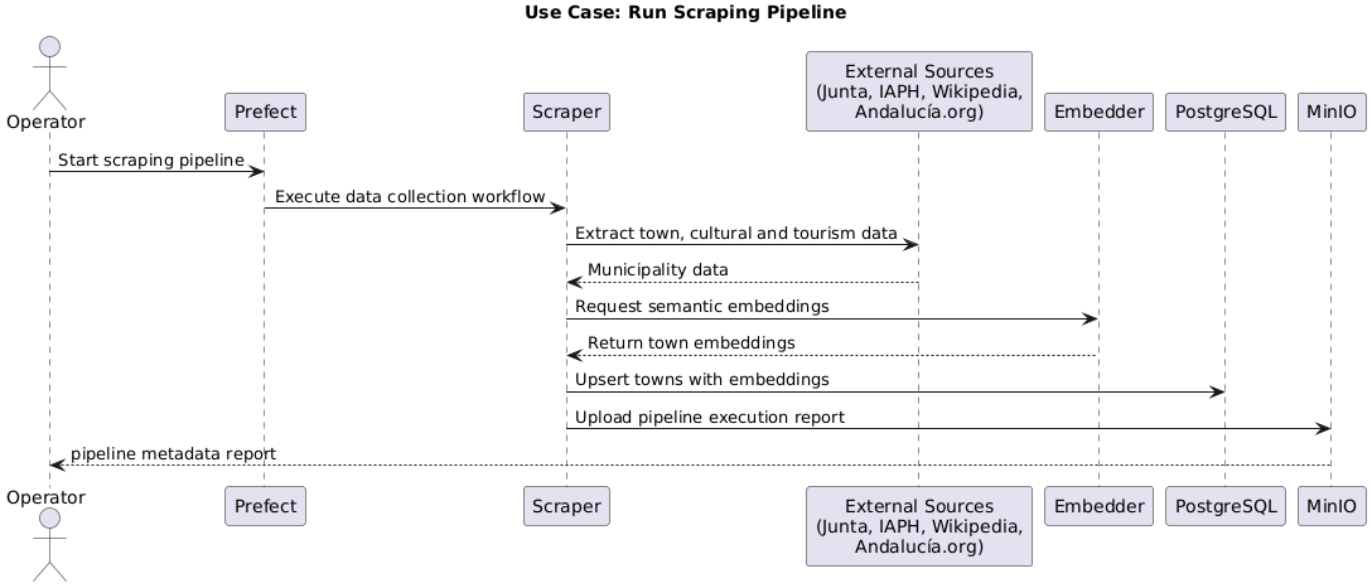

Fig. 3. Sequence diagram illustrating the scraping pipeline from multiple official sources.

A. Technologies and Tools Used

The project is structured as a modular architecture composed of four interconnected components: the *scraper*, *back-end*, *front-end*, and a shared *utilities* package. All modules, except for the *front-end*, are implemented entirely in *Python*, chosen for its versatility, readability, and strong ecosystem in data engineering and machine learning applications.

1) Semantic Processing Technology Selection: For semantic understanding, ***SentenceTransformers*** [31] was selected over alternatives like *spaCy* [32] or *Gensim* [33]. While *Gensim* primarily focuses on word embedding models that work best for short documents through word embedding pooling, and *spaCy*'s transformers require GPU resources for effective deployment, *SentenceTransformers* provides sentence-level embeddings using state-of-the-art transformer architectures like ***BERT***, delivering superior semantic accuracy for our tourism content analysis.

The decision to use local embeddings over cloud-based solutions such as *Google Universal Sentence Encoder* [34] or *OpenAI Embeddings* [35] reflects a strategic choice for independence and reproducibility. While cloud services offer exceptional quality, the project prioritizes offline capability and eliminates external API dependencies, crucial factors for consistent performance and cost control.

2) Route Optimization Engine Selection: ***Valhalla*** was selected as the routing engine over alternatives like *OSRM* due to specific project requirements. While *OSRM* excels at high-performance matrix queries on continental scales, *Valhalla* offers superior support for dynamic requests and time-aware routing. In contrast to *OSRM*, *Valhalla* does not require the entire graph to be stored in memory, making it suitable for memory-constrained environments while still allowing parameter variation at runtime. Furthermore, its native support for isochrone generation, visualizing areas reachable within specific time constraints, makes it particularly valuable for filtering destinations within accessible zones.

3) Front-End Architecture Decisions: The user interface leverages ***Leaflet.js*** for interactive map rendering, chosen for its lightweight nature and seamless integration with open data sources. Although alternatives like *Mapbox GL JS* and *OpenLayers* offer advanced visualization features, *Leaflet.js* provides the optimal balance of functionality and simplicity for this project's requirements.

React forms the declarative foundation of the interface, selected not only for its component reusability and modern development paradigm but also for its strong market presence, a practical consideration for long-term maintainability and developer availability. Styled with ***Tailwind CSS*** [36], the result is a responsive, modular interface that adapts seamlessly across devices. Alternatives such as *Vue.js* or *Svelte* were considered but discarded due to *React*'s market dominance and extensive ecosystem.

The *front-end* integrates ***html2canvas*** and ***jsPDF*** [46], [47] to allow PDF generation from the itinerary view, and uses *Vite* [48] for fast development builds.

4) Data Acquisition Strategy: The *scraper* uses ***Selenium*** for dynamic web pages and ***BeautifulSoup*** for static HTML parsing. *Selenium* was chosen over *Scrapy* or *Playwright* because, even though *Scrapy* offers superior performance for rule-based crawling and *Playwright* provides modern automation capabilities, *Selenium*'s mature ecosystem and broad compatibility make it ideal for mixed content scenarios encountered in tourism websites.

Prefect orchestrates the entire ETL workflow due to its lightweight approach to monitoring and retry logic, providing significant advantages over heavier alternatives like *Apache Airflow* or *Luigi*, especially in *Python*-centric projects.

5) Back-End Infrastructure Choices: ***FastAPI*** serves as the backbone of the server architecture, enabling rapid development of asynchronous APIs with automatic documentation generation. Its performance advantages and native asynchronous support make it particularly well suited for the embedding and routing calculations that form the core of the recommendation engine. It was chosen over *Flask* due to better performance and native support for asynchronous operations.

Database interactions are managed through *SQLModel* [37], a modern ORM built on top of *SQLAlchemy* [38] and *Pydantic* [45]. This approach simplifies the mapping between *Python* objects and SQL tables, while also allowing type validation and asynchronous queries.

The ***scikit-learn*** [44] library is used for cosine similarity computation between user preferences and municipal embeddings, ensuring efficient ranking in recommendation tasks.

6) Development and Deployment Infrastructure: ***Docker*** ensures consistency across development and production environments, isolating dependencies and configurations for reproducible builds. The choice of ***uv*** [39] as the *Python* dependency manager reflects its superior speed and lockfile support, particularly valuable in containerized workflows.

7) Data Storage Architecture: Structured data is stored in ***PostgreSQL*** [41], chosen for its performance, compatibility with ORMs, and strong community support. Pipeline run information is stored in ***MinIO*** [42], an object storage system compatible with the *Amazon S3* [43] API, ensuring portability and seamless integration.

B. System Design

The architecture is organized into four main components: **scraper**, **back-end**, **front-end**, and **utils**, plus external services such as *PostgreSQL*, *MinIO*, and *Valhalla*. Figure 4 shows the global system architecture.

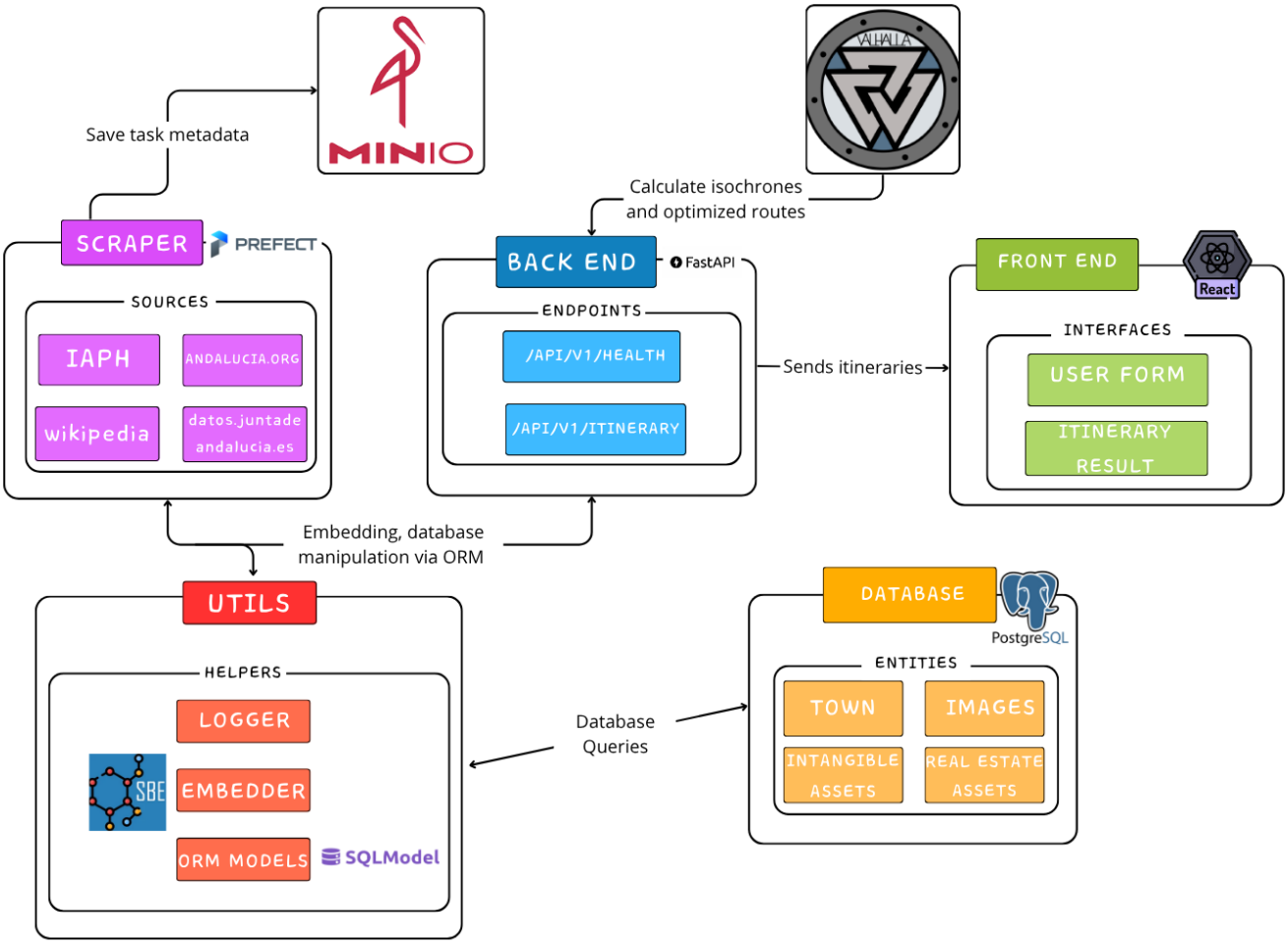

Fig. 4. System Architecture

a) Database: The database contains **towns**, **images**, **real_estate_assets**, and **intangible_assets** tables, storing geocultural and multimedia data used for both recommendation generation and presentation in the *front-end*. The entity relationships are illustrated in Figure 5.

