POLITECNICO
MILANO 1863

Software Engineering for Geoinformatics

(A.Y.2024)

DESIGN DOCUMENT

AUPE RiskMonitor
RISK RADAR

Authors:

Mohammad Ammar Mughees

Mohmmad Umayr Romshoo

Sayed Erfan Eshghelahi

Praveenkumar Saminathan

1. Introduction

1.1 Design Document

Before beginning the coding and implementation of software, it's crucial to establish clear design goals for the web application. These goals should be detailed in a Design Document. This document defines the structure and organization to be followed throughout the project, helping to prevent confusion and errors. It serves as a critical guideline for the engineering team, ensuring everyone understands the project's direction. The Software Design Document is a technical guide for both developers and clients, aiming to meet client objectives while facilitating an efficient workflow for the development team.

We should remember that the Design Document should function as a requirements and functionality guide, not an implementation specification. The specified characteristics should remain consistent and serve as implicit knowledge for the development team. At a minimum, the document should describe the application, criteria for completion, and milestone.

1.2 Purpose:

In today's world, where we are overwhelmed with information and facing global crises, there is a critical need for easy-to-access, high-quality information. People increasingly want accurate information, so we need tools that help them check data from trustworthy, unbiased sources. These tools can help individuals support their concerns when negotiating with governments and other important groups, encouraging democratic participation.

The AUPE Project aims to develop an interactive application to support disaster management activities during the prevention and preparedness phases, focusing on flood and landslide hazards in Italy. Public authorities in Italy collect, process, and analyze hazard-related datasets to produce detailed hazard maps. These maps, when combined with population and building census data, enable the mapping and visualization of exposure to relevant hazards. This project leverages these datasets to create a tool that aids decision-makers, such as civil protection authorities and insurance companies, in effectively analyzing and visualizing this critical information.

The core of the project is a client-server application that integrates data from the Italian Institute for Environmental Protection and Research (ISPRA) IdroGEO API, particularly the Hazards and Risk Indicators (PIR). The application is designed to provide users with a robust platform for querying, processing, and visualizing hazard exposure data, thereby enhancing disaster preparedness and risk mitigation efforts.

1.3 Scope

Considering the risks of landslides, the planned application aims to offer a personalized tool that provides easy access to relevant data. This tool is designed to help users in their personal lives and as active members of a democratic society. The application will help visualize existing data sources rather than act as an independent information store. The following sections will describe the target users, situations in which they might use the app, and the events related to its use. Additionally, specific requirements are listed to guide the app's development.

1.4 Project Overview:

The application is composed of three main components:

- Database**:** A centralized system to store and manage hazard and census data ingested from the ISPRA IdroGEO API. This database is optimized for handling spatial data and supports efficient querying and analysis.
- Web Server (Backend): A RESTful API, developed using Flask, that serves as the intermediary between the database and the client application. The web server handles data preprocessing and cleaning before returning results to users in a structured JSON format.
- Interactive Dashboard (Frontend): Built using Jupyter Notebooks, this component provides an intuitive interface for users to interact with the data. The dashboard supports various data manipulation and visualization techniques, including maps and interactive graphs, allowing users to generate customized views and insights.

1.5 Objectives:

The main objectives of the project are:

- Data Integration: To seamlessly ingest and preprocess hazard and census data from the ISPRA IdroGEO API into the database.
- Efficient Data Retrieval: To design and implement a REST API that facilitates efficient querying and retrieval of data from the database.
- Interactive Visualization: To develop a user-friendly dashboard that allows users to process and visualize data dynamically, enhancing their ability to analyze hazard exposure.
- Agile Development: To employ an Agile-like software development process, ensuring iterative development, regular feedback, and continuous improvement.

1.6 Definitions:

• “The web app” or “app” – The application that is being developed
• “Use case” – An example of user interaction with the system
• “User” – A person using the website that satisfies the description under the “user characteristics” section.
• “Bookmark” – Saving a snapshot of the information presented at a location on the map
• JSON – JavaScript Object Notation
• JS – JavaScript 5
• df - Data frame, a pandas data frame object
• gdf – geodata frame, a geopandas data frame object
• RASD – Requirements Analysis and Specification Document easy later access.
• “API” – Application Programming Interface
• “OSM” – Open Street Map. A database with open access geographical data.

2. Architectural Design:

2.1 Overview

The web app is built on a flask app at the bottom which collects data from the API needed, runs needed calculations, formats output data and serves it to the web. The goal of this product is to inform and engage users about landslides which threatened buildings in region of Italy through web application. This application will enable users to query, visualize, and analyze data, as well as save their analyses. The website will also provide general information on landslides and allow users to comment on data analyses.

2.2 Project database:

Data for this project, including point positioning and indicators of risks of landslides, will be retrieved from the API provided by **https://idrogeo.isprambiente.it**. We will preprocess this data before making it available to users. The web application will interact with a DBMS, specifically using PostgreSQL, to store user requests and enhance the application's quality.

2.3 Postgres database:

We decided to store our retrieved data on an AWS RDS PostgreSQL database for several reasons. PostgreSQL was chosen due to its robust support as an open-source relational database management system. It provides native support for georeferenced data operations through PostGIS, which is crucial for our needs. PostGIS enables us to work with geometry data and perform spatial operations such as buffering, unioning, and clipping. Moreover, it facilitates spatial queries such as measuring intersections, areas, and lengths, which are essential for our spatial data analysis tasks.

Additionally, utilizing AWS RDS allows us to leverage the benefits of a cloud-based database. By hosting our PostgreSQL database on AWS RDS, we ensure that our data is accessible from anywhere with an internet connection. This flexibility is crucial for our team and stakeholders who need to access and interact with the data regardless of their physical location.

2.4 Component Diagram:

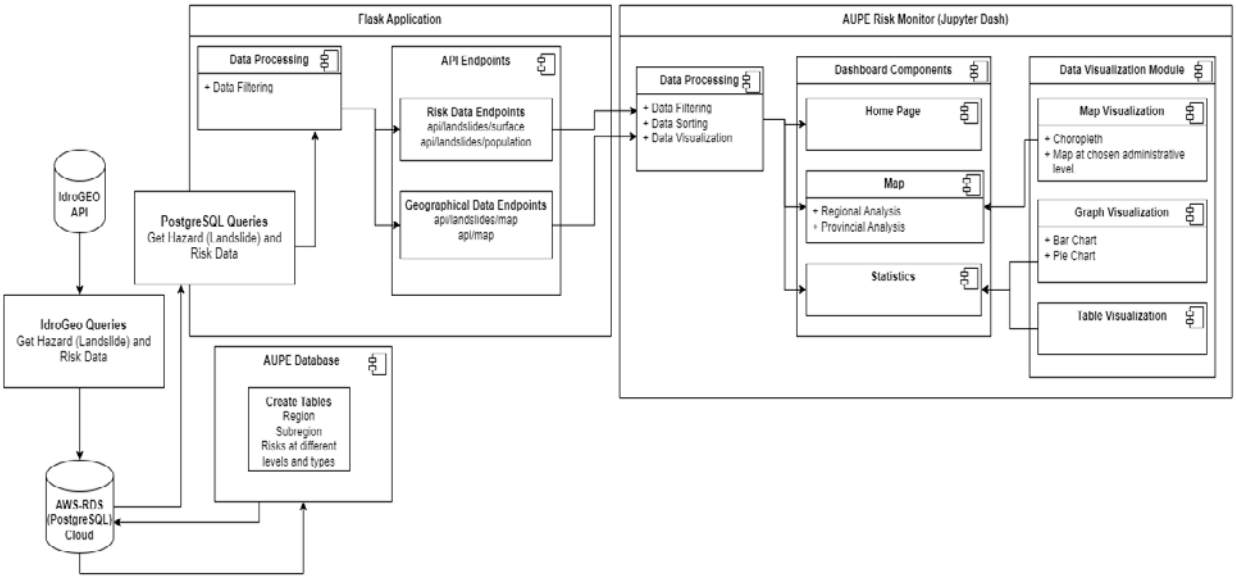


Figure 1: Component Diagram

3. Software Design:

3.1 Software Architecture

The AUPE application exploits the widespread client-server architectural paradigm, were client software requests services and resources from server software. The server software processes incoming client requests and responds with the requested service or resource. This client-server architecture arises from the need for modern software applications to be modular rather than monolithic. Modern applications are composed of autonomous, specialized software components that interact to provide the final client (the user) with the requested services or resources.

The modularity of software is both functional and logical/physical. Functional modularity means different functionalities are provided by diverse software components. Logical modularity refers to the design of the application as a multi-tier system, with multiple logical layers (tiers) each dealing with specific tasks. Physical modularity refers to the execution of software components on different machines connected through a network.

AUPE RiskMonitor is conceived as a 3-tier client-server application composed of:

Data Layer/Tier: A database server where data is ingested and stored. This tier is responsible for data management and retrieval.

Logic Layer/Tier: A web server (backend) that exposes a REST API used for querying the database and retrieving data. This tier handles business logic and data processing.

Presentation Layer/Tier: A dashboard that interacts with the web server to request, process, and visualize data (e.g., maps, dynamic graphs). This tier provides the user interface for data visualization and interaction.

