Green Living

Arango DB Hackathon

Building the Next-Gen Agentic App

with GraphRAG & NVIDIA cuGraph

01
Introduction

The Team

Waiz Al Qorni

Data Analyst | Data Engineer

Jafar Aziz

Software Engineer

Background

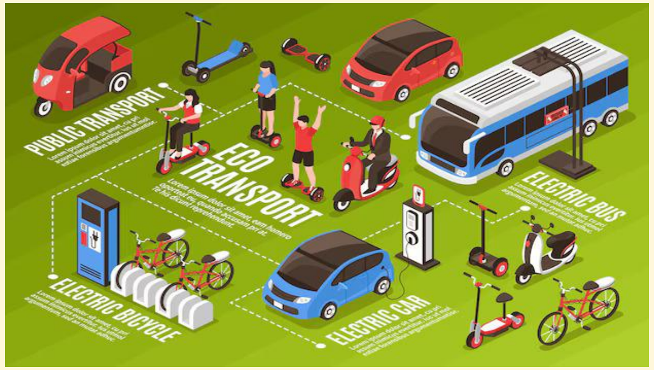
“Our addiction to fossil fuels is akin to a 'Frankenstein's monster,' unleashing havoc on our planet.”

(Antonio Guterez, UN General Secretary)

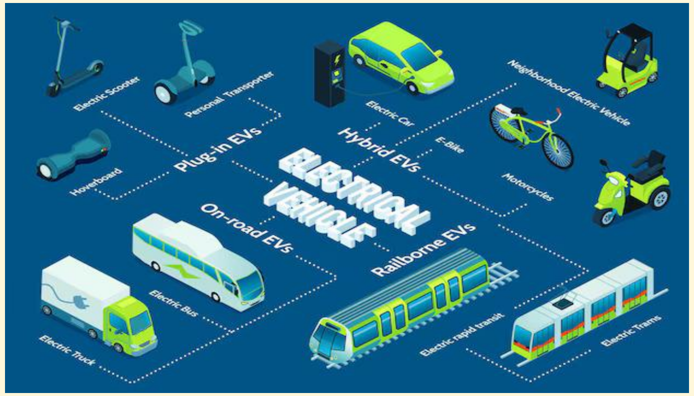
“The collapse of our civilizations and the extinction of much of the natural world is on the horizon”

(Sir David Attenborough, Biologist | Broadcaster)

Background



"Green Living Is On You"



02

Data & Business Problem

Data

In our project, we integrated three primary datasets to construct a comprehensive environmental knowledge graph:

- Sentinel Copernicus Satellite Imagery: This dataset provides high-resolution Earth observation data, enabling us to assess environmental factors such as land use, vegetation cover, and pollution levels.

- OpenStreetMap (OSM): An open-source mapping platform that offers detailed information on various geographical features, including infrastructure like EV charging stations, parks, waste recycling facilities, and administrative boundary.

- Event Registry: A platform that aggregates global news articles, allowing us to extract and analyze news related to environmental events and trends.

Business Problem Addressed

- Environmental Monitoring: Providing up-to-date information on pollution levels, green spaces, and renewable energy infrastructure to inform public awareness and policy decisions.

- Infrastructure Accessibility: Identifying the availability and distribution of facilities like EV charging stations, public transport station, renewable power generator and waste recycling facilities to encourage their utilization.

- Information Dissemination: Aggregating and analyzing news related to environmental issues to keep communities informed and engaged.

By leveraging these datasets, our knowledge graph serves as a dynamic tool to facilitate data-driven decisions, ultimately contributing to a more sustainable future.

03

Processing Graph

Data Processing to Graph

Geospatial Data Processing:

- We utilized formats like Parquet and GeoPackage (GPKG) to store geospatial data efficiently, ensuring quick access and reduced storage overhead.

- By performing spatial joins, we integrated various geospatial datasets, aligning features based on spatial relationships to enrich our data.

News Data Processing with NER and LLM:

- We applied Named Entity Recognition (NER) techniques to extract entities such as organizations, locations, and environmental terms from news articles.

- Leveraging Large Language Models (LLMs), we contextualized these entities, linking them to our existing graph nodes and uncovering new relationships.