b) Utils: This shared module provides ORM models, an embedding generator, and a centralized logging system, ensuring consistent data handling across the *scraper* and *back-end*.

c) Scraper: Implements Prefect-based tasks to collect municipality data from sources such as *Junta de Andalucía* [49], *andalucia.org* [50], *Wikipedia* [51], and the *Instituto Andaluz de Patrimonio Histórico* (*IAPH*) [52], enriching them with embeddings and storing results in *PostgreSQL*.

d) Back-end: A *FastAPI* service that integrates the database, Valhalla routing, and semantic ranking to deliver itinerary recommendations via REST endpoints.

e) Front-end: A *React + Tailwind CSS + Leaflet.js* application that interacts with the *back-end* to visualize itineraries, display detailed cultural information, and export PDF reports.

IV. Experiments

A. Database

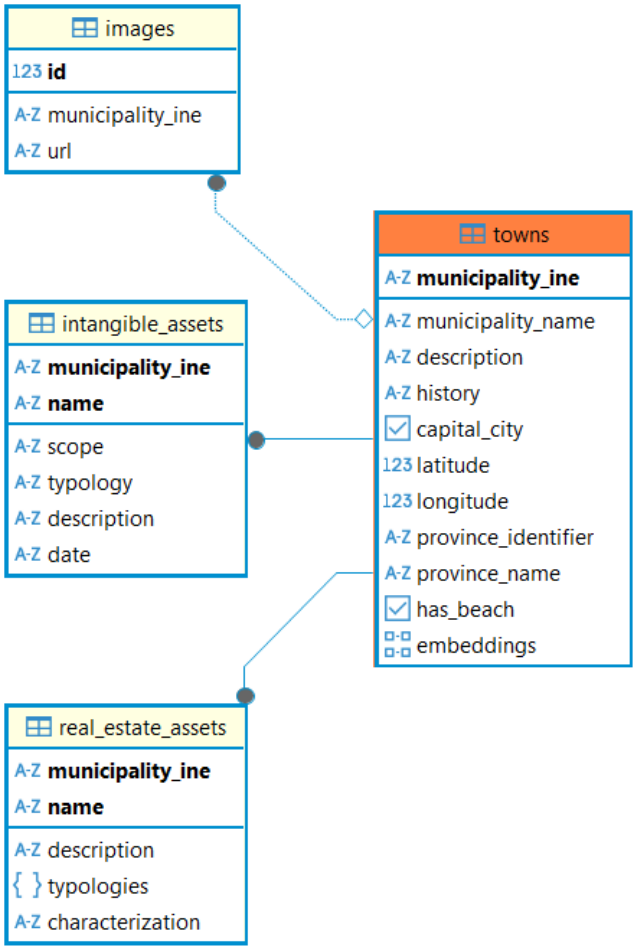

Fig. 5. Database entities and relations

The central model is **towns**, which stores detailed information for each Andalusian municipality and is linked to the other models through the shared key municipality ine. The towns table includes the following fields:

- municipality ine: Unique identifier for each municipality.
- municipality name: Official name of the municipality.
- description: Brief textual overview of the municipality.
- history: Historical background and context.
- capital city: Boolean value indicating whether it is a provincial capital.
- latitude and longitude: Geographic coordinates.
- province identifier: Numeric code representing the province.
- province name: Name of the corresponding province.
- has beach: Indicates whether the municipality has a beach.
- embeddings: Numerical vector representing the semantic profile of the municipality for recommendation tasks.

The **image** table stores visual resources associated with each municipality:

- id: Unique identifier for each image.
- municipality ine: Foreign key referencing the towns table.
- url: URL pointing to the stored image.

The **real estate assets** table captures information about the built heritage in each town:

- municipality ine: Foreign key linked to the towns table.
- name: Name of the real estate asset.
- description: General description of the heritage item.

- typologies: Set of associated architectural typologies.
- characterization: Detailed characterization based on technical or historical attributes.

The **intangible assets** table documents intangible cultural heritage linked to each municipality:

- municipality ine: Foreign key referencing the towns table.
- name: Name of the intangible cultural asset (e.g., festival or tradition).
- scope: Social or geographical scope of the cultural manifestation.
- typology: Type of expression (e.g., oral, festive, ritual).
- description: Detailed description of the asset.
- date: Date associated with the event or celebration.

These relationships form a rich and structured representation of both geographic and cultural data, essential for generating personalized travel itineraries and delivering an informative user experience.

B. Scraper

The scraping system is orchestrated through a *Prefect* flow. This pipeline coordinates a sequence of modular tasks, some executed in parallel, to extract, enrich, transform, and store structured information about Andalusian municipalities.

Prefect uses decorators to declare a flow or a task. On each decorator, you could configure key parameters such as:

- `name`: Assigns a human-readable name to the flow for easier identification in the *Prefect* UI or logs.
- `retries`: Specifies the number of automatic retry attempts for any task that fails.
- `retry_delay_seconds`: Sets the delay between retry attempts.
- `task_runner=ConcurrentTaskRunner()`: Enables parallel task execution using a thread-based task runner.

For example:

```
@flow(
    name="Town Scraper",
    retries=3,
    retry_delay_seconds=5,
    task_runner=ConcurrentTaskRunner(),
)
def main():
  ...
```

Listing 1. Flow configuration using decorator

Prefect also allows to run multiple tasks concurrently, you must call them using the `.submit()` method instead of invoking them directly. For example:

```
beach_map_future = get_towns_with_beaches_from_wikipedia.submit()
iaph_data_future = get_info_from_iaph.submit()

beach_map = beach_map_future.result()
iaph_data = iaph_data_future.result()
```

Listing 2. Parallel execution of tasks

Here, both `get_towns_with_beaches_from_wikipedia()` and `get_info_from_iaph()` are launched asynchronously. This allows them to execute in parallel, reducing total runtime. The results are later retrieved using `.result()`, which blocks execution until the corresponding task completes. This design improves performance and resource utilization, especially when tasks involve I/O-bound operations like web scraping or API calls.

The main tasks from the pipeline are:

- **Municipality list extraction:** retrieval of the official list of Andalusian municipalities from the Junta de Andalucía open data API.
- **Tourism website scraping:** extraction of descriptions, history, and images from *andalucia.org* using Selenium.
- **Supplementary data sources:** enrichment with additional information, including beach towns (Wikipedia) and cultural heritage assets (IAPH API).
- **Data consolidation:** aggregation of multi-source information into `MunicipalityInfo` objects, later converted into database-ready models.
- **Embeddings generation:** transformation of textual and structured data into semantic vectors using a *SentenceTransformer* model.
- **Database integration:** upsert of towns, images, and cultural assets into PostgreSQL through conflict-safe batch inserts.
- **Metadata storage:** upload of detailed execution reports to *MinIO* for auditing, reproducibility, and monitoring.

*1) **Municipality list extraction**:* The first task uses the *requests* library to call the open data API of the *Junta de Andalucía*, returning a list of municipalities. This serves as the base input for the rest of the flow.

*2) **Tourism website scraping**:* Once the names of all Andalusian municipalities are obtained, they are used to scrape additional information from the official tourism website *andalucia.org*. The URL for each municipality follows a predictable pattern:

`https://andalucia.org/listing/es/{municipal_name}`

To extract tourism data from *andalucia.org*, the scraper uses **Selenium** to load each municipality's dynamic page, handle the cookie banner if present, and extract three main fields: *description*, *history*, and *images*.

a) Crawl Delay and Navigation: When *respect_crawl_delay* is enabled, the `ensure_crawl_delay()` function enforces a minimum pause between requests to avoid overloading the target site. The scraper then navigates to the target URL and waits for the page tittle to be present before proceeding:

```
if respect_crawl_delay:
    ensure_crawl_delay()
    driver.get(url)
    WebDriverWait(driver, 30).until(
        EC.presence_of_element_located((By.TAG_NAME, "h1"))
    )
```

Listing 3. Open tourism page with crawl delay

b) Content Extraction: If a cookie banner is detected, it is accepted automatically. The scraper then extracts paragraphs before and after the `Historia` section, and retrieves image URLs from the gallery:

```
accept_cookies_if_present(driver)
desc = get_paragraphs_before_section_h2(driver, "Historia")
hist = get_paragraphs_after_section_h2(driver, "Historia")
imgs = extract_images(driver)
```

Listing 4. Extract description, history, and images

c) Batch Processing Strategy: Instead of opening one browser per municipality, towns are grouped into *batches*, with each batch processed by a single browser instance. This design offers:

- **Speed:** multiple batches run in parallel via `ThreadPoolExecutor`, with `max_workers` controlling the number of concurrent browsers.
- **Robustness:** if a browser instance fails, only its batch is lost, and failed URLs are logged for later reprocessing.
- **Long-run stability:** restarting the browser for each batch prevents memory leaks and gradual performance degradation that could lead to `TimeoutException` or missing element errors during prolonged scraping runs.

```
with ThreadPoolExecutor(max_workers=max_workers) as executor:
    futures = [
        executor.submit(process_batch, batch, i+1) for i, batch in enumerate(batches)
    ]
```

Listing 5. Parallel batch execution

This architecture ensures isolated browser sessions per batch, optimized performance, controlled resource usage, and reduced risk of instability in long scraping sessions.