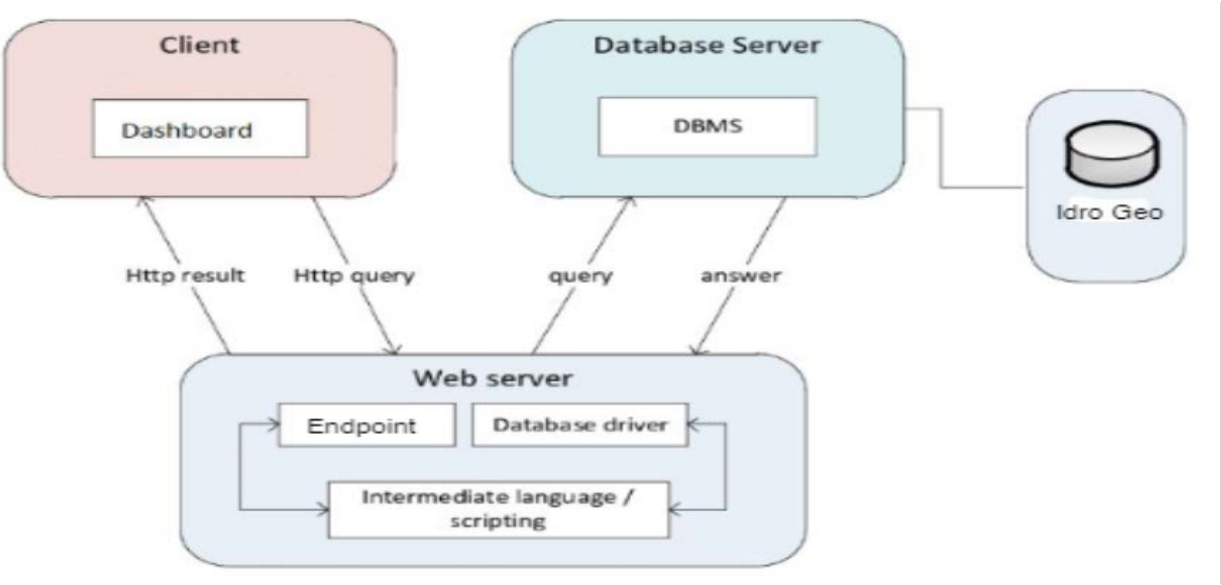


Figure 2: Architecture Diagram

3.2 Database Design:

Purpose:

The database is designed to store and manage landslide risk data for the Lombardy region and its provinces. It facilitates efficient data retrieval and supports the generation of geographic visualizations and analyses in the AUPE RiskMonitor dashboard.

Components Overview:

➢ Database Configuration
➢ Data Fetching and Preprocessing
➢ Schema Definition
➢ Data Insertion
➢ Database Connection and Session Management

3.2.1. Database Configuration

The database connection configuration, which includes the database name, user credentials, and host details, is defined in a dedicated configuration file named `config.py`. Users do not need to interact with this file as it has already been pre-configured.

```
from sqlalchemy import create_engine, Column, Integer, String, Float
from config import db_name, user, password, host
```

- create_engine: Function from SQLAlchemy to create a new database connection.
- Column, Integer, String, Float: SQLAlchemy classes used to define table columns.
- config: A custom module where the database configuration parameters are stored.

3.2.2. Data Fetching and Preprocessing

This component fetches landslide risk data from the ISPRA Idrogeo API and preprocesses it for database insertion.

```
import requests
import json
import pandas as pd
import geopandas as gpd

# Fetch and convert data from the ISPRA Idrogeo API for the Lombardy region and
its provinces into DataFrames.
def jason2dataf(url):
    response = requests.get(url)
    raw_data = response.text
    data = json.loads(raw_data)
    data_df = pd.json_normalize(data)
    return data_df

# Fetch data for Lombardy and its provinces
dataf_lombardia =
jason2dataf('https://test.idrogeo.isprambiente.it/api/pir/regioni/3')
data_prov_milan =
jason2dataf('https://test.idrogeo.isprambiente.it/api/pir/province/15')
# (Repeat for other provinces...)

# Merge the DataFrames
merged_df = pd.concat([data_prov_milan, ...]).reset_index(drop=True)
merged_df.rename(columns={'cod_prov':'COD_PROV'}, inplace=True)

# Load GeoData for the region and provinces
shapefile_path_region =
"./Limiti01012024_g/Reg01012024_g/Reg01012024_g_WGS84.shp"
```

```
shapefile_path_province =
"./Limiti01012024_g/ProvCM01012024_g/ProvCM01012024_g_WGS84.shp"
gdf_region = gpd.read_file(shapefile_path_region)
gdf_province = gpd.read_file(shapefile_path_province)

# Filter GeoDataFrame based on desired COD_PROV values
desired_cod_RIP = [15, 12, 13, ...]
filtered_gdf = gdf_province[gdf_province['COD_PROV'].isin(desired_cod_RIP)]

# Merge GeoDataFrame with the merged data
geometry = filtered_gdf.pop('geometry')
filtered_gdf3 = gpd.GeoDataFrame(filtered_gdf, geometry=geometry)
merged_df3 = pd.merge(filtered_gdf3, merged_df, on='COD_PROV', how='left')
merged_gdf = gpd.GeoDataFrame(merged_df3, geometry='geometry')
```

➢ requests.get(url): Fetches data from the specified API URL.
➢ json.loads(raw_data): Converts JSON string data into a Python dictionary.
➢ pd.json_normalize(data): Converts JSON data into a pandas DataFrame.
➢ gpd.read_file(shapefile_path): Reads geospatial data from shapefiles into GeoDataFrames.
➢ pd.concat([...]): Concatenates multiple DataFrames into one.
➢ pd.merge(...): Merges two DataFrames based on a common column.

3.2.3. Schema Definition

Defines the structure of the dataset table in the database using SQLAlchemy ORM. This includes various attributes related to landslide risk and a column for geometric data in WKT format.

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Float, Text
from geoalchemy2 import Geometry

Base = declarative_base()

class Dataset(Base):
    __tablename__ = 'dataset'

    id = Column(Integer, primary_key=True)
    cod_reg = Column(Integer)
    cod_rip = Column(Integer)
```

```
    cod_prov = Column(Integer)
    ar_kmq = Column(Float)
    nome = Column(String)
    uid = Column(Integer)
    ar_fr_p3p4 = Column(Float)
    ar_fr_p2 = Column(Float)
    ar_fr_p1 = Column(Float)
    ar_fr_p3 = Column(Float)
    ar_fr_p4 = Column(Float)
    ar_fr_aa = Column(Float)
    ar_frp3p4p = Column(Float)
    ar_frp4_p = Column(Float)
    ar_frp3_p = Column(Float)
    ar_frp2_p = Column(Float)
    ar_frp1_p = Column(Float)
    ar_fraa_p = Column(Float)
    pop_fr_p2 = Column(Float)
    pop_fr_p1 = Column(Float)
    pop_fr_p3 = Column(Float)
    pop_fr_p4 = Column(Float)
    popfr_p3p4 = Column(Float)
    pop_fr_aa = Column(Float)
    popfrp4_p = Column(Float)
    popfrp3_p = Column(Float)
    popfrp2_p = Column(Float)
    popfrp1_p = Column(Float)
    popfrp3p4p = Column(Float)
    popfraa_p = Column(Float)
    ed_fr_p4 = Column(Float)
    ed_fr_p3 = Column(Float)
    ed_fr_p2 = Column(Float)
    ed_fr_p1 = Column(Float)
    ed_fr_p3p4 = Column(Float)
    edfrp3p4p = Column(Float)
    geometry = Column(Text)  # Geometry as WKT

    def __repr__(self):
        return f"<Dataset(id={self.id}, nome='{self.nome}')>"
```

- declarative_base(): Creates a base class for declarative class definitions.
- Column(...): Defines a column in the table with its data type and constraints.
- Geometry: GeoAlchemy2 class to handle geometric data.

3.2.4. Data Insertion:

This component handles the conversion of geometric data into WKT format and inserts the preprocessed data into the database.