Data Processing : Data Wrangling & Conversion

```
# Set the directory containing Parquet files
directory = "Energy/"  # Change this to your folder

# List all Parquet files in the directory
parquet_files = [os.path.join(directory, f) for f in os.listdir(directory) if f.endswith('.parquet') and f != 'solar_new.parquet']

# List to store processed DataFrames
dataframes = []

# Process each Parquet file
for file in parquet_files:
    print(f"Processing: {file}")

    # Read the Parquet file
    data = pd.read_parquet(file)

    # Convert WKB geometry to Shapely
    data['geom'] = data['geometry'].apply(lambda x: loads(x))

    # Create obj_type column
    data['obj_type'] = data['generator:source'] + ' power generator'

    # Select relevant columns
    processed_data = pd.DataFrame(data[['id', 'obj_type', 'geom']])

    # Store DataFrame
    dataframes.append(processed_data)

# Merge all DataFrames
merged_data = pd.concat(dataframes, ignore_index=True)
```

```
Processing: Energy/battery.parquet
Processing: Energy/geo.parquet
Processing: Energy/hydro.parquet
Processing: Energy/tidal.parquet
Processing: Energy/wave.parquet
Processing: Energy/wind.parquet
```

```
data_Rc['obj_type'] = 'waste recycle facility'
data_Rc_use = pd.DataFrame(data_Rc[['id','obj_type','geom']])
data_Rc_use
```

	id	obj_type	geom
0	node/20944608	waste recycle facility	POINT (12.6362535 55.6571437)
1	node/26066489	waste recycle facility	POINT (6.655354 51.4405287)
2	node/26066525	waste recycle facility	POINT (6.6427162 51.4507804)
3	node/26209296	waste recycle facility	POINT (9.8365819 52.2240181)
4	node/26209297	waste recycle facility	POINT (9.8443886 52.2245821)
...	...	...	...
85829	node/12628343471	waste recycle facility	POINT (13.9538195 50.1473361)
85830	node/12628723501	waste recycle facility	POINT (6.1924241 51.0619215)
85831	node/12628735121	waste recycle facility	POINT (14.5521379 50.0648706)
85832	node/12628735122	waste recycle facility	POINT (14.5518273 50.0650073)
85833	node/12629315013	waste recycle facility	POINT (11.9719773 50.9741364)

85834 rows × 3 columns

```
# Convert to GeoDataFrame (optional)
gdf = gpd.GeoDataFrame(data_Rc_use, geometry='geom')

# Save as a new Parquet file
output_file = "Recycling/use/rc_all_data.parquet"
gdf.to_parquet(output_file)

print(f"✅ Merged data saved to {output_file}")
```

```
✅ Merged data saved to Recycling/use/rc_all_data.parquet
```

Data Processing : Spatial Join

```
# Load point and polygon layers from GeoPackage
gdf_polygons = gpd.read_file("Data_use\gpkg\europe.gpkg")
```

```
gdf_points = gpd.read_file("Data_use\gpkg\\pt_all_data.gpkg")
```

```
# Perform spatial join (assigning polygon attributes to points)
joined_gdf = gpd.sjoin(gdf_points, gdf_polygons, how="left", predicate="within")
joined_gdf
```

	id	obj_type	geometry	index_right	name	country_name
0	node/104734	public transport station	POINT (-1.78588 51.56565)	885.0	Swindon	England
1	node/105105	public transport station	POINT (-0.02697 52.05321)	774.0	Hertfordshire	England
2	node/223749	public transport station	POINT (29.77056 59.98477)	NaN	NaN	NaN
3	node/271281	public transport station	POINT (-1.50497 50.85505)	782.0	Hampshire	England
4	node/271323	public transport station	POINT (-1.60995 50.78471)	782.0	Hampshire	England
...	...	...	...	...	...	...
58654	node/12624445080	public transport station	POINT (5.95133 46.57483)	66.0	Bourgogne-Franche-Comté	France
58655	node/12625428589	public transport station	POINT (-1.52439 55.10327)	890.0	Northumberland	England
58656	node/12625592670	public transport station	POINT (34.49079 50.56930)	690.0	Sumy Oblast	Ukraine
58657	node/12625592675	public transport station	POINT (34.47248 50.57964)	690.0	Sumy Oblast	Ukraine
58658	node/12625853738	public transport station	POINT (17.64656 59.85822)	641.0	Uppsala län	Sweden