*3) **Supplementary data sources**:* To further enrich municipality info, two more data extraction tasks are executed. As previously seen in the flow definition, these are executed concurrently: one for obtaining which towns have beaches via Wikipedia, and another for retrieving cultural asset data from the IAPH API.

- The **beaches towns** task uses the *BeautifulSoup* library to parse the HTML content of the Wikipedia page https://es.wikipedia.org/wiki/Anexo:Playas_de_Andaluca. This approach is lightweight and efficient, as the content is static and does not require dynamic rendering. Using *BeautifulSoup* instead of *Selenium* significantly reduces execution time and resource consumption.
- Meanwhile, the **IAPH data** task performs asynchronous requests to the open data API of the *Instituto Andaluz del Patrimonio Histórico*, using the high-performance *httpx.AsyncClient*. This is essential because hundreds of individual requests must be sent, one for each cultural asset. Thanks to Python's `asyncio.gather()` and `Semaphore`, the requests are managed concurrently with controlled parallelism:

```
    async with httpx.AsyncClient(...) as client:
        results = await asyncio.gather(*tasks)

```

Listing 6. Concurrent API requests with httpx

This asynchronous, non-blocking architecture allows the scraper to make efficient use of network resources and dramatically speeds up the process compared to a traditional synchronous approach.

*4) **Data consolidation**:* After all data sources have been collected and processed, the system consolidates them into a unified representation for each town in a task. Rather than directly generating database entries, this task constructs a list of `MunicipalityInfo` objects, defined as *Pydantic* models. These objects are used as a staging format to facilitate the creation of *SQLModel* models, which are the ones actually stored in the database.

Each `MunicipalityInfo` instance contains the core attributes of a town, including name, coordinates, province, and whether it is a capital city. In addition, it aggregates two related lists: `real_estate_assets` and `intangible_assets`, as well as a list of associated image URLs.

These related entities, defined as *Pydantic* models `RealEstateAsset` and `IntangibleAsset`, implement a `__str__` method. This makes it possible to convert them into readable strings that describe their typology, historical context, and other metadata.

For example:

```
def __str__(self):
    return f"The intangible asset '{self.name}' is a {self.typology} of scope {self.scope}.
    Description: {self.description}"
```

Listing 7. IntangibleAsset string representation

Thanks to this, the `MunicipalityInfo` model can define a method `get_embedding_text()` that generates a detailed, descriptive string combining all information related to the municipality:

```
class MunicipalityInfo(BaseModel):
    name: str
    #  ...(MORE ATTRIBUTES]

    real_estate_assets: List[RealEstateAsset]
    intangible_assets: List[IntangibleAsset]


    def get_embedding_text(self) -> str:
        parts = [f"{self.name or 'Unknown'} is a municipality"]
        parts.append("that is a provincial capital." if self.capital else "that is not a
    provincial capital.")

        if self.description:
            parts.append(f"Description: {self.description}")
        if self.history:
            parts.append(f"History: {self.history}")

        for asset in self.real_estate_assets:
            parts.append(str(asset))
        for intangible in self.intangible_assets:
            parts.append(str(intangible))

        return " ".join(parts)

```

Listing 8. Generate unified embedding text

This structure ensures that each town is not only enriched with multi-source data, but also represented semantically in a vector space for personalized recommendations.

*5) **Embeddings generation**:* After constructing the list of `MunicipalityInfo` objects, the `generate_embeddings()` task performs two main operations: generating semantic embeddings and converting the enriched data into *SQLModel* models ready for insertion into the database.

This task begins by processing the municipalities in batches. For each batch, it first calls the `get_embedding_text()` method on every municipality object. This method produces a comprehensive string that summarizes the town's description, history, and associated cultural assets.

```
batch_strings = [item.get_embedding_text() for item in current_batch]
```

Listing 9. Generate a list of embedding strings

These strings are then passed to the `get_embedding()` function defined in the shared *utils* module. This function leverages a *SentenceTransformer* model to encode each string into a high-dimensional semantic vector:

```
batch_embeddings = get_embedding(batch_strings)
```

Listing 10. Transform text to vector embeddings

Once the embeddings have been computed, the task creates *SQLModels* models from the original `MunicipalityInfo` objects. The main town information is converted into a `Town` instance, and each related asset is converted into either an `Intangible`, `RealEstate`, or `ImageTown` object:

```
town = Town(
    municipality_ine=municipality.ine,
    municipality_name=municipality.name,
    ...
    embeddings=embedding.tolist(),
)

for asset in municipality.intangible_assets:
    intangible_assets.append(Intangible(...))

for asset in municipality.real_estate_assets:
    real_estate_assets.append(RealEstate(...))

for url in municipality.images:
    images.append(ImageTown(...))
```

Listing 11. Create *SQLModel* database objects

These objects are collected into four lists: `towns`, `intangible_assets`, `real_estate_assets`, and `images` and returned as a tuple. They are now fully structured and ready to be upserted into the database. In summary, this task forms the bridge between semantic enrichment and persistent storage. It transforms structured, textual, and multimedia information into vectorized representations and SQLModel-compatible database objects, making them ready for recommendation and retrieval.

6) ***Database integration****:* The final stage of the scraping pipeline is handled by the task `load_info_to_postgres()`, which inserts all enriched and embedded data into a *PostgreSQL* database using **upsert** logic. This means that if a record already exists (based on a defined uniqueness constraint), it will be updated instead of causing a conflict or duplication.

Upsert operations are implemented via *SQLModel*'s support for *PostgreSQL*'s `ON CONFLICT` clause. The helper function `build_upsert_stmt()` receives a list of model instances or dictionaries and generates a bulk insert with upsert behavior:

```
stmt = pg_insert(model).values(dict_rows)
stmt = stmt.on_conflict_do_update(
    index_elements=conflict_cols,
    set_={col.name: stmt.excluded[col.name] for col in model.__table__.columns if col.name not
    in conflict_cols}
)
```

Listing 12. *SQLModel* upsert with conflict resolution

This mechanism ensures that the same data can be re-processed and reinserted multiple times without causing duplicate entries or violating primary key constraints. It makes the entire pipeline **idempotent**, which is critical for robustness in data workflows that may be re-executed due to retries or scheduled runs.

Before insertion, each batch is **deduplicated** by applying a custom key function that filters out repeated entries. For instance, towns are deduplicated using the `municipality_ine` identifier, while cultural assets are filtered using the tuple `(municipality_ine, name)`.

```
deduplicated_towns = deduplicate_records(towns, key_func=lambda x: x.municipality_ine)
```

Listing 13. Deduplication by custom key

The records are processed in small chunks (default: 1000 records per batch), which reduces memory usage and avoids database overload. Once all upsert statements are executed, a single transaction commit ensures consistency:

```
session.commit()
```

Listing 14. Commit all batched inserts

Thanks to the combination of deduplication, conflict-safe upserts and batch processing, this function guarantees that the entire scraping pipeline can be safely rerun at any time without corrupting the dataset or introducing redundancy.

The last task in the scraping flow is `save_task_metadata_to_minio()`, which uploads a structured report of the execution to an object storage service in this case, ***MinIO***. This report includes detailed metadata for every task run, such as start and end times, duration, state, retries, and execution identifiers.

7) ***Metadata storage****:* Object storage services like *MinIO* organize files into **buckets**, which function similarly to folders in a cloud filesystem. Buckets allow logical grouping of reports by execution date, version, or environment, and support fast and scalable retrieval of historical logs. In this project, the reports are stored in timestamped paths such as:

`2025-07-17T14-45-00Z/task_metadata.json`

These reports are useful for:

- **Auditing and monitoring** the flow's performance over time.
- **Debugging errors** or tracking failures during retries.

- **Reproducibility**, by providing a snapshot of what was executed, when, and under what conditions.

Under the hood, the *MinIO* API is fully **Amazon S3-compatible**, meaning that the same logic could be used with *AWS S3* [43] or other cloud providers by simply switching the endpoint and credentials.

The report is serialized to JSON and uploaded via the *MinIO* client:

```
minio_client.put_object(
    bucket_name=BUCKET,
    object_name=object_name,
    data=byte_stream,
    length=len(json_bytes),
    content_type="application/json",
)
```

Listing 15. Upload metadata to textitMinIO

This design decouples execution reporting from the database, and makes the metadata persistently accessible, portable, and ready for future integration with dashboards or monitoring systems.

C. Back-end

The back-end is implemented using ***FastAPI*** and exposes two main endpoints: a `/health` endpoint to check system status, and a `/itinerary` endpoint that generates a personalized travel route.

a) Health Endpoint: The `/health` endpoint verifies that the API is running, the database connection is active, and the *Valhalla* routing service is available. It returns a simple status response used by monitoring tools.

b) Itinerary Generation Endpoint: The recommendations obtained from are a direct result of the semantic similarity search applied to the answers in the form that will be send from the front-end. The system converts the textual preferences, such as type of village, environment, monuments of interest, historical periods, cultural influences, travel interests, and traditions, into an embedding vector. This vector is then compared against the stored embeddings of all towns in the database using cosine similarity.

1) **Filtering Towns**: Towns are fetched from the database applying filters such as beach preference.

```
query = select(Town).options(
    selectinload(Town.images),
    selectinload(Town.intangible_assets),
    selectinload(Town.real_estate_assets),
)
if form_data.beach and form_data.beach != "indiference":
    query = query.where(Town.has_beach == (form_data.beach == "yes"))
    result = await db_session.execute(query)
    towns = result.scalars().all()

```

Listing 16. Filter towns based on form preferences

If the user provides a location and travel time limit, the list is refined using a *Valhalla* isochrone filter:

```
towns = await filter_by_location_polygon(
    form_data.location, form_data.travelTimeLimit, towns
)

```

Listing 17. Filter towns by isochrone reachability

2) **Semantic Ranking**: The user's answers are converted into a text string and embedded into a vector. The towns are ranked by cosine similarity between the user embedding and each town embedding, and the top 3 are selected:

```
form_embedding = get_embedding(form_data.get_embedding_text())
ranked_towns = rank_towns_by_similarity(form_embedding, towns) top_towns = ranked_towns[:3]

```

Listing 18. Rank towns by embedding similarity

Similarity between the user embedding and each town embedding is computed using `cosine_similarity` from `scikit-learn`, which returns values in the range [0, 1] (1 = most similar). The towns are then ranked in descending order of similarity:

```python
from sklearn.metrics.pairwise import cosine_similarity

def rank_towns_by_similarity(user_embedding, towns: list[Town]) -> list[Town]:
    town_vectors = ...
    similarities = cosine_similarity([user_embedding], town_vectors)[0]
    sorted_towns = ...
    return sorted_towns

```

Listing 19. Rank towns by cosine similarity (scikit-learn)

3) **Optimal Route Calculation** The selected towns are ordered to minimize travel time using *Valhalla*'s routing engine:

```python
    trip = await get_optimal_route([
        Coordinate(lat=town.latitude, lng=town.longitude)
        for town in top_towns
    ])

```

Listing 20. Generate optimized travel route

This modular approach ensures that each step can be maintained and improved independently, while ***Pydantic*** models guarantee validation and consistent data formats in API responses.