```
from shapely.wkb import loads as wkb_loads
from shapely.wkt import dumps as wkt_dumps

# Function to convert WKB to WKT
def convert_wkb_to_wkt(geom):
    if isinstance(geom, str):
        geom = bytes.fromhex(geom)
    if isinstance(geom, (bytes, bytearray)):
        return wkt_dumps(wkb_loads(geom))
    elif isinstance(geom, (MultiPolygon, Polygon)):
        return wkt_dumps(geom)
    else:
        raise TypeError(f"Unexpected geometry type: {type(geom)}")

# Apply conversion function to the 'geometry' column
selected_df2['geometry_wkt'] = selected_df2['geometry'].apply(convert_wkb_to_wkt)

# Insert data into the database
for index, row in selected_df2.iterrows():
    dataset_entry = Dataset(
        cod_reg=row['cod_reg'],
        cod_rip=row['cod_rip'],
        cod_prov=row['COD_PROV'],
        ar_kmq=row['ar_kmq'],
        nome=row['nome'],
        uid=row['uid'],
        ar_fr_p3p4=row['ar_fr_p3p4'],
        ar_fr_p2=row['ar_fr_p2'],
        ar_fr_p1=row['ar_fr_p1'],
        ar_fr_p3=row['ar_fr_p3'],
        ar_fr_p4=row['ar_fr_p4'],
        ar_fr_aa=row['ar_fr_aa'],
        ar_frp3p4p=row['ar_frp3p4p'],
        ar_frp4_p=row['ar_frp4_p'],
        ar_frp3_p=row['ar_frp3_p'],
        ar_frp2_p=row['ar_frp2_p'],
        ar_frp1_p=row['ar_frp1_p'],
        ar_fraa_p=row['ar_fraa_p'],
        pop_fr_p2=row['pop_fr_p2'],
        pop_fr_p1=row['pop_fr_p1'],
```

```
        pop_fr_p3=row['pop_fr_p3'],
        pop_fr_p4=row['pop_fr_p4'],
        popfr_p3p4=row['popfr_p3p4'],
        pop_fr_aa=row['pop_fr_aa'],
        popfrp4_p=row['popfrp4_p'],
        popfrp3_p=row['popfrp3_p'],
        popfrp2_p=row['popfrp2_p'],
        popfrp1_p=row['popfrp1_p'],
        popfrp3p4p=row['popfrp3p4p'],
        popfraa_p=row['popfraa_p'],
        ed_fr_p4=row['ed_fr_p4'],
        ed_fr_p3=row['ed_fr_p3'],
        ed_fr_p2=row['ed_fr_p2'],
        ed_fr_p1=row['ed_fr_p1'],
        ed_fr_p3p4=row['ed_fr_p3p4'],
        edfrp3p4p=row['edfrp3p4p'],
        geometry=row['geometry_wkt']  # Insert
```

3.2.5. Database Connection and Session Management

The database connection is established using SQLAlchemy's create_engine. Session management is handled to ensure data integrity and proper resource management.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import psycopg2

# Create an engine and connect to the PostgreSQL database
engine =
create_engine(f'postgresql+psycopg2://{user}:{password}@{host}:5432/{db_name}')
Base.metadata.create_all(engine)

# Create a session to interact with the database
Session = sessionmaker(bind=engine)
session = Session()

# Connect to the default database to create the new database
conn = psycopg2.connect(dbname='postgres', user=user, password=password,
host=host)
conn.autocommit = True
cur = conn.cursor()
```

```
# Create the new database if it does not exist
try:
    cur.execute(f"CREATE DATABASE {db_name}")
except psycopg2.errors.DuplicateDatabase:
    print(f"Database {db_name} already exists.")

cur.close()
conn.close()
```

Database Schema:

Table: **dataset**

The dataset table includes the following columns:

id:	Primary key auto-increment integer.
cod_reg:	Integer, corresponds to the region code.
cod_rip:	Integer, possibly related to regional planning or administrative codes.
cod_prov:	Integer, province code.
ar_kmq:	Float, area in square kilometers.
nome:	String, name.
uid:	Integer, unique identifier.
Various columns (ar_fr_p3p4, ar_fr_p2, etc.):	Float, representing different area fractions.
Various columns (pop_fr_p2, pop_fr_p1, etc.):	Float, representing population fractions.
Various columns (ed_fr_p3, ed_fr_p4, etc.):	Float, representing educational fractions.
geometry:	Text, stores the WKT representation of geometry.

- **Geometry Storage**: Geometry data is stored as WKT (Well-Known Text) in a Text column named geometry.
- **Database Connection**: The SQLAlchemy create_engine function is used to connect to the PostgreSQL database specified in your config.py.
- **Table Creation**: Base.metadata.create_all(engine) creates the necessary table (dataset) in the specified PostgreSQL database.
- **Data Insertion**: Data from the processed GeoDataFrame (selected_df2) is inserted row by row into the dataset table using SQLAlchemy's ORM (session.add() and session.commit()).

3.3 REST APIs

General Description:

REST APIs (Representational State Transfer Application Programming Interfaces) provide a standardized way for systems to communicate over HTTP, enabling efficient data exchange between client applications and servers. They are commonly used in web development to facilitate interoperability and integration across different platforms.

The REST APIs for the AUPE project are designed to provide access to hazard-related data, including landslide and flood information, for various purposes such as mapping, analysis, and visualization. The APIs are implemented using Flask and allow for efficient querying and retrieval of data from the PostgreSQL database.

Overview:

The APIs are built using Flask, a lightweight and flexible web framework for Python. The APIs connect to a PostgreSQL database using the psycopg2 library, which facilitates interaction with the database and execution of SQL queries. The database connection details are stored in a configuration file (config.py), ensuring separation of concerns and secure management of credentials.

Database Connection:

The `get_db_connection` function establishes a connection to the AWS RDS PostgreSQL database using the credentials provided in the `config.py` file. It utilizes the `psycopg2` library to connect to the cloud-hosted database and handles any connection errors by logging them and returning a `None` value.

```
from flask import Flask, jsonify, send_file
import psycopg2
from psycopg2.extras import RealDictCursor
from config import db_name, user, password, host

# Initialize Flask application
app = Flask(__name__)

# Function to get database connection
def get_db_connection():
    try:
        # Establish connection to the PostgreSQL database
        conn = psycopg2.connect(
            dbname=db_name,
```

```
            user=user,
            password=password,
            host=host
        )
        return conn
    except psycopg2.Error as e:
        # Log error if connection fails
        app.logger.error(f"Error connecting to database: {e}")
        return None
```

API Endpoints:

The following API endpoints are provided:

- GET /map: Retrieves landslide data in WKT format for mapping purposes.
- GET /landslides/surface: Retrieves landslide surface data.
- GET /landslides/map: Retrieves landslide data for mapping purposes, including WKT geometry.
- GET /landslides/population: Retrieves landslide population data.

Each endpoint establishes a connection to the database, executes a SQL query, processes the results, and returns them in JSON format. If any errors occur during database connection or query execution, appropriate error messages are logged and returned to the client.

GET /map:

This endpoint retrieves landslide data in WKT format, which is useful for mapping purposes. The 'ST_As_Text' function converts the geometry to WKT format, and 'ST_SnapToGrid' is used to adjust the precision of the geometry.

```
@app.route('/map', methods=['GET'])
def get_landslide_data():
    conn = get_db_connection()
    cursor = conn.cursor(cursor_factory=RealDictCursor)
    cursor.execute('''
        SELECT
            cod_reg,
            cod_rip,
            COD_PROV,
            ar_kmq,
            nome,
            uid,
```

```
            ar_fr_p3p4,
            ar_fr_p4,
            ar_fr_p3,
            ar_fr_p2,
            pop_fr_p4,
            pop_fr_p3,
            pop_fr_p2,
            ed_fr_p4,
            ed_fr_p3,
            ed_fr_p2,
            ar_frp3p4p,
            popfrp3p4p,
            ed_fr_p3p4,
            edfrp3p4p,
            ST_AsText(ST_SnapToGrid(geometry, 0.001)) as geom_wkt  -- Adjust
0.001 for your desired precision
        FROM dataset
    ''')
    data = cursor.fetchall()
    cursor.close()
    conn.close()
    return jsonify(data)
```

GET /landslides/surface:

This endpoint retrieves landslide surface data, including area fractions for different categories.

```
@app.route('/landslides/surface', methods=['GET'])
def get_landslides():
    conn = get_db_connection()
    if conn is None:
        return jsonify({"error": "Database connection error"}), 500

    try:
        cursor = conn.cursor(cursor_factory=RealDictCursor)
        cursor.execute("SELECT ar_kmq, nome, ar_fr_p3p4, ar_fr_p2, ar_fr_p1,
ar_fr_p3, ar_fr_p4, ar_fr_aa, ar_frp4_p, ar_frp3_p, ar_frp2_p,ar_frp1_p,
ar_fraa_p, ar_frp3p4p FROM dataset;")
        landslides = cursor.fetchall()
        cursor.close()
        conn.close()
        return jsonify(landslides)
```

```
    except psycopg2.Error as e:
        # Log error if SQL query fails
        app.logger.error(f"Error executing SQL query: {e}")
        return jsonify({"error": "Internal Server Error"}), 500
```

GET /landslides/map:

This endpoint retrieves landslide data for mapping purposes, including WKT geometry. The geometry column is converted to WKT format for each row, and the original geometry_wkt column is deleted.

```
@app.route('/landslides/map', methods=['GET'])
def get_landslides_map():
    conn = get_db_connection()
    if conn is None:
        return jsonify({"error": "Database connection error"}), 500

    try:
        cursor = conn.cursor(cursor_factory=RealDictCursor)
        cursor.execute("SELECT ar_kmq, nome, ar_fr_p3p4, ar_fr_p2, ar_fr_p1,
ar_fr_p3, ar_fr_p4, ar_fr_aa, ar_frp4_p, ar_frp3_p, ar_frp2_p, ar_frp1_p,
ar_fraa_p, ar_frp3p4p, pop_fr_p2, pop_fr_p1, pop_fr_p3, pop_fr_p4, pop_fr_aa,
popfrp4_p, popfrp3_p, popfrp2_p, popfrp1_p, popfrp3p4p, popfr_p3p4, popfraa_p,
ST_AsText(geometry) AS geometry_wkt FROM dataset;")
        landslides_map = cursor.fetchall()

        # Process each row to handle WKT geometry data
        for row in landslides_map:
            wkt_data = row['geometry_wkt']
            try:
                row['geometry'] = wkt_data  # Directly assign the WKT data
            except Exception as e:
                # Log error if processing WKT data fails
                app.logger.error(f"Error processing WKT data: {e}")
                row['geometry'] = None  # Handle or log this error as needed
            del row['geometry_wkt']

        cursor.close()
        conn.close()
        return jsonify(landslides_map)
    except psycopg2.Error as e:
        # Log error if SQL query fails
        app.logger.error(f"Error executing SQL query: {e}")
        return jsonify({"error": "Internal Server Error"}), 500
```