58659 rows × 6 columns

```
import pandas as pd
joined_df = pd.DataFrame(joined_gdf)
joined_df
```

	id	obj_type	geometry	index_right	name	country_name
0	node/104734	public transport station	POINT (-1.78588 51.56565)	885.0	Swindon	England
1	node/105105	public transport station	POINT (-0.02697 52.05321)	774.0	Hertfordshire	England
2	node/223749	public transport station	POINT (29.77056 59.98477)	NaN	NaN	NaN
3	node/271281	public transport station	POINT (-1.50497 50.85505)	782.0	Hampshire	England
4	node/271323	public transport station	POINT (-1.60995 50.78471)	782.0	Hampshire	England

Data Processing : LLM Generated NER and Knowledge Graph

```
    Example response:
    [
        {{"word": "Jensen Huang", "entity": "Person"}},
        {{"word": "electric car", "entity": "Electric Vehicle"}},
        {{"word": "Germany", "entity": "Country"}},
        {{"word": "Munich", "entity": "Location"}}
    ]
    article :
    {input_data['txt']}
    """
    response = llm.invoke(query)
    return index, query, response  # Return the index to maintain order

requests_processed = 0  # Counter to track the number of requests processed

with tqdm.tqdm(total=len(data_news_use), desc="Processing") as pbar:
    for i, row in data_news_use.iterrows():
        index, query, response = process_row(i, row)
        queries[index] = query  # Store at correct index
        responses[index] = response
        requests_processed += 1

        # Introduce 1-minute delay after every 10 requests
        if requests_processed % 10 == 0:
            print("Processed 10 requests, waiting for 1 minute...")
            time.sleep(60)  # Delay for 1 minute

        pbar.update(1)
```

```
Processing:   3%|█          | 9/353 [01:19<35:33,  6.20s/it]  Processed 10 requests, waiting for 1 minute...
Processing:   5%|█          | 19/353 [03:28<42:58,  7.72s/it]  Processed 10 requests, waiting for 1 minute...
Processing:   8%|█          | 29/353 [06:06<57:04, 10.57s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  11%|█          | 39/353 [07:58<37:29,  7.16s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  14%|█          | 49/353 [09:55<32:48,  6.48s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  17%|██         | 59/353 [11:45<33:11,  6.77s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  20%|██         | 69/353 [13:53<28:43,  6.07s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  22%|██         | 79/353 [16:04<34:09,  7.48s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  25%|██         | 89/353 [18:19<54:57, 12.49s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  28%|██         | 99/353 [20:30<28:11,  6.66s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  31%|███        | 109/353 [22:35<29:51,  7.34s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  34%|███        | 119/353 [24:33<23:28,  6.02s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  37%|███        | 129/353 [26:24<21:59,  5.89s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  39%|███        | 139/353 [28:25<30:02,  8.42s/it]  Processed 10 requests, waiting for 1 minute...
Processing:  42%|███        | 149/353 [30:18<18:19,  5.39s/it]  Processed 10 requests, waiting for 1 minute...
```

```
[ ] data_join = pd.merge(data_ent_country, data_country, how='left', left_on='word', right_on='country_name')
    data_join
```

	uri	type	word	entity	country_name
0	8571512101	article	South Africa	Country	NaN
1	8571512101	article	UK	Country	NaN
2	8571512101	article	US	Country	NaN
3	8571512101	article	Germany	Country	Germany
4	8571512101	article	France	Country	France
...	...	...	...	...	...
130	8574199196	article	SA	Country	NaN
131	8574184065	article	Czech Republic	Country	Czech Republic
132	8574193273	article	Kazakhstan	Country	Kazakhstan
133	eng-10384176	event	New Mexico	Country	NaN
134	eng-10294659	event	Americans	Country	NaN

135 rows × 5 columns

```
[ ] data_ent_country_join = pd.merge(data_entity, data_join[['word','country_name']], how='left', left_on='word', right_on='word')
    data_ent_country_join
```

	uri	type	word	entity	country_name
0	8571515568	article	Currys	Brand	NaN
1	8571515568	article	MailOnline	Organization	NaN
2	8571515568	article	Jessica Watson	Person	NaN
3	8571515568	article	Gloriah	Brand	NaN
4	8571515568	article	Holly Jackson	Person	NaN