D. Front-end

The front-end, built in *javascript* with *React* has a main `Home` component, which functions as a **state machine** controlling the interface flow between the form view and the results view. This is managed using the *React* `useState` hook with two possible states: `'form'` and `'results'`.

```javascript
const [state, setState] = useState('form'); // 'form' | 'results'
```

Listing 21. Basic state machine logic

When a user submits the form, the front-end sends a request to the back-end to generate a tailored itinerary. If successful, the application transitions to the `results` state and displays the recommendations. Otherwise, an error toast is shown and the view returns to the form.

To preserve the application state across page reloads or accidental navigation, the component leverages the browser's ***localStorage* API**. It stores the current state, the itinerary results, and the user-selected location. This is handled reactively via `useEffect()`:

```javascript
if (state === 'results') {
  localStorage.setItem('itineraryResults', JSON.stringify(results));
  localStorage.setItem('viewState', 'results');
} else {
  localStorage.removeItem('itineraryResults');
  localStorage.setItem('viewState', 'form');
}
```

Listing 22. Persist view state in localStorage

The `Home` component conditionally renders two key subcomponents based on the application state:

- `VillageForm`: a detailed form that collects the user's preferences regarding town type, culture, traditions, proximity to beaches, and location. It uses reusable components like `Section`, `Textarea`, and `LocationPicker`.
  The `LocationPicker` component provides an interactive map interface that allows users to search and select a location within the *Andalucía* region. It integrates several libraries to offer geospatial functionalities. At its core, it uses ***react-leaflet***, a React wrapper for the *Leaflet* JavaScript library, to render the interactive map, handle markers, layers, and events. The ***leaflet-geosearch*** library, combined with the *OpenStreetMapProvider*, enables address search through a custom geocoding control embedded in the map. The component also loads a regional *GeoJSON* file containing the boundaries of Andalucía and uses a library called *@turf/turf* [53] to verify whether the selected location falls within these boundaries. This ensures that users can only confirm valid locations.
- `ItineraryResults`: an interface that displays a dynamic map (via *Leaflet*) along with recommended towns, embedded images, and heritage data. It also supports full-screen modal interactions and a rich info panel with expandable sections. This component also uses a report generation component that relies on a combination of front-end libraries to create a dynamic and visually rich PDF itinerary. ***jsPDF*** is the core library used to create and format the PDF document, while ***html2canvas*** captures visual content from the DOM and converts it into canvas images for embedding into the PDF. ***leaflet-image*** is employed to render the interactive *Leaflet* map view into an image, including custom polyline routes derived from decoded coordinates using *@mapbox/polyline* [54].

Additional libraries enhance the user experience: ***react-toastify*** [55] provides user notifications about the report generation status, ***lucide-react*** [56] supplies iconography for visual feedback, and React's native ***useState*** hook is used to manage the loading state. Together, these tools orchestrate the transformation of itinerary data—towns, descriptions, monuments, traditions, and images—into a structured, multi-page report with embedded maps and localized cultural information.

The entire interface is styled using ***Tailwind CSS***, a utility-first *CSS* framework that enables rapid prototyping and responsive design directly in the *JSX* code. Responsive design is handled seamlessly through breakpoint utilities such as `sm:flex-row` and `lg:grid-cols-8`, ensuring the interface adapts across devices without writing custom *CSS*.

E. Tests

Unit tests were conducted to verify the correct functioning of both the back-end and the scraper. No tests were implemented for the front-end since its logic is limited to presentation and does not include critical business processes or data transformations that would justify automated unit testing; instead, it will be validated through manual user interface testing and visual inspection.

1) Back-end: : Back-end tests focused on ensuring uniform JSON serialization of responses, correct construction of *Pydantic* models from ORM instances, input normalization in forms, robustness of the health endpoint under partial subsystem failures, and the proper orchestration of the itinerary controller.

a) **GenericResponse** tests confirm that `to_json_response()` produces a proper FastAPI `JSONResponse`: it preserves the HTTP status code and serializes the `code`, `message`, and `data` fields into JSON. This ensures that all backend controllers can consistently wrap results with a uniform response contract.

b) **TownOut** the Pydantic model `TownOut` is validated directly from an ORM instance of `Town`. Tests confirm correct mapping of nested relations (first image URL, real estate assets, intangible assets), demonstrating that SQLModel instances serialize into API output as expected.

c) **FormResponse** Tests cover multiple normalization rules:

- Empty strings are coerced to `None` (except `beach`, which defaults to `"indiference"`).
- `travelTimeLimit` parses numeric strings into integers and raises `ValidationError` on invalid values.
- `get_embedding_text()` builds a descriptive phrase concatenating all preferences (location label, interests, periods, environment, influences, village type), while returning an empty string when no preferences are set.

d) **Health endpoint (`/api/v1/health`)** Four scenarios are covered:

- *Healthy*: database and Valhalla both connected ⇒ overall `"ok"`.
- *Degraded (DB)*: database fails, Valhalla healthy ⇒ overall `"degraded"`, with `database="disconnected"`.
- *Degraded (Valhalla)*: Valhalla unhealthy, DB connected ⇒ overall `"degraded"`, with `valhalla="disconnected"`.
- *Degraded (both)*: both subsystems fail ⇒ overall `"degraded"` with both marked as `"disconnected"`.

`respx` stubs the Valhalla `/status` call, while FastAPI dependency overrides inject fake DB sessions to simulate connectivity.

e) **Itinerary controller (`get_itinerary`).** Tests validate the orchestration of ranking, filtering, and routing using doubles and monkeypatching:

- *Happy path*: returns `200` with a valid trip, limiting results to three towns.
- *No matches*: empty isochrone filter ⇒ `404` with error message.
- *Not enough locations*: only one destination and no start point ⇒ `204`.
- *Routing error*: exception during route computation ⇒ `500`.
- *Beach prioritization*: with `beach="yes"`, beach towns are promoted even if inland towns are present.
- *Ranking order*: cosine similarity ordering is strictly descending, ensuring closest matches appear first.

All responses are wrapped using `GenericResponse.to_json_response()`, guaranteeing a consistent output structure across success and error cases.

2) Scraper: The following list enumerates the test cases defined for each task, helper, and the main flow of the scraper. We summarize what is being validated and highlight the significant use of mocks and patches.

a) **`get_andalusia_towns_ubi_and_name`** tests verify that JSON responses are correctly filtered to Andalusian towns and coordinates are parsed into floats. Another case ensures HTTP errors (status code 500) raise exceptions. Here, `monkeypatch` replaces the `requests` module to simulate both successful and failed responses.

b) **`scrape_from_turismo_andalucia`** cases include: (i) Unicode name normalization (diacritics and spaces), (ii) acceptance of cookies when the button is present or missing, (iii) extraction of paragraphs before/after specific `<h2>` sections, (iv) image extraction with CSS selectors, (v) full scrape returning description/history/images with patched DOM helpers, (vi) retry logic on timeouts until success, and (vii) graceful handling of empty batches. Selenium interactions are mocked with `MagicMock`, while `patch.object` is used to substitute helper functions during scraping.

c) **`generate_embeddings`** validates that the function always returns a 4-tuple, collects texts from towns in a single batch call, and correctly maps real estate and intangible assets into model objects. Tests cover image flattening,

preservation of optional fields, batching (splitting 35 towns into two calls of 32+3), and error handling when an embedding call fails. The embedding function is replaced by a fake function with `monkeypatch.setattr` to capture texts and simulate failures.

d) **`save_data`** Unit tests validate utility functions: `chunked` splits iterables correctly, `build_upsert_stmt` works with empty input, and `deduplicate_records` removes duplicates. Integration tests confirm that towns are persisted into an in-memory SQLite database using patched `get_engine` and `get_session`. Duplicates are skipped, and errors in engine acquisition are caught without crashing.

e) **`upload_report`** tests focus on the `json_serial` helper, ensuring correct serialization of `datetime`, `timedelta`, and `UUID`, and raising `TypeError` for unsupported types. The main async task is tested in two scenarios: (i) happy path with two fake Prefect task runs, uploaded to MinIO as a JSON object, and (ii) missing/optional fields where defaults are applied (e.g., unknown state, empty tags). Prefect clients and MinIO are replaced with fakes, and `patch_minio` captures `put_object` calls for assertion.

f) **`merge_municipality_info`** cases verify normalization of names with articles (*El, La, Los, Las*), trimming and lower casing. Grouping assets by normalized municipality name is checked. The merging process is tested with full data (town, assets, beaches) and ensures correct mapping of fields and asset attachment. A negative case ensures towns without names are skipped.

g) **`wikipedia_beach_check`** tests confirm correct extraction of towns from Andalusian provinces in HTML, filtering out non-Andalusian ones, handling empty sections, and removing duplicates. Errors (HTTP 500) are caught, returning an empty list. HTML input is crafted inline and injected by monkeypatching the `requests` module.

h) **`get_info_from_iaph`** asynchronous tests verify retry behavior on 503 errors, trimming of intangible typologies into sets, and correct merging of base and detail data. Edge cases include empty base data (returns empty list) and failed detail fetches (falls back to stub). Async HTTP clients are replaced with a mock that records calls.

i) **helpers.minio** tests check that the *MinIO* helper creates buckets when missing and skips creation if they already exist. The *Minio* client class and settings are monkey patched with fakes to capture behavior.

j) **helpers.selenium** tests cover driver aliveness checks (valid, missing session, invalid service, exception), reusing existing drivers, creating new ones if none exist or if the previous is dead, and thread-local isolation. `patch.object` is used to substitute driver initialization and status checks.

k) **models.municipality** tests ensure that `RealEstateTypology.__str__` is robust against missing or partial fields, covering cases with only name, ethnicity, period, activity, and combinations of all of them.They validate that the string never leaks the literal "None" and that spacing/normalization is consistent.
`RealEstateAsset.__str__` is tested with default fallbacks, multiple typologies, and safe handling of empty names or lists, ensuring outputs always remain meaningful
`IntangibleAsset` tests verify that typology strings are deterministic (sorted), that missing fields use explicit fallbacks, and that the string is always coherent.
Finally, `MunicipalityInfo.get_embedding_text()` is checked to aggregate description, history, capital flags, real estate assets, and intangible assets into a single text without leaking "None", even over large batches.

l) **main flow** tests validate the execution order of the flow: towns → enrichment → merge → embeddings → save. Concurrent tasks (`get_beaches`, `get_iaph`, `upload_report`) are verified to be submitted and awaited. Data plumbing is tested by setting fake inputs and outputs and ensuring they are passed correctly. Finally, the flow's return value is checked to be `None`. All dependencies are mocked with a single fixture (`mock_all_tasks`).