GET /landslides/population:

This endpoint retrieves landslide population data, including population fractions for different categories.

```
@app.route('/landslides/population', methods=['GET'])
def get_landslides_population():
    conn = get_db_connection()
    if conn is None:
        return jsonify({"error": "Database connection error"}), 500

    try:
        cursor = conn.cursor(cursor_factory=RealDictCursor)
        cursor.execute("SELECT ar_kmq,pop_fr_p2, pop_fr_p1, pop_fr_p3, pop_fr_p4,
pop_fr_aa, popfrp4_p, popfrp3_p, popfrp2_p, popfrp1_p, popfrp3p4p, popfr_p3p4,
popfraa_p, nome FROM dataset;")
        landslides_population = cursor.fetchall()
        cursor.close()
        conn.close()
        return jsonify(landslides_population)
    except psycopg2.Error as e:
        # Log error if SQL query fails
        app.logger.error(f"Error executing SQL query: {e}")
        return jsonify({"error": "Internal Server Error"}), 500
```

Error Handling:

The API implementation includes error handling for database connection and SQL query execution. If a database connection cannot be established, an appropriate error message is logged in and returned to the client with a 500-status code. Similarly, any SQL query execution errors are logged and result in a 500-status code response.

Running the Flask Application:

The Flask application can be run locally by executing the script. It listens on 127.0.0.1 at port 5000 and has debugging enabled for development purposes.

```
if __name__ == '__main__':
    # Run the app with debugging enabled, on localhost and port 5000
    app.run(debug=True, host='127.0.0.1', port=5000, use_reloader=False)
```

3.4 Dashboard Design:

Purpose

Description: The AUPE Risk Monitor Dashboard aims to provide an interactive platform for visualizing and analyzing landslide risk data across various regions.

Objectives:

- Enable stakeholders (government officials, researchers) to explore different types of landslide risk metrics.
- Facilitate data-driven decision-making in risk management and mitigation strategies.

The dashboard provides a user-friendly interface to interact with the web application. It includes various features for data visualization, user management, and system monitoring. The web application can be accessed at the following address:

- **URL**: http://127.0.0.1:3000

Technology Stack

Frameworks and Libraries Used:

- **Dash**: Python web framework for building interactive web applications.
- **Plotly**: Visualization library for creating interactive graphs and charts.
- **Folium**: Python wrapper for Leaflet.js maps, used for interactive geographic visualizations.
- **GeoPandas**: Geospatial data manipulation library for handling and analyzing geographic data.

Dependencies:

- pandas, numpy: Data manipulation and numerical computing.
- dash: Core Dash functionalities for building web apps.
- dash-core-components, dash-html-components: Components for building Dash apps with HTML and CSS.
- dash-bootstrap-components: Bootstrap-themed components for Dash apps.
- plotly: Interactive visualizations and charts.
- folium: Interactive mapping and geospatial visualization.
- geopandas: Handling geospatial data structures and operations.

3.4.1. Importing Libraries

The first section of the code involves importing necessary libraries for the application:

```
import requests
import pandas as pd
import numpy as np
import plotly.express as px
import dash
from dash import Dash, dcc, html, dash_table
from dash.dependencies import Input, Output
import geopandas as gpd
from shapely import wkt
import folium
from folium import LayerControl, GeoJson, GeoJsonPopup, GeoJsonTooltip
import branca
import io
import base64
from dash_dangerously_set_inner_html import DangerouslySetInnerHTML
```

Explanation:

➢ requests: For making HTTP requests to fetch data from APIs.
➢ pandas, numpy: For data manipulation and analysis.
➢ plotly.express: For creating interactive plots.
➢ dash: For creating web applications.
➢ dash.dependencies: To manage input-output relationships in the Dash app.
➢ geopandas, shapely, folium: For geospatial data manipulation and visualization.
➢ branca, io, base64: For additional functionality needed in the app.
➢ dash_dangerously_set_inner_html: To render raw HTML content in Dash.

3.4.2. Reading HTML File Content

The next section reads the content of an HTML file:

```
with open('home.html', 'r') as file:
    home_page_content = file.read()
```

Explanation:

- Opens and reads an HTML file named home.html into a variable called home_page_content.

3.4.3. Fetching Data from APIs

Functions to fetch data from a Flask API:

```
def fetch_data_er():
    response = requests.get('http://127.0.0.1:5000/landslides/surface')
    return pd.DataFrame(response.json())

def fetch_population_data():
    response = requests.get('http://127.0.0.1:5000/landslides/population')
    return pd.DataFrame(response.json())

def fetch_map():
    response = requests.get('http://127.0.0.1:5000/landslides/map')
    return pd.DataFrame(response.json())
```

Explanation:

- Each function makes a GET request to a specific endpoint of a Flask API and returns the data as a Pandas DataFrame.

3.4.4. Creating a Folium Map

Function to create a Folium map:

```
def create_folium_map(data, data_type):
m- folium.Map(location-[45.6101948758674, 9.481178872400372], zoom_start=8)
# Create a GeoDataFrame from the geometries
geometries
data['geometry'].tolist()
gdf = gpd.GeoDataFrame(data, geometry-[wkt.loads(geom) for geom in geometries])
if gdf.crs is None:
gdf.set_crs(epsg-32632, inplace=True) # Set the original CRS
gdfgdf.to_crs(epsg-4326)
# Add layers, controls, and other features to the map (code not fully shown in
the extract)
return m._repr_html_() # Return the map as HTML
```

Explanation:

- ➢ Initializes a Folium map centered at specific coordinates.

➢ Converts data into a GeoDataFrame, ensuring it has the correct coordinate reference system.
➢ The map is returned as an HTML iframe.

3.4.5. Creating a choropleth Map

The choropleth map visualizes landslide risks across different provinces, using varying shades of color to represent different levels of risk.

The following function render_regional_analysis() creates a Folium map with choropleth layers based on the selected risk type and level:

```
import folium
import geopandas as gpd
def render_regional_analysis():
# Load GeoJSON data for the map
geo_data = gpd.read_file('path_to_geojson_file.geojson')
# Create a folium map centered at an initial location
m = folium.Map(location=[0, 0], zoom_start=2, tiles='OpenStreetMap')
# Fetch data and prepare it for the choropleth
data = fetch_data()
selected_data = data[(data['risk_type'] == 'surface_area') & (data['risk_level']
=='high')]
# Create the choropleth map
folium. Choropleth(
geo_data-geo_data,
name='choropleth',
data-selected_data,
columns=['province', 'risk_value'],
key_on='feature.properties.province',
fill_color='YlOrRd",
fill_opacity=0.7,
line_opacity=0.2,
legend_name='Landslide Risk Level'
).add_to(m)
# Save the map to an HTML file and render it in an iframe
map_html = 'regional_map.html"
m.save(map_html)
return html.Iframe(srcDoc-open (map_html, 'r').read(), width='100%',
height='600')
# Callback to update map based on user selection
@app.callback(
Output('regional-map-container', 'children'),
[Input('risk-type-dropdown', 'value'),
```

```
Input('risk-level-dropdown', 'value')]
def update_regional_map(selected_risk_type, selected_risk_level):
return render_regional_analysis(selected_risk_type, selected_risk_level)
```

Explanation:

1. **Import Libraries**: The function uses folium for creating the map and geopandas for handling GeoJSON data.
2. **Load GeoJSON Data**: Geographic data is loaded using geopandas.read_file().
3. **Create Folium Map**: Initializes a Folium map centered at a default location.
4. **Fetch Data**: Data is fetched using a fetch_data() function (not shown here but part of the overall project).
5. **Filter Data**: Filters the data based on the selected risk type and level.
6. **Create Choropleth**: Adds a choropleth layer to the map using folium.Choropleth.
7. **Render Map**: Saves the map as an HTML file and renders it within an iframe in the dashboard.

3.4.6. Dash Callbacks

Callbacks to update the dropdown options and visualizations in the Dash app:

```
@app.callback(
Output('province-dropdown-map', 'options'),
Input('data-type-dropdown-map', 'value')
def update_map_province_dropdown(data_type):
if data_type == 'surface':
df- fetch_data_er()
elif data_type ■ 'population':
else:
df
fetch_population_data()
return []
options = [{'label': name, 'value': name} for name in
np.sort(df['nome'].unique())]
return options
```

Explanation:

➢ The update_map_province_dropdown function updates the options of the province-dropdown-map based on the selected data-type.
➢ Fetches appropriate data and generates a list of options for the dropdown.

3.4.7. Running the Dash App

The final part runs the Dash application:

```
if __name__ == '__main__':
    app.run_server(debug=True, port=3000)
```

Explanation:

- Runs the Dash app with debugging enabled.