Data Processing to graph

```
[ ] # Initialize directed graph
    G = nx.DiGraph()
```

```
# • Vectorized Batch Node Insert
G.add_nodes_from([(row["country_name"], {"label":row["country_name"],"type": "Country"})
                  for row in data_country.to_dict(orient="records")])

G.add_nodes_from([(row["name"], {"label":row["name"], "type": "City", "geometry": row["geometry"]})
                  for row in data_city_use.to_dict(orient="records")])

G.add_nodes_from([(row["id"], {"type": "PowerGenerator", "location": row["location"]})
                  for row in data_energy_use.to_dict(orient="records")])

G.add_nodes_from([(row["id"], {"type": "EVChargingStation", "location": row["location"]})
                  for row in data_ev_use.to_dict(orient="records")])

G.add_nodes_from([(row["id"], {"type": "GreeneryLand", "location": row["location"]})
                  for row in data_green_use.to_dict(orient="records")])

G.add_nodes_from([(row["id"], {"type": "PublicTransportStation", "info":row["add_info_custom_by_waiz"], "location": row["location"]})
                  for row in data_transport_use.to_dict(orient="records")])

G.add_nodes_from([(row["id"], {"type": "WasteRecycleFacility", "location": row["location"]})
                  for row in data_recycling_use.to_dict(orient="records")])

G.add_nodes_from([(row["obj_type"], {"label": row["obj_type"],"type": "ObjectType"})
                  for row in data_obj_type.to_dict(orient="records")])

# • Vectorized Batch Edge Insert
G.add_edges_from([(row["name"], row["country_name"], {"relation": "located_in"})
                  for row in data_city_use.to_dict(orient="records")])

G.add_edges_from([(row["id"], row["name"], {"relation": "located_in"})
                  for row in data_energy_use.to_dict(orient="records")])

G.add_edges_from([(row["id"], row["name"], {"relation": "located_in"})
                  for row in data_ev_use.to_dict(orient="records")])

G.add_edges_from([(row["id"], row["name"], {"relation": "located_in"})
                  for row in data_green_use.to_dict(orient="records")])
```

WE use networkx Di graph for convert the data to graph

```
[ ] G.add_nodes_from([(row["word"], {"label":row["word"],"type": "NewsEntity"})
                      for row in data_news_entity_use_node.to_dict(orient="records")])

    G.add_nodes_from([(row["entity"], {"label":row["entity"],"type": "NewsEntityType"})
                      for row in data_news_entity_type.to_dict(orient="records")])

    ## Edge
    G.add_edges_from([(row["uri"], row["word"], {"relation": "mention"})
                      for row in data_news_entity_use.to_dict(orient="records")])

    G.add_edges_from([(row["uri"], row["name"], {"relation": "related_to"})
                      for row in data_news_entity_use_edge_city.to_dict(orient="records")])

    G.add_edges_from([(row["uri"], row["country_name"], {"relation": "related_to"})
                      for row in data_news_entity_use_edge.to_dict(orient="records")])

    G.add_edges_from([(row["word"], row["entity"], {"relation": "belongs_to"})
                      for row in data_news_entity_use.to_dict(orient="records")])
```

```
print(G)
```

```
DiGraph with 1973159 nodes and 2228222 edges
```