V. Results

A. Scraper

The execution of the flow may occasionally fail due to the unreliability of the *Junta de Andalucía*'s data API. Sometimes the API fails to return any data at all, an error originating from the API itself. This issue is handled by retrying the corresponding task. Fortunately, this step is quite fast, as it only involves a single API request.

From there, the workflow proceeds to the task responsible for scraping *andalucia.org* using ***Selenium***. This is a relatively slow process because it simulates user interactions by clicking through each tab on the website. However, the task is parallelized to improve performance. On the test machine, the parallelism was capped at 8 concurrent executions. The time required for this scraping stage can vary significantly depending on the machine's power.

Once completed, the flow moves on to the Wikipedia scraping task, which is much faster thanks to the use of ***BeautifulSoup*** to extract data from static HTML pages.

The subsequent task targets the IAPH source, which is slower due to the large number of API calls involved. Although concurrency employed, which is beneficial in high I/O-bound scenarios, it still takes considerable time because of the volume of requests. Concurrency helps here by overlapping waiting times and maximizing throughput when handling many simultaneous HTTP operations.

Next, the data transformation and embedding generation phase is relatively fast, especially when the embedding model is preloaded and stored in memory. The insertion of data into the *PostgreSQL* database is performed in batches, which significantly

reduces execution time by minimizing the overhead of multiple individual database transactions. Finally, a task uploads the execution report to ***MinIO*** (see Figure 6).

Execution times for the entire workflow are summarized in Figure 7, although it's important to note that these mat vary significantly depending on the machine used to execute the pipeline.

In summary, as a result of the flow execution we will have the complete database service with all the data of the municipalities and their embeddings, in other words, we will have all the entities described above (Figure 5) and we will have the reports in the service of *Minio*.

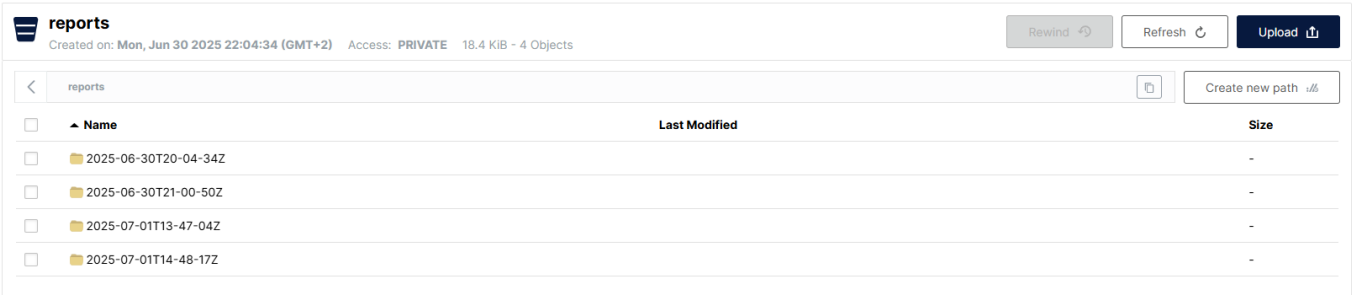

Fig. 6. Reports store in *MinIO*

Fig. 7. *Prefect* dashboard that shows a pipeline flow run

B. Back-end

Once the *Valhalla* and *back-end* services are up and running, two main API endpoints become available for interaction. These endpoints can be accessed and tested via the automatically generated Swagger documentation, as shown in Figure 8.

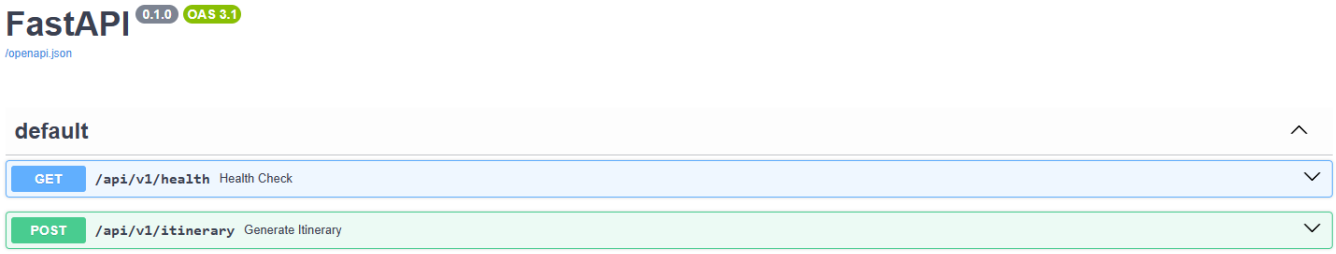

Fig. 8. API documentation deployed when the API is active

To test the effectiveness of the embedding, a practical example was carried out in which the expected data was sent to the endpoint to verify whether the response made sense (Figures from 9 to 11) The data sent to the endpoint was:

- `villageType`: A medium-sized urban center with historical character, preserving a rural environment and keeping traditional customs alive. Narrow cobblestone streets, whitewashed houses with balconies full of flowers, and a clear

harmony between architecture and landscape. The place should convey authenticity, with a strong identity and a direct connection between its history and daily life.

- `environment`: A peaceful and safe atmosphere, with a leisurely pace that invites strolling and quiet observation, yet enough cultural activity to discover exhibitions, concerts, markets, and social gatherings. The setting should foster harmony between residents and visitors, with open spaces and panoramic views of a mountainous natural landscape.
- `monuments`: Monumental structures that combine historical engineering and aesthetic value: defensive fortifications with walls and towers, medieval-era bath systems, public squares of historical significance, palaces blending multiple architectural styles, and religious buildings with artistic and architectural merit. Museums and interpretive spaces explaining the different stages of its development.
- `historicalPeriods`: Ancient times that preserve what remains from the Roman era, medieval periods with Andalusian influence, and phases of the Modern Age when major architectural works were developed. I am also interested in the contemporary period that kept traditions alive and generated its own cultural imagery, including the Romanticism of the 19th century
- `culturalInfluences`:Especially those that bear traces of Andalusí culture in their urban layout, defensive structures, and domestic architecture, as well as remnants from the Roman era in the form of infrastructure, buildings, and archaeological artifacts. I am drawn to places where different cultures have left a recognizable and still-visible mark on the present.
- `travelInterests`: To deeply understand the history of a place through its monuments, museums, and local stories; to savor its traditional cuisine made with regional products; to uncover legends and tales passed down through generations; to enjoy its unique cultural and artistic activities; and to stroll through nearby natural surroundings—all for a complete experience that blends culture, nature, and gastronomy.
- `traditions`: Festivals that blend cultural, artistic, and gastronomic activities, particularly those with deep historical or symbolic significance. Patron saint celebrations with processions, fairs featuring traditional performances, and gatherings showcasing local music, dance, and crafts. I'm also keen on taking part in markets and events promoting native agricultural products and regional foods.
- `beach`: Indiference (open to inland or coastal towns).
- `location`: None (no starting point specified).
- `travelTimeLimit`: None (no isochrone filter applied).

The results obtained were:

- **Baños de la Encina**: Historic town with a large medieval fortress, religious monuments, and a preserved old quarter.
- **Vera**: Coastal town with an important historic centre, traditional architecture, and cultural events.
- **Alájar**: Mountain village with traditional whitewashed architecture, surrounded by natural parks, and known for its festivities and gastronomy.

This result is reasonable because the description emphasized historical architecture, multiple cultural layers, and popular traditions, characteristics present in all three towns. Since `beach = indiference`, the system did not exclude coastal towns, which allowed Vera to appear in the results. The absence of `location` and `travelTimeLimit` constraints meant that the system searched the entire Andalusian dataset, prioritizing thematic similarity over geographical proximity.

The recommendations are still relevant because they all offer significant monumental heritage, a quiet environment with preserved architecture, and festivals and traditions aligned with the stated interests. In this particular example, the matches are strong because the towns share key characteristics with the described preferences: a rural or small-town environment, preserved traditional architecture, and a rich historical and cultural heritage that includes notable monuments and popular festivities.

However, if the goal were to retrieve a specific single town by name, it would be challenging to achieve consistently. This is because the form is designed to capture thematic preferences (heritage, culture, traditions) rather than explicit location identifiers. Many towns in Andalusia share similar historical profiles (medieval fortifications, religious monuments, and traditional festivals) which increases the overlap in results.

To narrow the results toward a more precise match, adding a geographic constraint would be highly effective. For example, selecting a starting location and applying an isochrone filter (e.g., "within 60 minutes' drive") would restrict the search space and prioritize nearby options that still match the cultural profile.

After the top towns were chosen, the system generated an optimized *Valhalla* route: without a starting location, the route began from the first recommended town and calculated the most efficient visiting order. It included driving times, distances, and turn-by-turn navigation for each leg (Figure 13).

C. Front-end

To illustrate the user experience, this section presents several screenshots of the *front-end* interface. The graphical user interface has been designed with usability and clarity in mind, allowing users to easily select their preferences, explore towns, and visualize optimized travel routes. The *React* interface adapts responsively to different screen widths and allows downloading a detailed **PDF report** with the obtained itinerary.

1) Form: The form is divided into thematic sections where the user specifies preferences about town type, cultural and historical interests, experiences, traditions, and geographical constraints. Each section contributes to the generation of a personalized itinerary through semantic similarity searches.