This structure organizes the main components of the code: importing libraries, reading HTML content, fetching data from APIs, creating a Folium map, defining callbacks for interactive elements, and running the Dash application.

The dashboard section of our design document outlines the essential components and functionalities of the interactive platform for visualizing and analyzing landslide risk data. This dashboard serves as a critical tool for stakeholders, such as government officials and researchers, by providing comprehensive visual and geographic insights into landslide risks across various regions. The combination of technologies and libraries utilized ensures a robust and dynamic user experience, facilitating informed decision-making for risk management and mitigation strategies.

Key Points of Conclusion:

1. **Interactive and User-Friendly Design:**
    - The dashboard is designed with an intuitive interface that includes various interactive elements such as dropdowns, checkboxes, and sliders, allowing users to customize their data view and analysis effortlessly.
2. **Comprehensive Data Visualization:**
    - Utilizes a blend of bar charts, pie charts, tables, and interactive maps to provide a multifaceted view of landslide risk data. This ensures that users can visualize data in the format that best suits their analysis needs.
3. **Geospatial Analysis:**
    - The integration of GeoPandas and Folium for geospatial data handling and visualization allows users to explore the geographical distribution of landslide risks in detail, enhancing spatial awareness and targeted intervention.
4. **Dynamic Data Handling:**

- The dashboard fetches real-time data from APIs and processes it on-the-fly, ensuring that users have access to the most current and relevant information for their analysis.

5. **Scalability and Performance:**
   - Designed with scalability in mind, the dashboard can handle large datasets efficiently, ensuring smooth performance and quick response times, even as the volume of data grows.
6. **Extensibility:**
   - Built on the Dash framework, the dashboard is highly extensible, allowing for future enhancements and the addition of new features as the needs of the users evolve.
7. **Purpose-Driven Visualization:**
   - The dashboard's visualizations are tailored to highlight key metrics and trends in landslide risk, supporting data-driven decision-making processes and facilitating proactive risk management strategies.
8. **Deployment and Accessibility:**
   - Whether deployed locally for development or on a server for wider accessibility, the dashboard is designed to be easily accessible to all intended users, ensuring that crucial landslide risk data is available when and where it is needed.

The dashboard is a powerful tool that centralizes landslide risk data into an accessible and interactive format. By leveraging state-of-the-art technologies and providing a user-friendly interface, it empowers stakeholders to make informed decisions and take proactive measures in landslide risk management and mitigation. This aligns with our objective of enabling data-driven decision-making to enhance safety and preparedness in landslide-prone regions.

4. Application Overview

Data Sources:

- **Internal Databases**: Data fetched using custom functions (fetch_data_er, fetch_population_data) from internal databases.
- **File Sources**: Data may also be sourced from CSV files or other structured data formats.

Key Features:

- **Tabs**: Home, Statistics, Map.
  - **Home Tab**: Welcome message and introductory content.
  - **Statistics Tab**: Visualization of statistical data related to landslide risks.
  - **Map Tab**: Geographic visualization of landslide risk data using interactive maps.
- **Dropdowns and Checkboxes**:
  - Enable users to select data types (surface landslides, population landslides) and specific provinces for analysis.
  - Checkboxes allow selection of visualization types (bar chart, pie chart, table) for data presentation.
- **Visualizations**:
  - **Bar Charts and Pie Charts**: Display statistical distributions and comparisons based on selected metrics (e.g., area affected by different levels of landslide risk).
  - **Tables**: Provide detailed data in tabular format, sortable and filterable based on user selections.
- **Interactive Map**:
  - Utilizes Folium maps to visualize the geographical distribution of landslide risks.
  - Choropleth layers dynamically color regions based on selected risk metrics (e.g., surface area at high risk of landslides).

Architecture:

- **Component Overview**:
  - **Dash Layout**: Structured using html.Div, dcc.Dropdown, dcc.Checklist, and other HTML components for layout and user interaction.
  - **Callbacks**: Defined using @app.callback to update visualizations and maps based on user inputs.
- **Data Processing Flow**:
  - **Data Fetching**: Functions (fetch_data_er, fetch_population_data) retrieve data from databases or file sources.
  - **Data Transformation**: GeoPandas is used to handle geospatial data (e.g., converting WKT geometries, calculating statistical metrics).
  - **Visualization**: Plotly and Dash components (dcc.Graph, dash_table.DataTable) are used to create interactive visualizations.

Deployment Considerations

- **Local Deployment**: Run the Dash app locally for development and testing.
- **Server Deployment**: Deploy on a server for wider accessibility, considering scalability and performance implications.

Design Considerations

User Interface (UI) Design:

- **Layout and Styling**: Designed for clarity and ease of navigation, utilizing Dash and Bootstrap components.
- **Color Schemes**: Chosen to enhance readability and highlight important data points in visualizations.
- **Interactive Elements**: Designed to provide a responsive and engaging user experience.

Performance Optimization:

- **Data Handling**: Efficient data fetching and processing to handle potentially large datasets.
- **Visualization**: Optimize rendering of charts and maps for smooth user interaction.

Components Overview

1. **Header**:
    - **Purpose**: Displays the title "AUPE RiskMonitor" at the top of the dashboard for clear identification.
2. **Risk Type Dropdown**:
    - **Purpose**: Allows users to select different types of landslide risk indicators.
    - **Options**: Includes "Landslide Surface Area", "Population", "Buildings".
3. **Risk Level Dropdown**:
    - **Purpose**: Allows users to select different risk levels within a chosen risk type.
4. **Bar Chart**:
    - **Purpose**: Displays bar charts to compare selected risk indicators across provinces.
    - **Functionality**: Updates dynamically based on selected provinces and risk types.
5. **Slider**:
    - **Purpose**: Allows users to adjust the opacity for map visualization.
6. **Base Tile Dropdown**:
    - **Purpose**: Enables users to select different base tile layers for the map visualization.
7. **Map Container**:
    - **Purpose**: Placeholder for displaying the interactive map.
8. **Table Container**:
    - **Purpose**: Placeholder for displaying tabular data.

Development of AUPE RiskMonitor Dashboard:

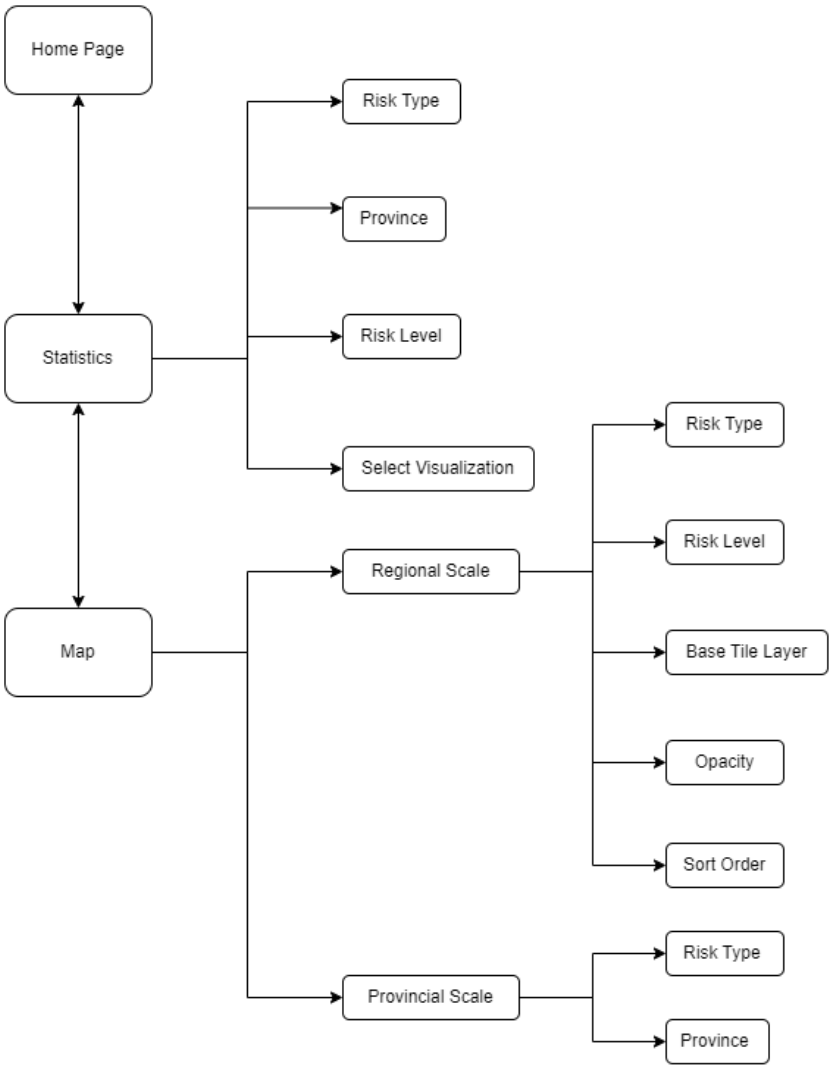


Figure 3: AUPE RiskMonitor Dashboard

5. Interface and User Cases Applications:

In this section we explain the software functionalities, interactions between the components. The purpose is to list the actions taken by the software and users, useful for explaining each internal process of the application.
We describe what is going on from both server-side and client-side on each use case, by specifying the different actions or events that take place in these situations