04

Persist the Graph

Persist the graph to Arango db

- We use Networkx arango db package (nxadb) to persist the graph into arango db

```
# Connect to ArangoDB
graph_name = "green_living_graph"
arangodb = nxadb.Graph(
    name=graph_name,
    incoming_graph_data=G,
    write_batch_size=50000 # feel free to modify
)

print("Graph structure created!")
```

```
[16:12:00 +0700] [INFO]: Graph 'green_living_graph' created.
[2025/03/07 16:12:04 +0700] [13672] [INFO] - adbnx_adapter: Instantiated ADBNX_Adapter with database 'green_living'
Output()
Output()
[2025/03/07 16:16:29 +0700] [13672] [INFO] - adbnx_adapter: Created ArangoDB 'green_living_graph' Graph
Graph structure created!
```

```
# Test Load Graph from Arango db to networkx
G_adb = nxadb.Graph(name="green_living_graph")

print(G_adb)
```

```
[09:44:07 +0000] [INFO]: Graph 'green_living_graph' exists.
INFO:nx_arangodb:Graph 'green_living_graph' exists.
[09:44:07 +0000] [INFO]: Default node type set to 'green_living_graph_node'
INFO:nx_arangodb:Default node type set to 'green_living_graph_node'
Graph named 'green_living_graph' with 1973159 nodes and 2228222 edges
```

```
[ ] # 3. Print the degree of a Node

G_adb.degree(1000)
```

```
2
```

```
# Traverse a node's 1-hop neighborhood
result = G_adb.query("""
    FOR node, edge, path IN 1..1 ANY 'green_living_graph_node/1' GRAPH green_living_graph
        LIMIT 1
        RETURN path
""")

print(list(result))
```

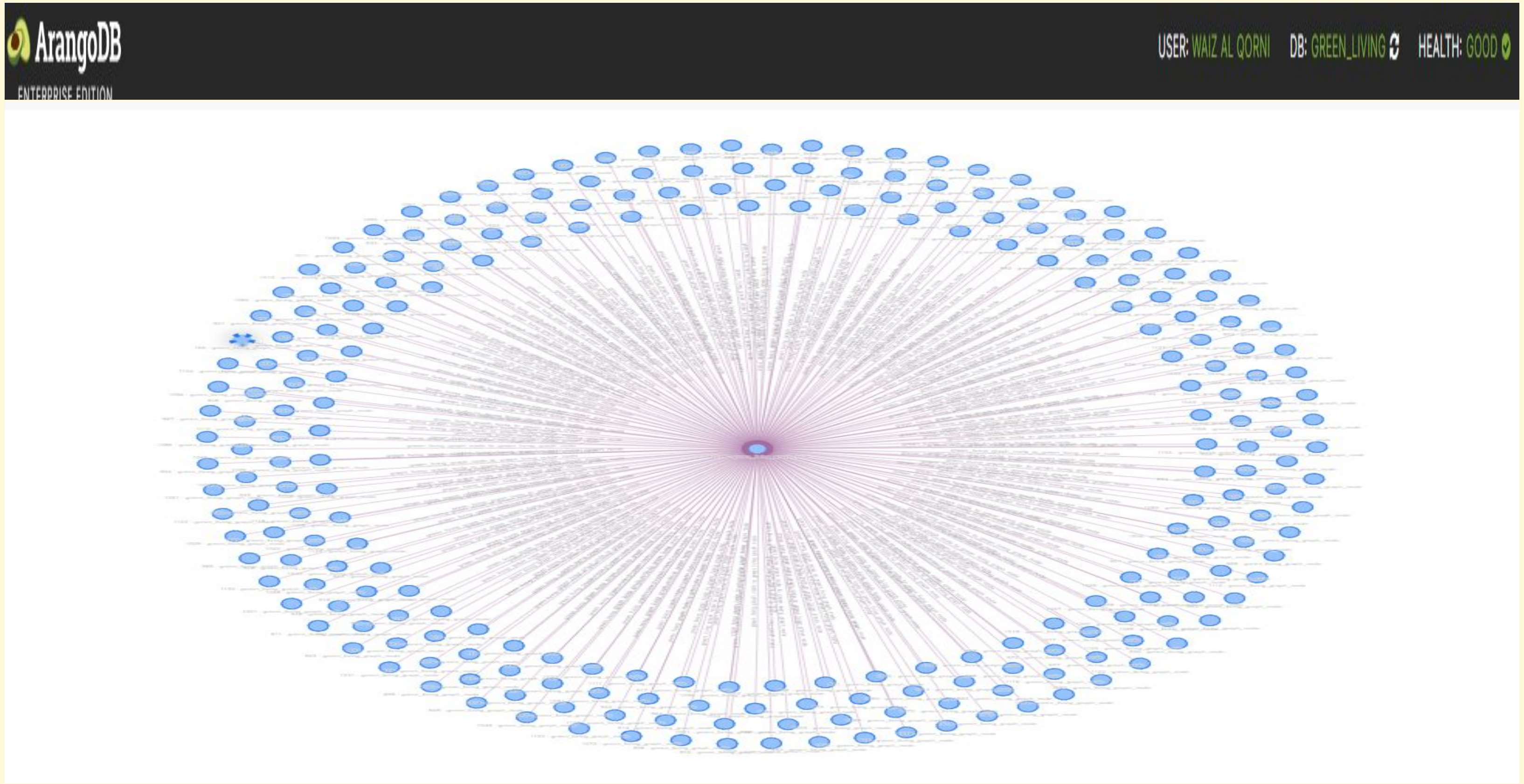
```
[{'_key': '456762', '_id': 'green_living_graph_node/456762', '_rev': '_jU9etp6--1', 'type': 'PopulationGrid', 'location': {'lat': 7.087916661, 'lon': 51.47041666}, 'value': 1844.3
----------
[{'_key': '975185', '_id': 'green_living_graph_node_to_green_living_graph_node/975185', '_from': 'green_living_graph_node/720513', '_to': 'green_living_graph_node/2', '_rev': '_jU
----------
[{'vertices': [{'_key': '1', '_id': 'green_living_graph_node/1', '_rev': '_jU9eXWu--_', 'label': 'Baden-Württemberg', 'type': 'City', 'geometry': "{'type': 'MultiPolygon', 'coordi
```