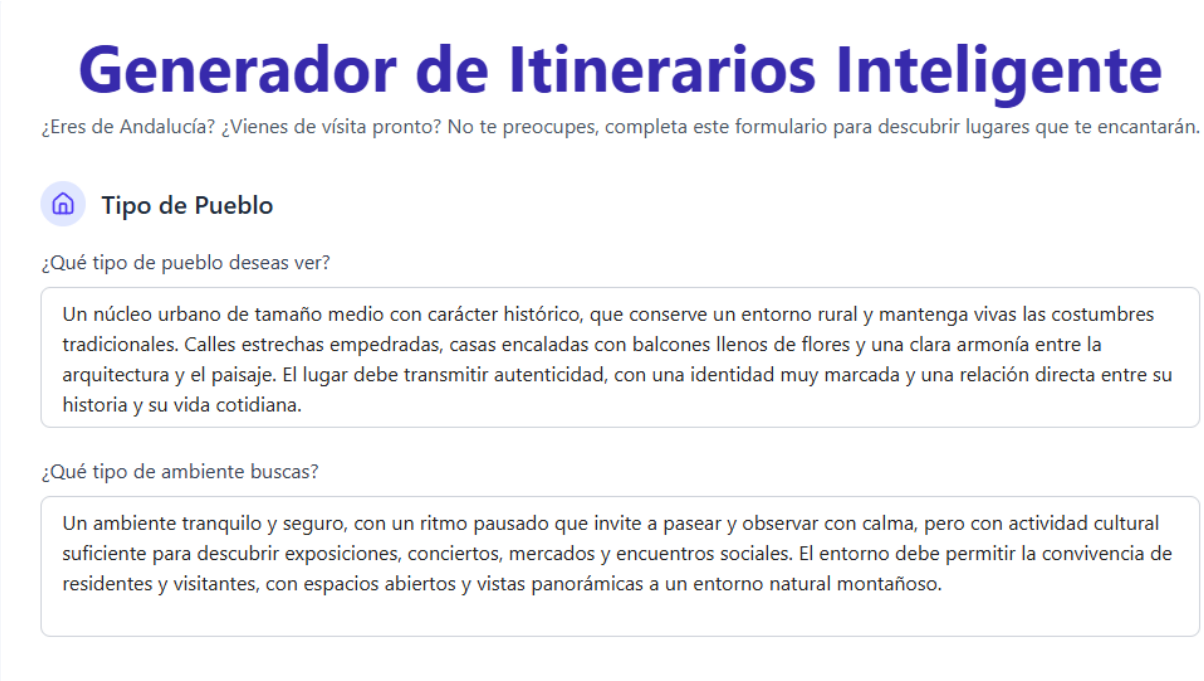

Fig. 9. *Town Type* section, where the user describes the desired characteristics of the town and the general atmosphere sought.

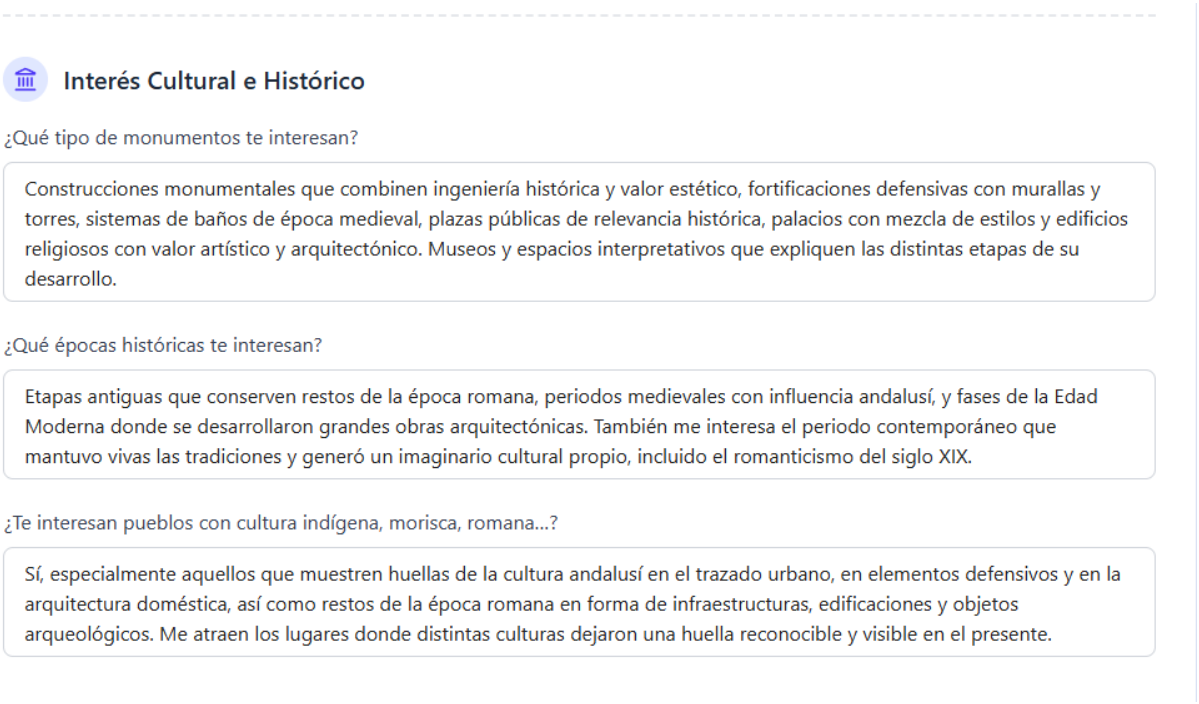

Fig. 10. *Cultural and Historical Interest* section, capturing the types of monuments, historical periods, and cultural influences of interest.

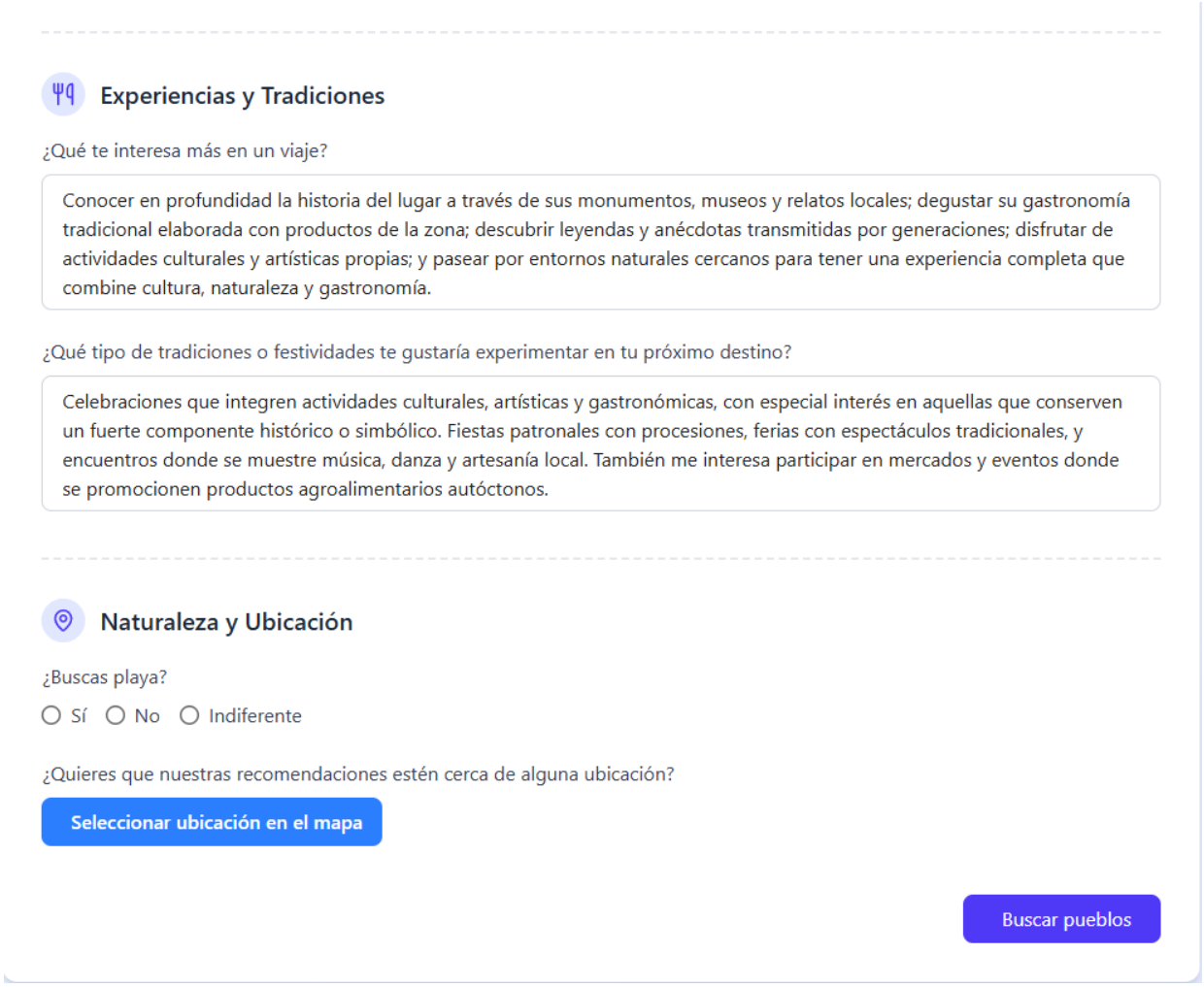

Fig. 11. *Experiences and Traditions* section, where the user selects the preferred cultural activities, festivities, and gastronomic experiences.

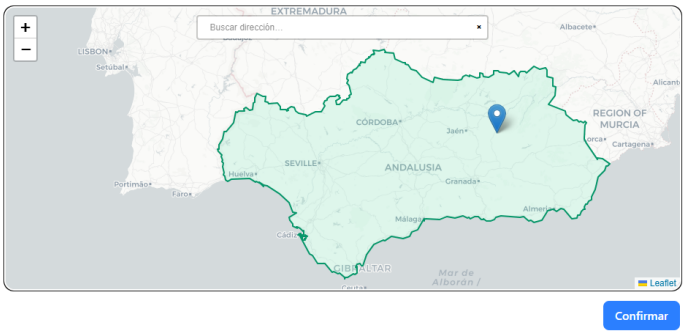

Fig. 12. Location selector with an interactive *Leaflet* map restricted to Andalusia, allowing the user to define a starting point by clicking or using the search bar.

