Home Zone Analyzer

A Comprehensive Software Solution for Real Estate Selection Using Spatial Data and User Preferences

Vittorio Rossetto
Sistemi Context Aware AA. 23/24
Alma Mater Studiorum University, Bologna
Bologna, Italia
vittorio.rossetto@studio.unibo.it

***Abstract*—This paper presents the development of a comprehensive software platform for spatial data management and processing, designed to assist users in selecting real estate properties in Bologna based on personal preferences. The platform consists of a back-end built with Express.js and a front-end developed using React.js, with spatial data handled by a POSTGRES database. Key features include a user questionnaire to capture preferences on Points of Interest (PoIs), interactive map visualizations for properties and areas, and a ranking system for properties and areas based on user preferences and PoI data. This platform aims to streamline the real estate decision-making process through advanced spatial data analysis and visualization techniques.**

I. Introduction

This paper describes the development of a software platform for the management and processing of spatial data, specifically designed to provide recommendations for purchasing real estate in the city of Bologna. The platform comprises a back-end responsible for data management and a front-end offering an interactive interface for end users. The primary objective of the system is to analyze user preferences regarding the proximity and density of various Points of Interest (PoIs) and to provide targeted recommendations on areas and properties that best meet their needs.

II. Project's Architecture

A. General Structure

The application is organized in a frontend-backend architecture, where the frontend serves the interface and handles user interactions and simple calculations, while backend serves data and take care of storing new data in the database.

```
|-- backend
    |-- db.js
    |-- geojson
        |-- PoI_complete.geojson
    |-- index.js
    |-- routes
        |-- lista_aree.js
        |-- lista_immobili.js
        |-- poi_data.js
        |-- survey.js
|-- frontend
    |-- src
        |-- css
            |-- map.css
            |-- survey.css
            |-- index.js
            |-- Map.js
            |-- Menu.js
```

Listing 1. Project's directory structure organization

B. Frontend

The frontend component serves as the user-facing interface, designed to provide a seamless and intuitive experience for interacting with the system's functionalities.

1) Menu.js: This module initiates the user with an interactive questionnaire featuring eighteen questions designed to capture preferences across various categories of Points of Interest (PoI). Users provide ratings on a scale from 0 to 5, indicating the importance of PoIs such as parks, parking facilities, bus stops, and other amenities. Upon submission, the data is securely transmitted to the backend for storage and to the next page for processing. If the user already utilized the application, he can also retrieve the last inserted preferences.

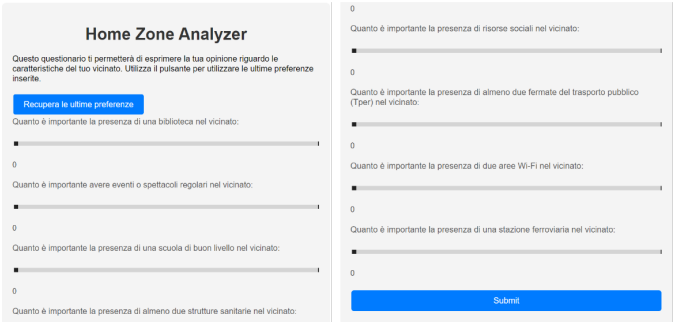

Fig. 1. Interface of the application's first page, the preferences survey.

2) Map.js: Following completion of the questionnaire in Menu.js, users transition to the Map.js interface.

This dynamic page gives the user acces to various functionalities:

a) Interactive Map Visualization: The map interface provides a visually immersive environment where users can view and interact with various PoIs across Bologna, with the simple use of checkboxes to add or remove them from the map. By clicking on any of those points, the platform displays a popup for usefull information: name, type and zone of the PoI.