05

Graph Visualization

Graph visualization

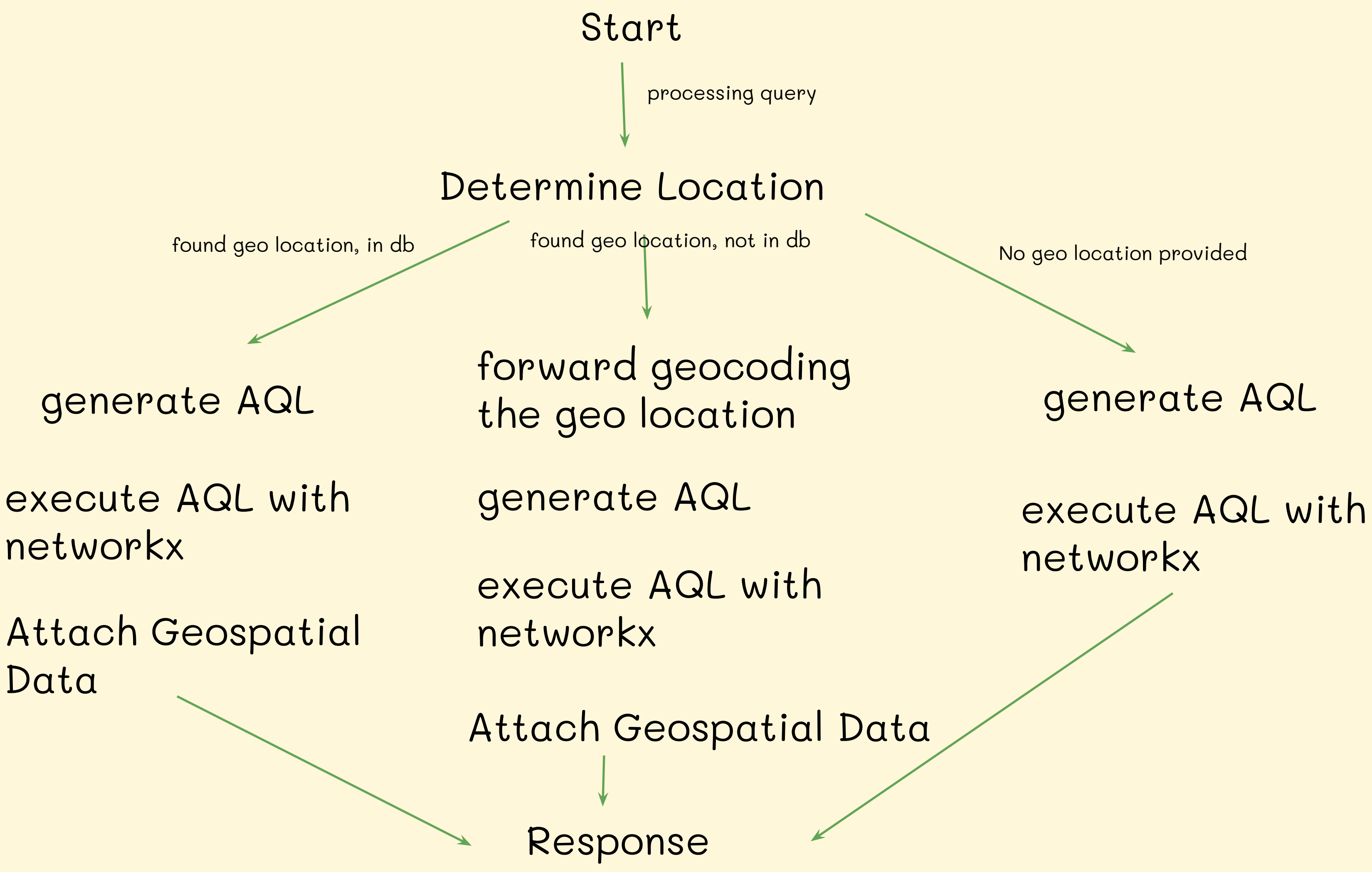
We choose to use the arango db web for graph visualization for ease to use and good performance