2) Itinerary Visualization: Once the form is submitted, the *front-end* displays the results as an interactive itinerary, showing selected towns, detailed cultural heritage information, and an optimized route computed with *Valhalla*.

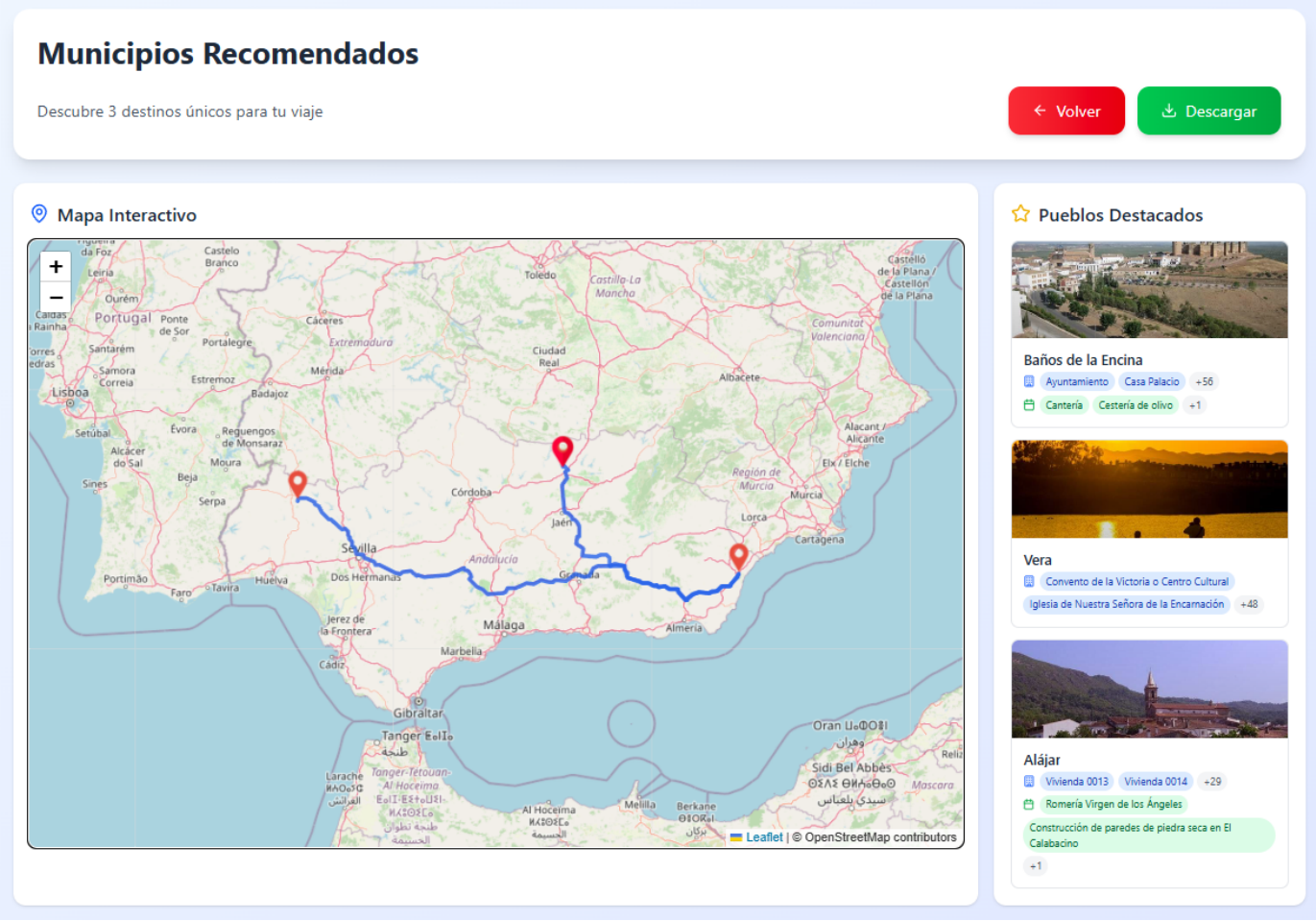

Fig. 13. Overview of the recommended municipalities with an interactive route map and a sidebar with highlighted towns.

Fig. 14. Full town profile view with images, description, history, and points of cultural interest.

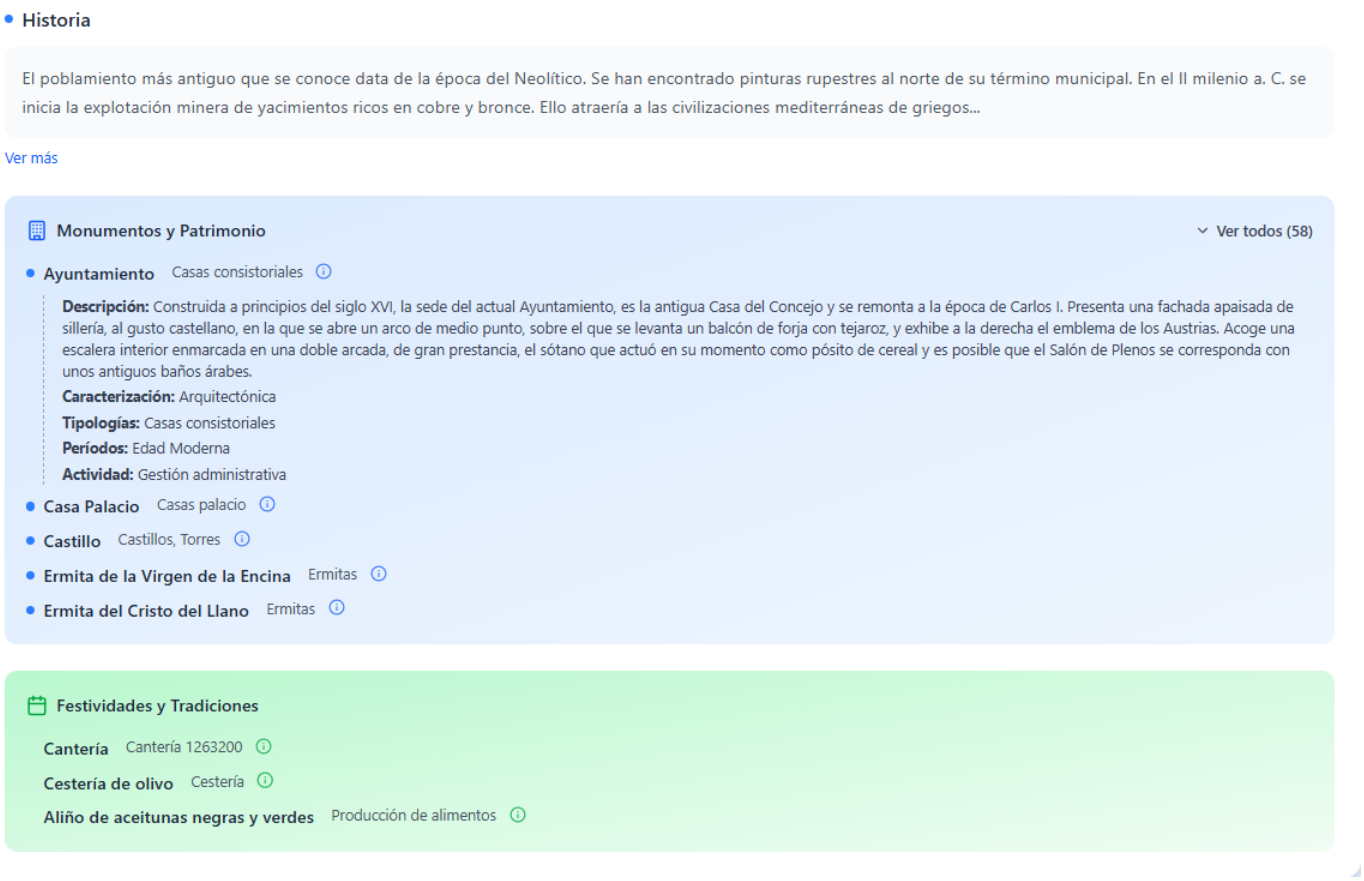

Fig. 15. Detailed cultural heritage and traditions for a selected town, including historical description and architectural assets.

VI. Conclusions

This work successfully delivered a reproducible, modular platform that combines **semantic ranking** and **geospatial routing** to generate personalized travel itineraries across Andalusia. The integration of *FastAPI*, *Valhalla*, *PostgreSQL*, *React*, and *Leaflet*, along with a robust ETL pipeline orchestrated via *Prefect*, demonstrates the feasibility of building an end-to-end tourism recommendation system with modern, open-source technologies.

The project achieves a clean separation of concerns across the *scraper*, *backend*, *frontend*, and *utils* modules, which improves maintainability and portability. The pipeline supports automated ingestion of heterogeneous cultural and tourism data, transforms it into semantically enriched embeddings, and serves them through a responsive API and user interface. The **object storage integration** with *MinIO* and the **idempotent upsert logic** in the database provide resilience against reprocessing and data duplication.

Beyond the technical execution, this platform demonstrates that **semantic search techniques** can be effectively adapted to geocultural contexts, producing itineraries that balance user preferences with travel-time constraints.

A. Achieved Objectives

- Implementation of a fully containerized, end-to-end stack covering scraping, semantic ranking, routing, and visualization.
- Development of an *interactive front-end* enabling location-based filtering, cultural data exploration, and PDF itinerary export.
- Deployment of a *FastAPI* back-end with asynchronous data access, semantic scoring, and *Valhalla*-powered routing.
- A reproducible ETL workflow, monitored and retriable, using *Prefect*, with detailed execution reports in *MinIO*.
- Clean integration of ORM models across modules using *SQLAlchemy* and *Pydantic*.

B. Challenges

- **Data reliability**: Upstream APIs—especially from *Junta de Andalucía*—were sometimes unstable, requiring retry mechanisms and fallback strategies.
- **Dynamic scraping complexity**: Handling *JavaScript*-heavy pages in *andalucia.org* with *Selenium* introduced performance constraints mitigated by batch parallelization.
- **Pipeline orchestration at scale**: Maintaining concurrency without overloading services demanded careful rate-limiting and resource allocation.
- **Embedding trade-offs**: Balancing the semantic accuracy of *SentenceTransformers* against inference latency for real-time recommendations.