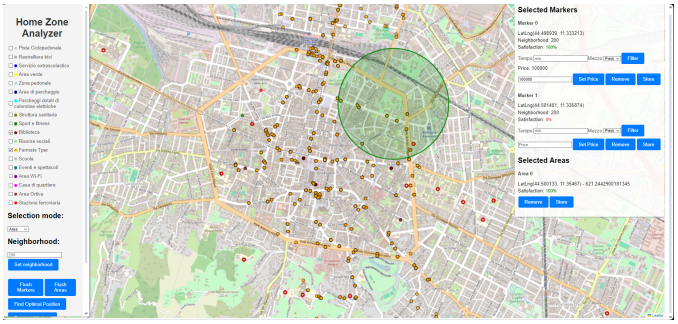

Fig. 2. Interface of the application's second page, displaying the map, dashboard on the left and selected markers or areas on the top right.

b) Marker Interaction: By clicking on the map, users can place markers to denote potential real estate properties. Otherwise, after switching selection mode, select candidate circular areas with two clicks (one for center and one for border). Each marker, or area, is colored based on a satisfaction score calculate based on how the neighborhood reflects user preferences captured in the initial questionnaire. This feature enables users to visually assess how well a property or area aligns with their specific preferences regarding nearby PoIs.

c) Customizable Neighborhood Parameters: Another feature of the platform allows users to define the neighborhood dimension around a marker, directly changing how the satisfaction score is calculated.

d) Optimal Property Location Recommendations: One of the most important features offered by the platform, is the automatic optimal position finding. The user can ask the platform to find the optimal position based on how well it satisfies his previously asserted preferences, and add a marker in that position.

e) PoI Filtering and Reachability: The map interface can also facilitate PoI filtering based on accessibility metrics such as travel time. Users can specify constraints (e.g., reachability within 10 minutes by car or 20 minutes walking) and directly visualize the PoI satisfying those constraints for the selected marker.

f) Moran's I Index Calculation: As users select properties and adjust parameters, the platform is able to computes the Moran's I Index. This statistical measure assesses spatial autocorrelation between property prices and their proximity.

g) Data Storage and Retrieval: All markers for properties and candidate areas, along with associated data, can be stored and retrieved in the PostgreSQL database. This ensures persistent recording of user interactions and data modifications.

h) User Interface Design: The user interface of Map.js prioritizes usability and interactivity. Intuitive controls and responsive design elements facilitate seamless navigation and interaction, ensuring users effectively explore and evaluate properties in Bologna based on personalized criteria and spatial preferences.

C. Backend

The backend component functions as a direct connection to the data storage, providing a fast way to store and retrieve important data.

1) Routes: The route modules define endpoints and operations for managing data related to candidate areas, markers, PoIs, and user survey responses. They enable seamless data retrieval, modification, and storage essential for supporting frontend interactions and calculations.

2) Database: PostgreSQL [4] ensures robust spatial data management capabilities. It efficiently stores and retrieves spatial and non-spatial data types, including user preferences from surveys, geographical coordinates of PoI, and metadata related to candidate properties and areas.

III. Project's Implementation

The application's frontend had been built on ReactJS [2] (javascript framework), and it is organized between two pages: '/' and '/map/'. The backend has been built using Express.js [3] as the primary backend framework, Express.js facilitates robust API development and routing management. It efficiently handles incoming requests from the frontend, orchestrates data processing tasks, and manages interaction with the PostgreSQL [4] database.

A. Menu.js (/)

That is the starting point on entering the application, here the user can express his preferences on PoI, using interactive sliders, then, once he presses a button all the values are stored inside a JSON object, where each element presents this structure:

`poi_type: {value: num, count: num}`

where `poi_type` is the same identifier used in PoI data, `value` is the inserted preference value from 0 to 5 and `tcount` is the number of PoIs of the given type in the neighborhood necessary to satisfy the condition. This object is then sent to '/map' and to the backend at `/api/data/preferences` once the user submits the survey. Otherwise if there is already stored data about preferences, the user can use a button to navigate to '/map' and utilize the last inserted preferences. Last preferences are retrieved by the backend via the query

```
SELECT * FROM preferences ORDER BY id DESC LIMIT 1
```

B. Map.js (/map)

The application relies on the react hook useEffect() to initialize the map of Bologna with the use of Leaflet [1] and display it, the moment the page starts. In the same way, it retrieves data relative to PoI from the database via a fetch call GET to the backend at the route '`/api/data/poi_data`', and once those are retrieved, a menu is created with a list of check boxes (one for each type of PoI) on check each of those mounts a new layer on the map to show the relative PoIs. Each PoI is stored as a layer addable to the map with coordinates, style and a leaflet popup to display the relative

data such as position, type and belonging area. The entirety of data relative to PoIs were obtained from a dataset [7] in the platform OpenData of the city of Bologna.