5.1. Home Page:

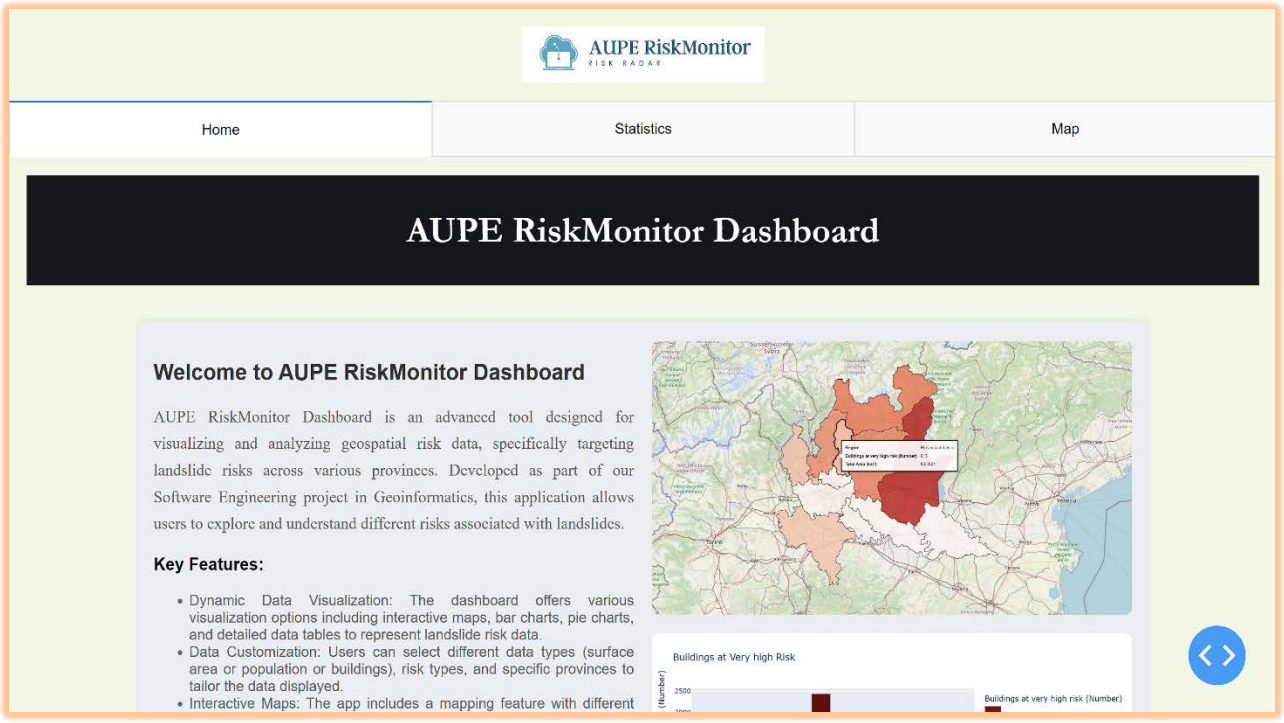


Figure 4: Homepage of AUPE RiskMonitor Dashboard

5.2. Statistics:

Statistics	
Name identifier	DD 1
User	All users
Input	The user has to click on risk type, province, risk level and select visualization and then the bar/and pie chart along with the table appears below.
Possible Actions	The user can hover on the charts to visualize a pop-up window with the basic landslide risk information about the provinces selected. The user can also keep on adding the data from the dropdown option and see how the bar /and the pie chart together with the table go on populating with the users input to have a better visualization of the data.

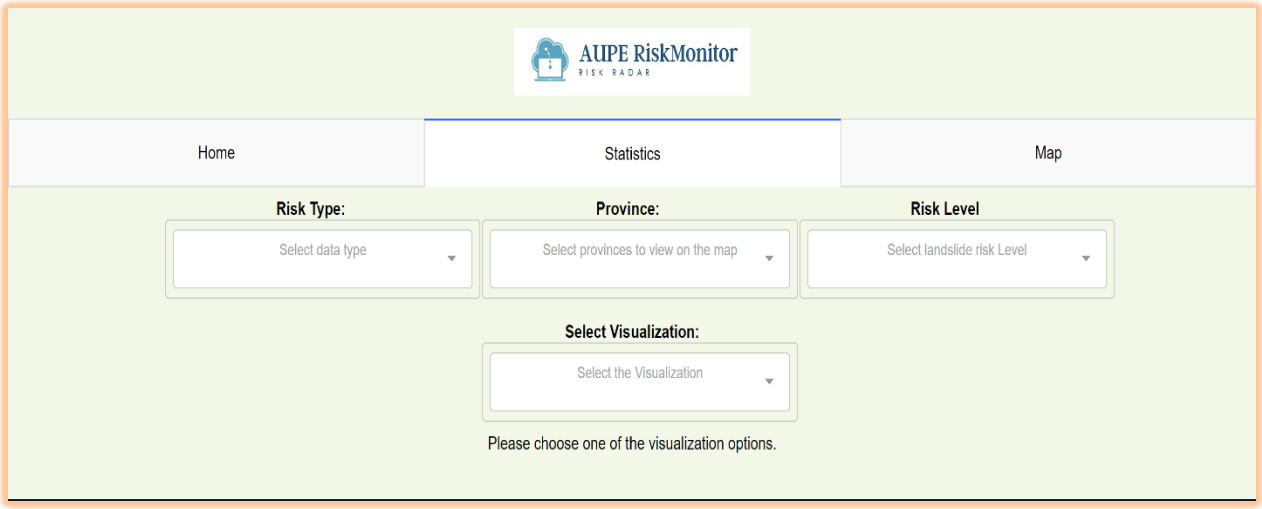


Figure 5: Statistics of AUPE RiskMonitor Dashboard 1

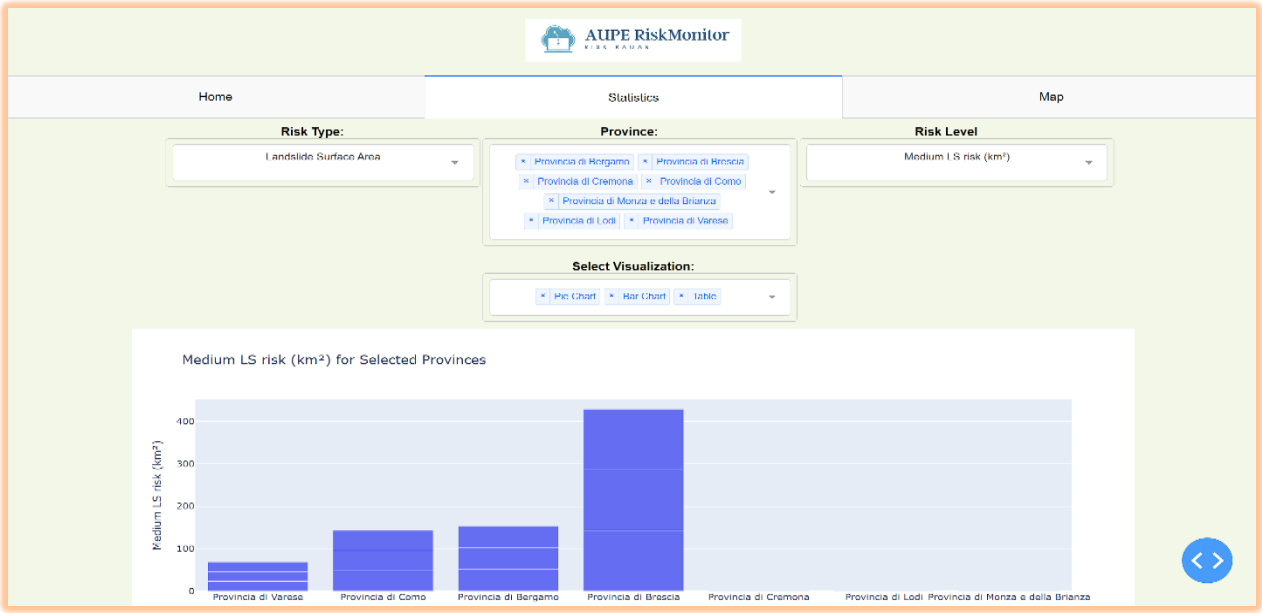
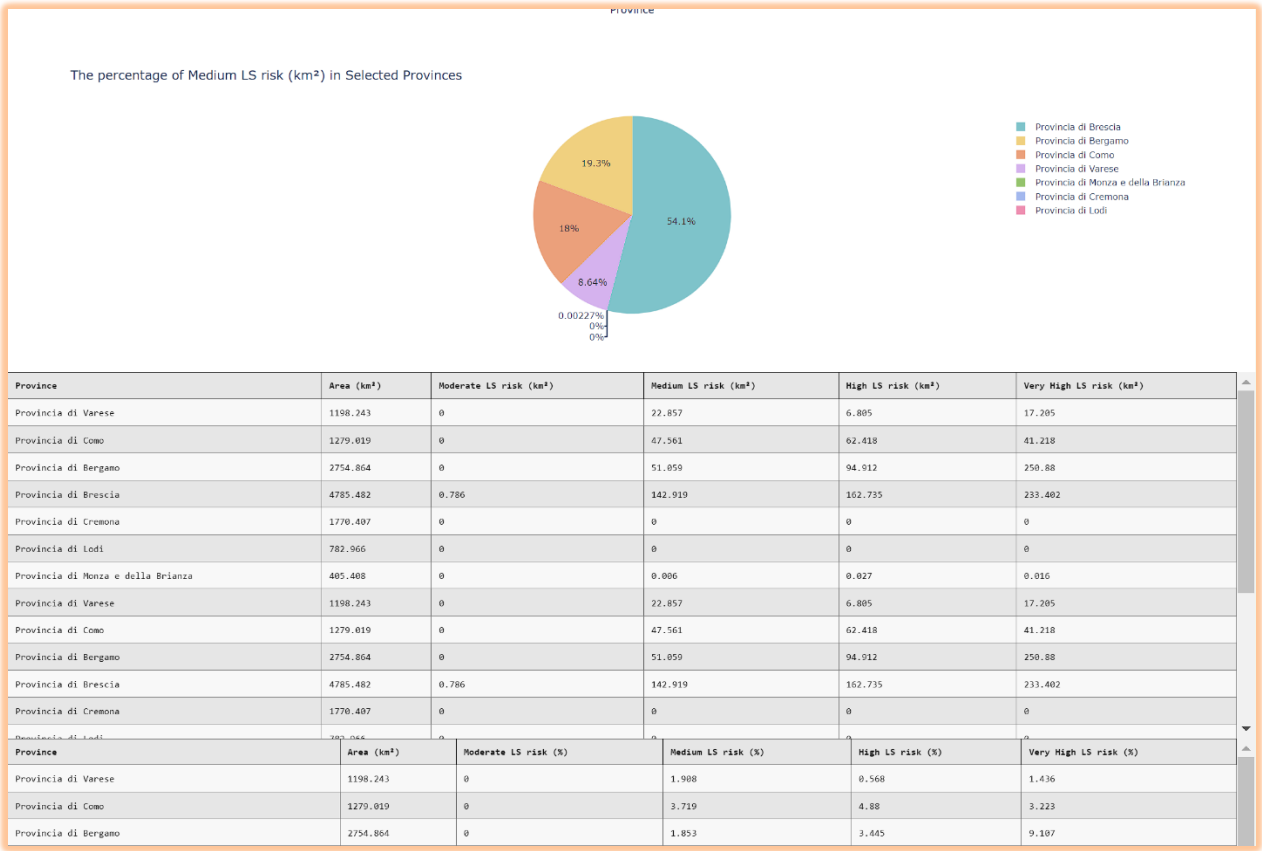