C. Improvements and Future Work

- **Evaluation with user studies**: Conduct systematic usability testing to measure the perceived relevance and cultural value of generated itineraries.
- **Caching and incremental ETL**: Reduce reprocessing costs by detecting unchanged data sources and persisting computed embeddings.
- **Advanced preference modeling**: Implement multi-objective ranking incorporating environmental, seasonal, and accessibility factors.
- **Internationalization and accessibility**: Expand UI translations, adapt for diverse audiences, and improve accessibility compliance.

In summary, this project not only provides a working system but also a blueprint for scalable, AI-driven tourism platforms that respect local heritage, integrate diverse data sources, and deliver personalized cultural experiences.

APPENDIX A
INSTALLATION MANUAL

Prerequisites

Before starting, ensure that you have the following installed:

- You can find the source code of the project in the *GitHub* repository: https://github.com/cberdejo/Smart-Itinerary-Generator
- ***Docker*** and ***Docker Compose*** for containerized deployment.
- Alternatively, **Python 3.12** with *uv* or *pip* for manual installation.
- ***Node.js*** and *npm* if running the ***React*** frontend without Docker.

Running the Entire Project with Docker

This is the simplest way to run all components (*scraper*, *utils-project*, *backend*, and *frontend*) at once. You can modify the service URIs and ports in the `docker-compose.yml` file.

```
git clone https://github.com/cberdejo/Smart-Itinerary-Generator.git
cd Smart-Itinerary-Generator
docker-compose up --build -d
```

Recommendation run the scraper pipeline first. Start the pipeline services with:

```
docker compose up --build -d minio prefect-server postgres scraper
```

You can monitor the pipeline execution at http://localhost:4200 (unless the URIs or ports were modified). This dashboard provides real-time information about the pipeline process. Once the pipeline has completed and the data is stored in the database, you can start the remaining services:

```
docker compose up --build -d valhalla backend frontend
```

The scraper service can be relaunched at any time to re-run the pipeline.

Manual Installation of Scraper

1) Navigate to the `scraper` folder.
2) Create a `.env` file with:

```
# PostgreSQL
POSTGRES_HOST=localhost
POSTGRES_PORT=5432
POSTGRES_DB=your_database
POSTGRES_USER=your_user
POSTGRES_PASSWORD=your_password

# MinIO (S3-compatible)
MINIO_ENDPOINT=http://localhost:9100
MINIO_ACCESS_KEY=minioadmin
MINIO_SECRET_KEY=minioadmin
MINIO_BUCKET=scraper-reports

# Embeddings / NLP
EMBEDDINGS_MODEL=sentence-transformers/all-MiniLM-L6-v2

# Optional: Prefect server
PREFECT_API_URL=http://localhost:4200/api

# Selenium settings
SELENIUM_MODE=headless
```

3) Install dependencies using *uv*:

```
uv pip install -e ../utils-project
uv pip install -e .
```

4) Or using *pip*:

```
python -m venv .venv
```

```
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -e ../utils-project
pip install -e .
```

5) (Optional) Start *Prefect* UI to monitor runs:

```
prefect server start
```

6) Run the scraper:

```
python -m pipeline.main
```

7) Ensure that **PostgreSQL** and **MinIO** services are running before execution.

Manual Installation of Backend

1) Navigate to the `backend` folder.
2) Create a `.env` file with:

```
HOST=0.0.0.0
PORT=8000
VALHALLA_URL=http://valhalla:8002
POSTGRES_DB=your_database
POSTGRES_USER=your_user
POSTGRES_PASSWORD=your_password
POSTGRES_HOST=your_db_host
POSTGRES_PORT=5432
```

3) Install dependencies using *uv*:

```
uv pip install -e ../utils-project
uv pip install -e .
uv run src/api/main.py
```

4) Or using *pip*:

```
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -e ../utils-project
pip install -e .
python src/api/main.py
```

5) Ensure a **Valhalla** routing service is running.

Manual Installation of Frontend

1) Navigate to the `frontend` folder.
2) Create a `.env` file with:

```
VITE_API_URL=http://localhost:8000/api/v1
```

3) Install dependencies:

```
npm install
npm run dev
```

4) The application will be available at http://localhost:5173.

APPENDIX B
TESTING MANUAL

1. Prerequisites

Before running the test suite, ensure that you have installed the project with development dependencies:

```
# Go to module with tests (backend or scraper)
cd backend
cd scraper

# Create venv
python -m venv .venv # UV: uv venv .venv

# Install package with dev dependencies with uv
uv pip install -e . --dev
uv pip install -e ../utils-project

# Or using pip
pip install -e .[dev]
pip install -e ../utils-project

# Activate venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
```

2. Running All Tests

To execute the full test suite with `pytest`:

```
pytest -v
```

3. Running Tests with Coverage

To measure test coverage:

```
pytest --cov=src --cov-report=term-missing
```

4. Running Specific Test Files

For example, to only run health checks:

```
pytest tests/test_health.py -v
```

5. Running Scraper Tests

Navigate to the scraper folder and run:

```
cd scraper
pytest -v
```

REFERENCES

[1] Alalwan, N. A., Alrawashdeh, R. (2024). *Enhancing Tourists' Satisfaction: Leveraging Artificial Intelligence in the Tourism Sector*. ResearchGate.https://www.researchgate.net/publication/381596975_Enhancing_Tourists%27_Satisfaction_Leveraging_Artificial_Intelligence_in_the_Tourism_Sector

[2] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv preprint*, arXiv:1301.3781. https://arxiv.org/abs/1301.3781

[3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint*, arXiv:1810.04805. https://arxiv.org/abs/1810.04805

[4] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[5] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *arXiv preprint*, arXiv:1908.10084. https://huggingface.co/papers/1910.01108

[6] OSRM – Open Source Routing Machine. https://project-osrm.org

[7] Valhalla OpenStreetMap Routing Service. https://github.com/valhalla/valhalla

[8] Google Maps Platform. https://developers.google.com/maps

[9] Mapbox GL JS. https://docs.mapbox.com/mapbox-gl-js

[10] Leaflet.js – Interactive Maps. https://leafletjs.com

[11] OpenLayers. High-performance, feature-rich maps. https://openlayers.org

[12] React – A JavaScript library for building UIs. https://reactjs.org

[13] Vue.js – The Progressive JavaScript Framework. https://vuejs.org

[14] Svelte. Cybernetically enhanced web apps. https://svelte.dev

[15] Selenium. Web Browser Automation. https://www.selenium.dev

[16] Playwright. Fast and reliable end-to-end testing. https://playwright.dev

[17] Puppeteer. (2025). *Headless Chrome Node.js API*. Retrieved from https://pptr.dev

[18] Beautiful Soup. Screen-scraping library. https://www.crummy.com/software/BeautifulSoup

[19] Scrapy. A Fast High-Level Web Crawling Framework. https://scrapy.org

[20] Apache Airflow. Programmatic workflow management. https://airflow.apache.org

[21] Prefect. Dataflow automation and orchestration. https://www.prefect.io

[22] Luigi. A Python module that helps you build pipelines. https://luigi.readthedocs.io

[23] FastAPI. High performance web framework. https://fastapi.tiangolo.com

[24] Flask. Lightweight WSGI web application framework. https://flask.palletsprojects.com

[25] Python Software Foundation. (2025). *Python Programming Language*. Retrieved from https://www.python.org

[26] Django REST Framework. https://www.django-rest-framework.org

[27] Spring Boot. https://spring.io/projects/spring-boot

[28] Express.js. Fast, unopinionated, minimalist web framework. https://expressjs.com

[29] Node.js. (2025). *Node.js JavaScript Runtime*. Retrieved from https://nodejs.org

[30] Docker. Empowering App Development for Developers. https://www.docker.com

[31] Sentence Transformers. Framework for sentence, text and image embeddings. https://www.sbert.net

[32] spaCy. Industrial-strength Natural Language Processing in Python. https://spacy.io

[33] Řehůřek, R., Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. *Proceedings of the LREC 2010 Workshop*. https://radimrehurek.com/gensim/

[34] Google Universal Sentence Encoder. https://tfhub.dev/google/universal-sentence-encoder

[35] OpenAI Embeddings. https://platform.openai.com/docs/guides/embeddings

[36] Tailwind CSS. https://tailwindcss.com

[37] SQLModel. SQL databases in Python, designed for simplicity, compatibility, and performance. https://sqlmodel.tiangolo.com

[38] SQLAlchemy. The Python SQL Toolkit and ORM. https://www.sqlalchemy.org

[39] uv – Fast Python package manager. https://github.com/astral-sh/uv

[40] Poetry. Python packaging and dependency management. https://python-poetry.org

[41] PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Available at: https://www.postgresql.org/

[42] MinIO. High Performance Object Storage. https://min.io

[43] Amazon Web Services. Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/

[44] Scikit-learn: Machine Learning in Python. https://scikit-learn.org

[45] Pydantic. Data validation using Python type annotations. https://docs.pydantic.dev

[46] html2canvas. Screenshots with JavaScript. https://html2canvas.hertzen.com

[47] jsPDF. Client-side JavaScript PDF generation. https://github.com/parallax/jsPDF

[48] Vite. Next Generation Frontend Tooling. https://vitejs.dev

[49] Datos Abiertos de la Junta de Andalucía. https://www.juntadeandalucia.es/datosabiertos

[50] Portal oficial de turismo de Andalucía. https://www.andalucia.org

[51] Wikipedia. Anexo: Playas de Andalucía. https://es.wikipedia.org/wiki/Anexo:Playas_de_Andaluc%C3%ADa

[52] Instituto Andaluz del Patrimonio Histórico (IAPH). https://www.iaph.es

[53] @turf/turf – A modular geospatial analysis engine. https://www.npmjs.com/package/@turf/turf

[54] @mapbox/polyline – A simple Google-esque polyline implementation in JavaScript. https://www.npmjs.com/package/@mapbox/polyline

[55] react-toastify – A React library for easy toast notifications. https://www.npmjs.com/package/react-toastify

[56] lucide-react – A Lucide icon library package for React applications. https://www.npmjs.com/package/lucide-react

[57] Wang, S., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. *arXiv preprint*, arXiv:2002.10957. https://arxiv.org/abs/2002.10957

[58] Berdejo, C. Smart-Itinerary-Generator. https://github.com/cberdejo/Smart-Itinerary-Generator