1) Selection on map: The platform offers the possibility to directly interact with map, with two different selection modes: marker and area. The user can switch mode from the dashboard, the click event listener is mounted on map in the moment formData (so data relative to the user survey) are retrieved.

a) Marker: Marker is the default selection mode its function take care of creating a new circle marker with the coordinates of the click, and a color based on satisfaction rate. To assign the color:

- The application pass through all the types of PoIs to find the ones the user has an interest in (ones with expressed value bigger than 0)
- Each point of those types is checked to see if it's inside the given neighborhood (default is 200 meters from marker)
- If so, the count of satisfying points is increased
- If the necessary count is reached, this type of PoI is checked as satisfied.
- Once this has been done for each interesting type, the value (expressed user during the survey) of each satisfied one is summed up.
- The satisfaction is converted to a percentage with this proportion:

$$\frac{S}{S_{\text{max}}} = \frac{S_{\text{p}}}{100}$$

  where S is the reached satisfaction rate, S_{max} is the sum of all the preferences values, and S_{p} is the satisfaction percentage.
- The color is assigned based on S_{p}
  - $0 \leq S_p \leq 25$ means red is assigned
  - $25 < S_p \leq 50$ means orange is assigned
  - $50 < S_p \leq 75$ means yellow is assigned
  - $75 < S_p \leq 100$ means green is assigned

Once a marker is created, it gets a popup to show its properties on click, and a custom function to show its neighborhood as a circle on hover.

b) Area: The second type of selection is area, when using this mode the user can create each area with two clicks, one for the center and one for the circle border. The system creates a circular area, with the first click position as center and the distance between the two clicks as radius, the area is the colored with the same method used for markers, with the sole difference that it uses area radius instead of neighborhood for evaluation.

2) Neighborhood dimension: The neighborhood dimension can be set from the application's dashboard, the default value is 200 meters in radius, once it is set the value will be used for all the following points until further modifications.

3) Optimal position recommendation: One of the most important application's functionalities, is the one offered by optimal position recommendation, to find it the process develops along two steps:

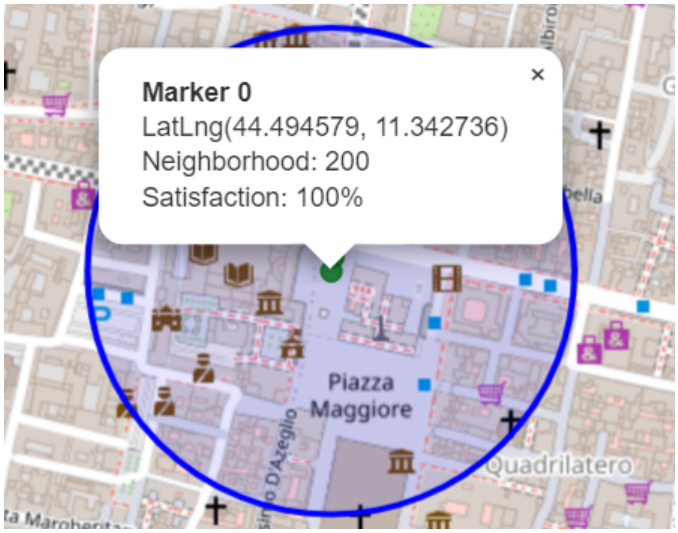

Fig. 3. Example of a marker on hover with open popup

a) Creation of the grid: First, the application creates a grid of points covering the whole city. To do this:

- Given the center of the city (the coordinates used to visualize the map), it selects the zone to create the grid on by calculating:
  - A starting latitude (`center - distance`)
  - An ending latitude (`center + distance`)
  - Where `distance` is the distance between the center and the border of the city.
- It then iterates over this distance with a step decided by the density.
- For each latitude:
  - It calculates the maximum longitude difference at that latitude to maintain the grid shape. This is necessary because the distance between longitudes changes with latitude due to the Earth's curvature. This is done using the formula:

$$\text{maxLngDiff} = \frac{\text{distance}}{\cos(\text{latitude} \cdot \frac{\pi}{180})}$$

    where `maxLngDiff` is the maximum longitude difference.
  - It then iterates from the minimum to the maximum longitude values at that latitude:
    - Starting from `center - maxLngDiff`
    - Ending at `center + maxLngDiff`
    - Using a step decided by the density
  - For each step, it generates a point at the current latitude and longitude
  - All generated points are then added to an array.

b) Calculation of optimal: The application iterates over all the generated points, and calculates satisfaction percentage for each point using the same function used to assign color to markers. If a certain points reaches 100% satisfaction the loop is stopped and the point is identified as optimal, otherwise the

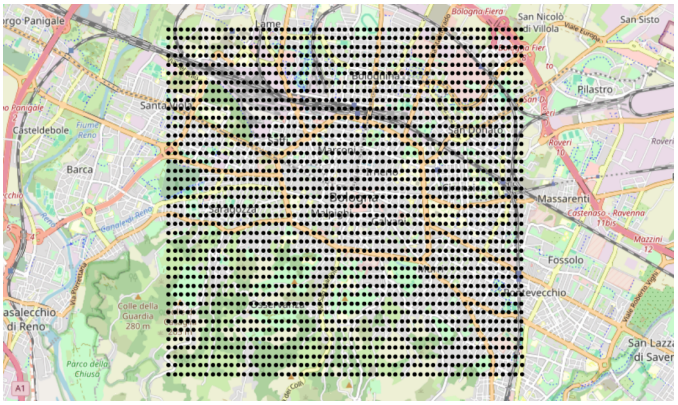

Fig. 4. Graphic representation of the generated points grid (in this picture has been used a density of 20 for better display, but the application actually uses a density of 100 for better precision)

optimal position is updated anytime a point reaches a higher satisfaction rate. Once the optimal position is found a marker with those coordinates is added to the map and visualized as other markers.

4) Moran's I Index: The platform offers the possibility of calculating the Moran's I index. This serves the purpose of measuring the spatial auto correlation of property prices based on their geographic locations and defined neighborhood sizes. The weight matrix is constructed such that weights are 1 if the distance between properties is within the neighborhood radius, otherwise, the weight is 0. This index helps understand if there is a pattern or clustering of high or low property prices in specific areas. To calculate the index, the user first has to indicate price for the inserted markers, directly in the platform, then it is calculated over the correlation between the inserted price and the satisfaction percentage used as weight, digging deeper into this:

- *Data gethering:* First all the markers with a given price are identified and their price, position and neighborhood are gathered in three arrays
- *Spatial weight matrix:* To fill the weight matrix, it iterates over all the gathered properties, each element weightMatrix[i][j] is set to 1 if the distance between the positions of the i-th and j-th properties is less than the neighborhood size of the i-th property. Otherwise, it is set to 0. This means the weight is 1 if the j-th property is within the neighborhood of the i-th property.
- Numerator for Moran's I: The numerator is calculated by summing the product of the weights and the deviations for each pair of properties.
- *Denominator for Moran's I:* The denominator is the sum of the squared deviations.
- *Calculate Moran's I Index:* Moran's I index is calculated using the formula:

$$I = \frac{N}{S_0} \cdot \frac{\sum_i \sum_j w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i (x_i - \bar{x})^2} \qquad (1)$$

where:

I : Moran's I index
N : Total number of spatial units
S_0 : Sum of all weights in the weight matrix
w_{ij} : Spatial weight between units i and j
x_i : Value of the variable of interest for unit i
$\bar{x}$: Mean of the variable of interest across all units

5) Proximity based on Travel Time: Once a marker is added to the platform the user can set a filter for the PoIs based on reachability within a given time, using a certain mode of transport (driving, cycling or walking). Once the filter is applied the points satisfying those conditions are displayed on map. The process to achieve this result the process develops along two steps:

- *Time distance calculation:*
  - First, the application pass through all the types of PoIs to find the ones the user has an interest in (ones with expressed value bigger than 0)
  - The platform calculates the travel time from the marker to each of those PoI using the MapBox routing API [8].
  - It constructs the appropriate URL using:
    - The given API access token
    - The marker's coordinates
    - The PoI's coordinates
    - The mode of transport
    - So the URL would look like:

```
https://api.mapbox.com/directions/v5
/mapbox/driving/{marker
coordinates};{PoI coordinates}
?access_token=YOUR_MAPBOX_ACCESS_TOKEN
```

  - It then sends a GET request to the constructed URL to retrieve the travel time.
- *Filtering and displaying:* When a travel time is calculated the platform checks if its less then the inserted maximum, if so, the PoI is displayed and its popup is modified so that it can also display the travel time to reach it.

The filtered point's layers can then be removed with the click of a button.

C. Data Storage and Retrieval

All the data relative to markers or areas can be stored in the postgreSQL database so a user can keep certain data even if he closes the application. Since all the data are handled the same way the explanation will only comprehend markers storage and retrieval to avoid repetition.

Explaining this better for markers (the explanation for areas would be the same, so only one will be disserted to avoid repetition):

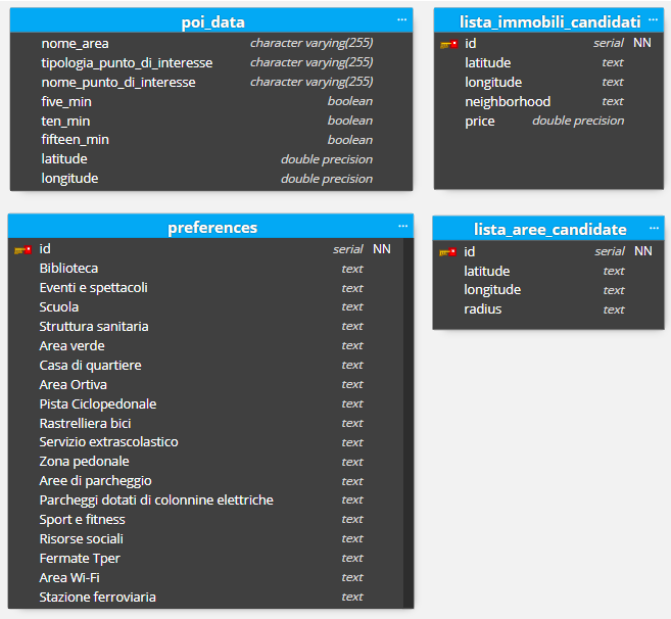

Fig. 5. Visual representation of the application's database schema

1) Storage:

- *Frontend:* To store a certain marker, the frontend creates a JSON object with all the marker's informations, then fetches a POST request to the backend at 'api/-data/lista_immobili' with the previously created JSON as body
- *Backend:* Once the backend has received the request, a callback function handles it.
  - The function extracts all the values from the request body and identifies corresponding keys. These keys will serve as column names for the database table.
  - The function checks if the database table exists. If it doesn't exist, the function creates it. To create the table named `lista_immobili_candidati`, the application constructs a query. This table includes:
    - A primary key column named `id` that auto-increments (SERIAL).
    - Additional columns corresponding to the keys in the request body, all defined as TEXT type.
    - The final query is built with this structure:
      ```
      CREATE TABLE IF NOT EXISTS
      lista_immobili_candidati
      (id SERIAL PRIMARY
      KEY,${columns.map(column =>
      `"$column" TEXT`).join("")})
      ```
  - Once the platform ensures the table exists, it proceeds to build a query to insert a new record into the table using the provided data:
    `INSERT INTO lista_immobili_candidati (${Columns Names}) VALUES ({values Array}) RETURNING *`

.