06

Agentic App

Structure


Start
processing query
Determine Location
found geo location, in db
found geo location, not in db
No geo location provided
generate AQL
execute AQL with networkx
Attach Geospatial Data
forward geocoding the geo location
generate AQL
execute AQL with networkx
Attach Geospatial Data
generate AQL
execute AQL with networkx
Response

Implementation

We implement the agentic app with langchain & langgraph

```python
# 🔹 LLM Function to Generate AQL Query
def gen_aql(query):
    """Uses LLM to generate good aql query"""

    llm = ChatOpenAI(temperature=0, model_name="gpt-4o")

    prompt = f"""
    Generate AQL QUery For given Task.
    Just return the Top 10 object/grid/news node, not the polygon based node data e.g. city or country.
    Attention to the following schema, and be detail to the schema, The Node Type City has edge INBOUND, Not OUTBOUND.
    database schema: {schema}.

    The NODE City HAS INBOUND EDGE!
    The NODE type News ONLY HAS OUTBOUND EDGE TO City!
    USE AQL DISTANCE(),
    DON'T USE GEO_DISTANCE(),
    GENERATE RESPONSE ONLY THE AQL, RETURN ONLY TOP 10 data, NOTHING ELSE!
    Task: "{query}"
    """

    response = llm.invoke(prompt).content.strip()

    return response  # No valid location found
```

```python
@tool
def text_to_nx_algorithm_to_text(AQLquery: str, main_query:str):
    """This tools is used for Perform graph analysis. and translating the results back
    to Natural Language, with respect to the main query. for find result object/grid, use the G_adb.query("<AQL>") function.
    if you need more compex analysis to generate insight, then use networkx"""
    llm = ChatOpenAI(temperature=0, model_name="gpt-4o")

    # 🔸 Generate Python Code for NetworkX
    # print("1) Executing NetworkX query...")

    FINAL_RESULT = G_adb.query(f"""
    {AQLquery}
    """)

    # 🔸 Convert Result to Natural Language
    nx_to_text = llm.invoke(f"""
    I have a NetworkX Graph `G_adb` with schema: {schema}.

    Language Query: {main_query}

    Result: {list(FINAL_RESULT)}

    Generate a concise natural language response to gain insight.
    Write only text natural language in text_generated_field, and the result query in QUERY_RESULT field.
    with the format :
      {{"text_generated":"...", "QUERY_RESULT":"..."}}

    Response:
    """).content

    return nx_to_text
```

```python
# from langchain.memory import ConversationBufferWindowMemory
from langchain.schema import SystemMessage

# 🔹 Agentic AI App with Query Tools
tools = [text_to_nx_algorithm_to_text, favourite_fruit, favourite_color]

# 🔹 Define Memory with 3-Turn Limit
# memory = ConversationBufferWindowMemory(k=3, return_messages=True)

def query_graph(query):
    """Agent function."""
    llm = ChatOpenAI(temperature=0, model_name="gpt-4o")

    user_schema = f"""
    You are working with an **ArangoDB Graph** and **NetworkX** from persistent arango db graph database.

    my data has following schema {schema}

    """

    # 🔹 System Message with Schema
    system_message = SystemMessage(content=user_schema)

    # 🔹 Define Agent with Tools and Memory
    app = create_react_agent(llm, tools)

    # 🔸 Determine Location in Query
    location = extract_location(query)

    print(location)

    if location:
        if location["type"] == "city":
            query = f"""I have 2 task for you. Find Top 10 most relevant geo point object/grid node data (with all attribute) with the main que
            Then Generate Natural language to gain insight about the query result, to fill the text_generated field.
            main query : {query}.
            AQL query : {gen_aql(query)}.
            leave empty for object.centre, fill the object.data_points, write all query result (TOP 20),
            Response format:
```