Figure 6: Statistics of AUPE RiskMonitor Dashboard 2



Province	Area (km²)	Moderate LS risk (km²)	Medium LS risk (km²)	High LS risk (km²)	Very High LS risk (km²)
Provincia di Varese	1198.243	0	22.857	6.805	17.205
Provincia di Como	1279.019	0	47.561	62.418	41.218
Provincia di Bergamo	2754.864	0	51.059	94.912	250.88
Provincia di Brescia	4785.482	0.786	142.919	162.735	233.402
Provincia di Cremona	1770.407	0	0	0	0
Provincia di Lodi	782.966	0	0	0	0
Provincia di Monza e della Brianza	405.408	0	0.006	0.027	0.016
Provincia di Varese	1198.243	0	22.857	6.805	17.205
Provincia di Como	1279.019	0	47.561	62.418	41.218
Provincia di Bergamo	2754.864	0	51.059	94.912	250.88
Provincia di Brescia	4785.482	0.786	142.919	162.735	233.402
Provincia di Cremona	1770.407	0	0	0	0

Province	Area (km²)	Moderate LS risk (%)	Medium LS risk (%)	High LS risk (%)	Very High LS risk (%)
Provincia di Varese	1198.243	0	1.908	0.568	1.436
Provincia di Como	1279.019	0	3.719	4.88	3.223
Provincia di Bergamo	2754.864	0	1.853	3.445	9.107

Figure 7: Statistics of AUPE RiskMonitor Dashboard 3

5.3. Map:

The map tab has been divided into two sub-tabs:

5.3.1. Regional Scale:

Regional Scale	
Name identifier	DD 2
User	All users
Input	The user has the options of risk type, risk level and base tile layer to choose from to visualize the choropleth map. The default options are correspondingly set at landslide area, very high and open street map respectively. The app also gives the user the option to change the opacity of choropleth map. Below the map the user is given the option to sort the table corresponding to the options he chooses for the choropleth map which appears below the map
Possible Actions	The user can move the cursor over the choropleth map to look at the data that he chooses from the dropdowns along with the choropleth visualization. Also, beside the map visualization the web app provides an overall graphical tabular representation of the data the user choses from the dropdown that user can sort to have better reading of the data alongside visualization.