2) Retrieval:

- *Frontend:*The frontend to retrieve the stored marker needs to send a fetch `GET` request to `'api/data/lista_immobili'`, which the backend handles, and respond by sending back all the stored markers in a JSON file as response. Once those data are recieved from frontend, it bulds a marker for each of the retrieved elements, using its data and adds it to the map. The retrieved markers than behave as all the other markers.
- *Backend:*When the backend receives a GET request, it procedes to putting together all the records of the table `lista_immobili_candidati` by executing the SQL query:
  `SELECT * FROM lista_immobili_candidati`
  to fetch all rows from the table, then putting them in a JSON object and sending it back as a response.

3) Routes: The route modules, built with the use of Express.js [3], define endpoints and operations for managing datasets related to candidate areas, the selected positions, PoI data, and user survey responses.

- lista_aree.js `/api/data/lista_aree`: Takes care of database operations with areas, stored in the table `lista_aree_candidate`
- lista_immobili.js `/api/data/lista_immobili`: Takes care of database operations with markers, stored in the table `lista_immobili_candidati`
- poi_data.js `/api/data/poi_data`: Takes care of database operations with PoIs, stored in the table `poi_data`
- survey.js `/api/data/survey`: Takes care of database operations with preferences, stored in the table `preferences`

D. Deployment Configuration

1) YAML Files: (Contained in ./k8s): These files specify deployment configurations for each component within the Kubernetes [6] cluster. They define container images, resource allocations, networking settings, and other parameters necessary for ensuring reliable performance and high availability.

2) k8sbuilder.sh: This shell script automates Kubernetes deployment tasks, streamlining cluster initialization, application component deployment, networking configuration, and persistent storage management (PVC - Persistent Volume Claims).

E. Utilized technologies

The application development required the employment of several technologies, each contributing uniquely to the functionality and performance of the system.

1) Frontend:

a) User Interface Development (React.js): React.js [2] was used for building the user interface of the application. Its component-based architecture and virtual DOM provide a fast and efficient way to build dynamic and interactive user interfaces. This allowed for the creation of a responsive

and interactive front-end, including features such as the user questionnaire.

b) Map Visualization and Interaction (Leaflet): Leaflet [1] was employed for map visualizations and interactions. As a lightweight, open-source library for interactive maps, it provides a simple API and supports various map layers and markers. Leaflet facilitated the display of properties and points of interest on the map, enabling users to interact with the geographical data intuitively.

2) Backend:

a) RESTful API (Express.js): Express.js [3] was used as the backend framework for building the RESTful API. It provides a simple and flexible way to handle routes, middleware, and request handling, making it well-suited for developing scalable server-side applications. Express.js facilitated efficient handling of HTTP requests, integration with the PostgreSQL database, and implementation of the API endpoints for managing spatial data and user preferences.

b) Data storage (PostgreSQL): : PostgreSQL [4] was used as the relational database management system. PostgreSQL offers robust and reliable database management. This technology enabled efficient storage, querying, and manipulation of spatial data, crucial for the project, such as points of interest and property locations.

3) Containerization and Orchestration:

a) Containerization (Docker): Docker [5] was used for containerizing the application components. It allows for consistent development and deployment environments, ensuring that the application runs reliably across different setups. Docker simplified the deployment process by packaging the backend, frontend, and database into separate containers, ensuring consistency and reducing configuration issues.

b) Orchestration (Kubernetes): Kubernetes [6] was utilized for orchestrating the Docker containers. It provides powerful orchestration features, including automated deployment, scaling, and management of containerized applications. Kubernetes managed the deployment and scaling of the application components, ensuring high availability and resilience of the system.

4) External APIs:

a) Travel Time Calculation (Mapbox Directions API): The Mapbox Directions API was integrated for calculating travel times between points. It provides accurate and detailed routing information, supporting various modes of transportation. By leveraging the Mapbox Directions API, the platform's functionality was enhanced, allowing users to filter points of interest based on travel time and receive more personalized and practical recommendations.

IV. Results

This project set out to develop a comprehensive software platform for spatial data management and processing, aimed at assisting users in selecting real estate properties in Bologna based on personal preferences. The primary objectives were to create a user-friendly interface for inputting preferences, visualize properties and points of interest on an interactive map, and provide ranked recommendations based on user-defined criteria. Key achievements include the successful integration of advanced spatial data analysis and visualization techniques, the implementation of a robust backend using Express.js and a dynamic frontend with React.js, and the effective management of spatial data through a POSTGRES database. The platform's core features, such as the user questionnaire, map visualizations, property ranking, and Moran's I index calculation, were all realized effectively, contributing to the system's overall functionality. The project met its initial goals, demonstrating that the platform can significantly aid users in making informed real estate decisions by offering detailed analyses and visualizations of proximity to preferred amenities. Despite certain challenges, such as optimizing computational efficiency and handling large datasets, the platform proved to be a valuable tool for spatial data analysis. In conclusion, this project illustrates the potential of combining spatial data analysis with user preference modeling to create a powerful tool for real estate decision-making. The developed platform not only simplifies the process of selecting a property but also highlights the effectiveness of modern web technologies in managing complex spatial datasets.

A. Future Improvements and directions

While the current platform offers robust functionality for selecting real estate properties in Bologna based on user preferences and spatial data analysis, several future improvements and expansions can enhance its utility and user experience. These include expanding geographic coverage, integrating real-time property data, and adding features to improve user experience.

a) Expanding Geographic Coverage: One of the primary future improvements involves expanding the platform beyond Bologna to other cities and regions. This would entail integrating spatial data from various portals for new locations, including points of interest (PoIs), and other geographical data. The user interface will be updated to allow users to select their desired city or region, tailoring property recommendations to their specific location.

b) Integrating real-time property data: This could be another crucial enhancement. By providing users with up-to-date information on properties for sale, the platform can significantly increase its value. This could involve collaborating with real estate agencies and listing platforms to access real-time property data, including price, size, amenities, and availability. APIs will be developed to fetch and update property data in real-time. Consequently, the user interface will be enhanced to display real-time property details alongside position recommendations.

c) Street View and Other Visualization Tools: To further enrich the user experience, adding a street view feature is proposed. This will enable users to virtually explore neighborhoods and properties. The implementation of interactive elements, such as hotspots for PoIs and property highlights, could be included to make the experience more engaging. Additionally, developing sophisticated visualization

tools, such as heat maps for property values and dynamic charts for market trends, will provide users with deeper insights.

By implementing these future improvements, the platform can significantly enhance its value proposition to users. Expanding geographic coverage will make the platform relevant to a broader audience, integrating real-time property data will provide comprehensive information, and adding street view and othe visualization tools will enrich user experience.

REFERENCES

[1] Joe Cheng and Barret Schloerke and Bhaskar Karambelkar and Yihui Xie. “Leaflet - an open-source JavaScript library for mobile-friendly interactive maps“, Available: https://leafletjs.com/reference.html. [Accessed: July 12, 2024]

[2] Facebook, Inc., “React: A JavaScript library for building user interfaces,“ Available: https://reactjs.org/. [Accessed: July 12, 2024].

[3] OpenJS Foundation, “Express - Node.js web application framework,“ Available: https://expressjs.com/. [Accessed: July 12, 2024].

[4] PostgreSQL Global Development Group, “PostgreSQL: The world’s most advanced open source database,“ Available: https://www.postgresql.org/. [Accessed: July 12, 2024].

[5] Docker, Inc., “Docker Documentation,” Available: https://docs.docker.com/. [Accessed: July 12, 2024].

[6] Craig McLuckie, Joe Beda and Brendan Burns, “Kubernetes Documentation,“ Available: https://kubernetes.io/docs/. [Accessed: July 12, 2024].

[7] Comune di Bologna, “Workshop BCN: Mappatura POI Completa Approfondimento Zone di Interesse,” Available: https://opendata.comune.bologna.it/explore/dataset/workshop-bcn-mappatura-poi-completa-approfondimento-zone-di-interesse/information/. [Accessed: July 12, 2024].

[8] Mapbox, “Mapbox routing API,” Available: https://www.mapbox.com/. [Accessed: July 12, 2024].