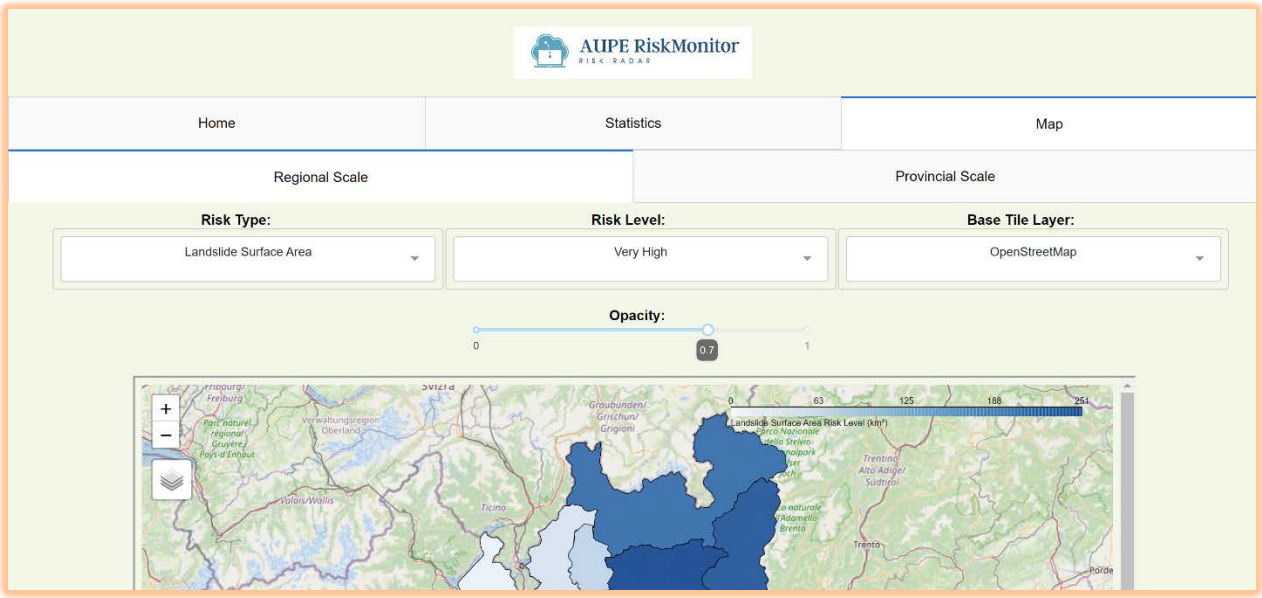


Figure 8: Regional Map Scale of AUPE Riskmonitor Dashboard 1

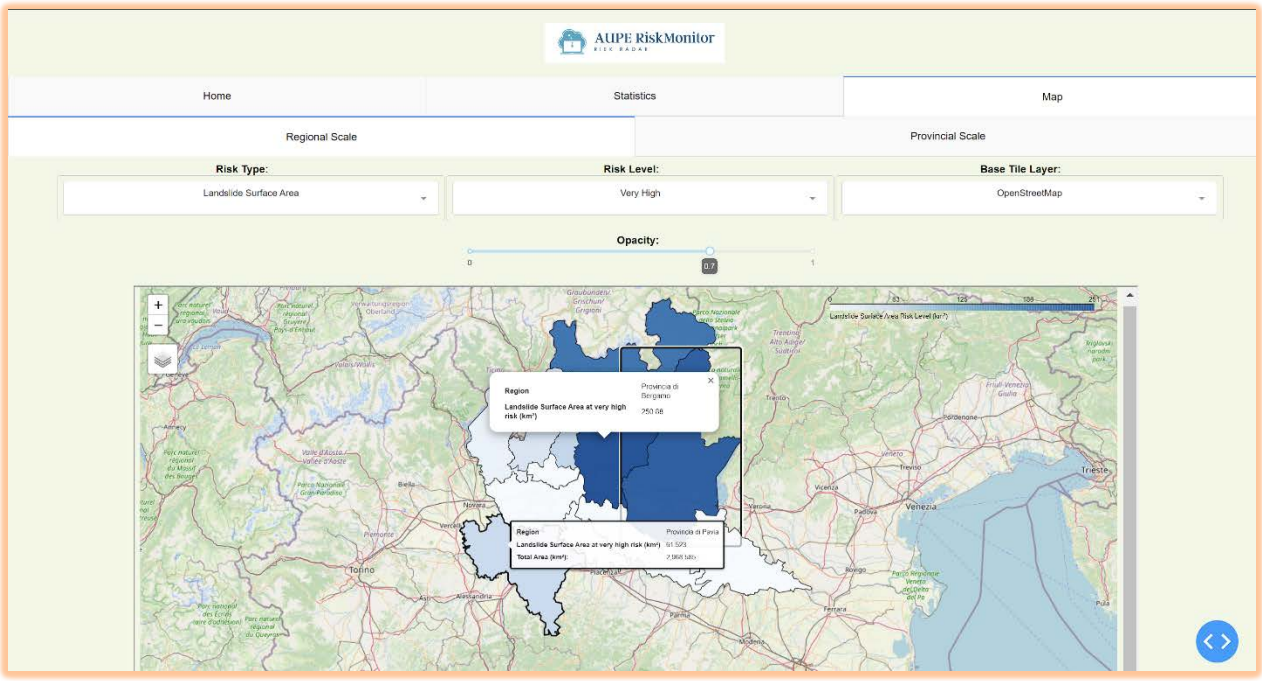


Figure 9: Regional Map Scale of AUPE Riskmonitor Dashboard 2

5.3.2. Provisional Scale:

Provisional Scale	
Name identifier	DD 3
User	All users
Input	The user has the options of risk type and province to select. There are no default options, the user has to first select the options then only the map will be displayed.
Possible Actions	After selecting the options, the user gets a map at the provincial scale (individual level) with more information data popups integrated across all the risk levels. Also, as the user keeps on selecting the data from the dropdowns, the provinces go on adding onto the basemap. The user can even cancel some of the provinces from the selected options.

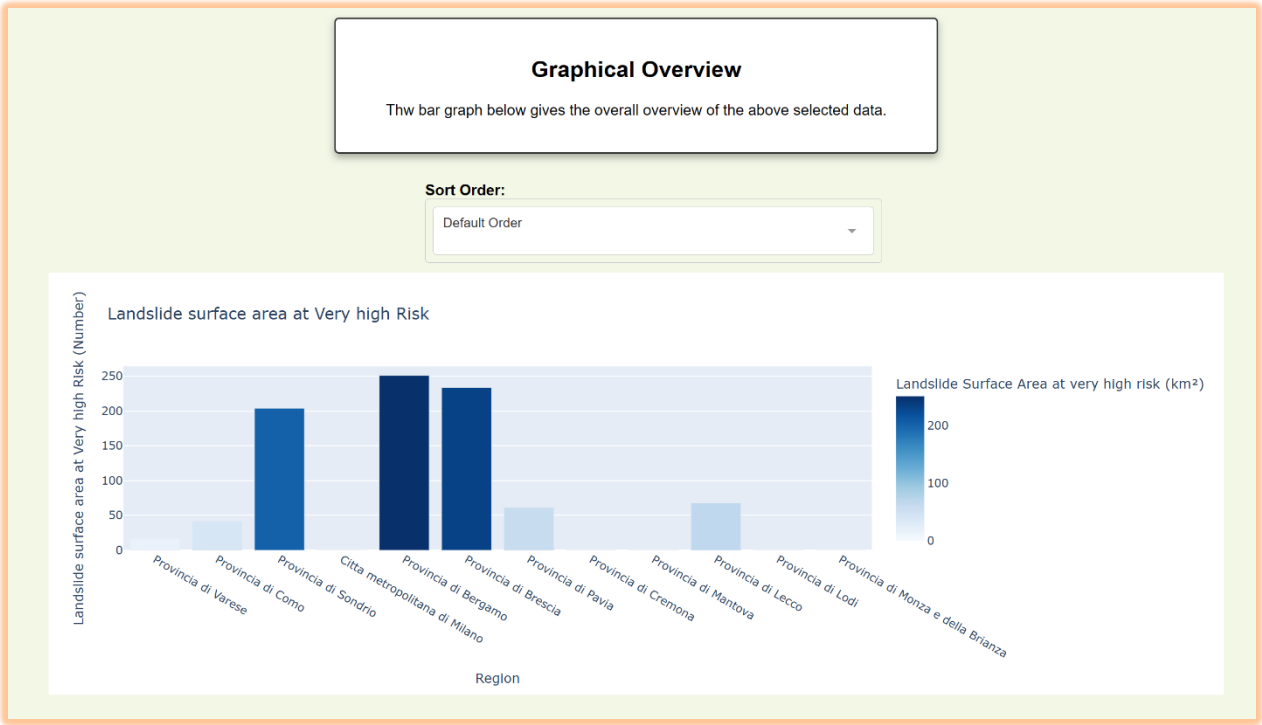


Figure 10: Map Provisional Scale of AUPE RiskMonitor Dashboard

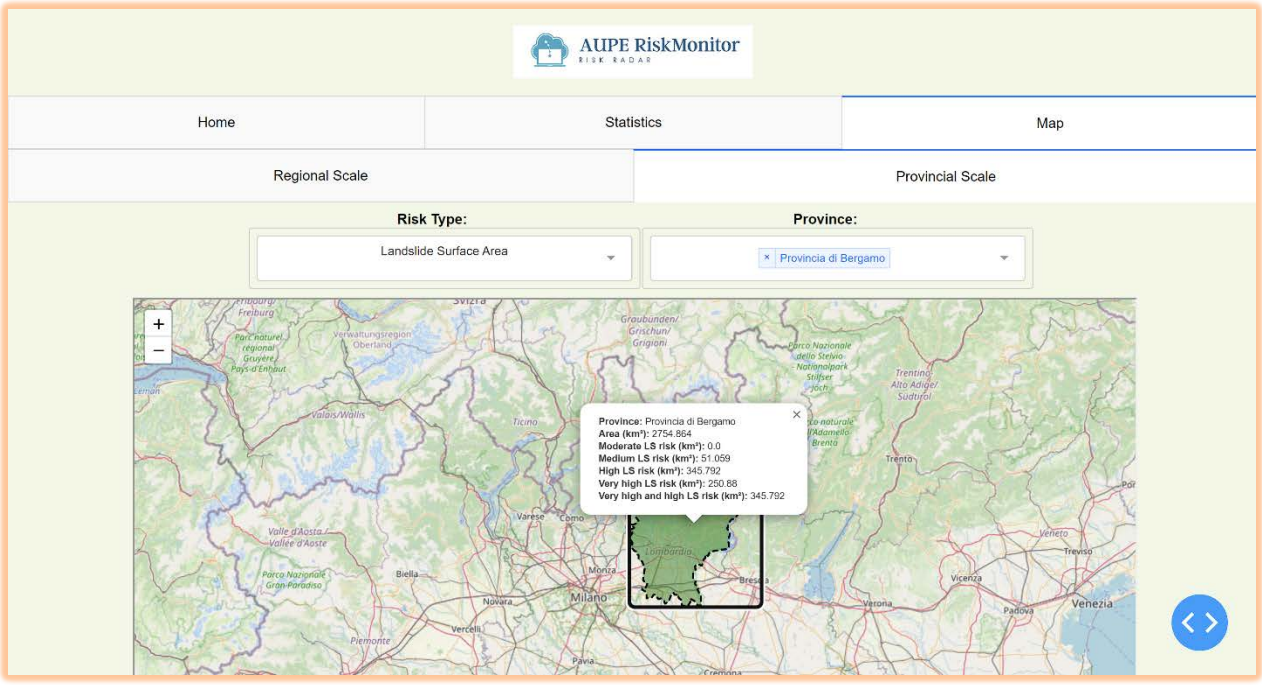


Figure 11: Map Provisional Scale of AUPE RiskMonitor Dashboard 2

6. Implementation Plan and Test Plan:

6.1 Implementation Plan

The implementation plan for the AUPE RiskMonitor dashboard includes developing interactive components using Dash and Plotly, integrating with a Flask API for data retrieval, and ensuring responsiveness across different devices. The plan follows an Agile methodology with bi-weekly sprints and leverages Python programming tools and libraries.

6.2 Test Plan

The test plan encompasses unit testing for individual components, integration testing for seamless interaction between modules, and user acceptance testing (UAT) for usability and performance validation. Testing will be automated using Pytest and conducted in a staging environment before deployment to production. **It's important to mention that the application is yet to be tested.**

References:

- https://idrogeo.isprambiente.it/app/
- Eisaa et al. (2022). Requirement Analysis and Specification Document - PRESENTATION OF AIR POLLUTION DATA USING AN INTERACTIVE WEB MAP.
- OpenAQ. n.d. Available: https://openaq.org/#/[Fetched: 2022-05-26]
- Kovacic, D., & Beard, K. (2021, April 29). Accessibility inspired: Dark mode. Habanero Consulting Inc. Retrieved June 7, 2022, from https://www.habaneroconsulting.com/stories/insights/2021/accessibility-inspired-dark-mode
- World Leaders in Research-Based User Experience. (n.d.). 10 usability heuristics for user interface design. Nielsen Norman Group. Retrieved June 7, 2022, from https://www.nngroup.com/articles/ten-usability-heuristics/
- Choosing dark mode for low vision. Veroniiiica. (2020, August 25). Retrieved June 7, 2022, from https://veroniiiica.com/2020/05/15/dark-mode-for-low-vision/