Implementation

```python
        final_state = app.invoke({"messages": [system_message, {"role": "user", "content": query}]})
        final_response = '{"data":'+final_state["messages"][-1].content+ f""" ,"data_polygon":{location["geometry"]}}"""

    elif location["type"] == "geocode":
        query = f"""I have 2 task for you. Find Top 10 most relevant geo point object/grid node data (with all attribute) with the main query within 5km of longitude : {location['lon']},
        Then Generate Natural language to gain insight about the query result, to fill the text_generated field.
        take the reference location longitude : {location['lon']}, latitude : {location['lat']},  as the object.centre, write all query result (TOP 20),
        main_query:{query.replace(location['name'],f"at the coordinate point longitude : {location['lon']}, latitude : {location['lat']}")}
        Response format:
          "object" :{{
            "data_points":[{{node data}}]
            "centre":{{"lat": ,"lon": }}
          }}
          "text_generated" : ""
          JUST ANSWER JSON and NOTHING ELSE
          """
        # ♦ Invoke Agent with Memory
        final_state = app.invoke({"messages": [system_message, {"role": "user", "content": query}]})
        final_response = '{"data":'+final_state["messages"][-1].content+ f""" ,"data_polygon":{{}}}}"""

else:
  query = f"""Generate AQL query you can get from the data in arangodb and gain insight about the query result, then attach it into text_generated field of response as a natural language
        main query : {query}.
        leave empty for object, fill the text_generated, Example response:
          "object" :{{
            "data_points":[]
            "centre":{{}}
          }}
          "text_generated" : ""
          JUST ANSWER JSON and NOTHING ELSE
          """
  # ♦ Invoke Agent with Memory
  final_state = app.invoke({"messages": [system_message, {"role": "user", "content": query}]})
  final_response = '{"data":'+final_state["messages"][-1].content+ f""" ,"data_polygon":{{}}}}"""

# # ♦ Save Interaction to Memory
# memory.save_context({"input": query}, {"output": final_state["messages"][-1].content})

return final_response.replace("```json", "").replace("```","")
```

```python
user_queries = [
    "Find EV charging stations in Berlin",
    "How many greenery land in Hamburg?",
    "Find waste recycle facility in Bremen",
    "Show me the location with the highest CO level in Bayern",
    "Retrieve news related to Berlin",
    "Find me highest population location in Hessen",
    "Find me highest population location in Munich",
]
```

```python
response = query_graph(user_queries[0])
print(response)
```


```
      "lat": 9.032560181821786,
      "lon": 50.103534971766315
    },
    "value": 3.549344845810154e-11
  },
  {
    "location": {
      "lat": 9.086459098868957,
      "lon": 49.96878767914839
    },
    "value": 3.54902739141405e-11
  },
  {
    "location": {
      "lat": 9.131374863074932,
      "lon": 49.959804526307195
    },
    "value": 3.5485413218960815e-11
  },
  {
    "location": {
      "lat": 9.131374863074932,
      "lon": 49.950821373466
    },
    "value": 3.5485413218960815e-11
  },
  {
    "location": {
      "lat": 9.131374863074932,
      "lon": 49.932855067783606
    },
    "value": 3.5484420957132556e-11
  }
  ],
  "centre": {}
 },
 "text_generated": "The location with the highest CO level in Bayern is at latitude 9.0505 and longitude 50.0946, with a CO value of approximately 3.55e-11."
}
,"data_polygon":{'type': 'MultiPolygon', 'coordinates': [[[[13.840348, 48.771848], [13.840177, 48.771222], [13.840547, 48.76788], [13.840453, 48.767016], [13.839958, 48.766436], [13.839958, 48.766389], [13.839831, 48.766288],
```

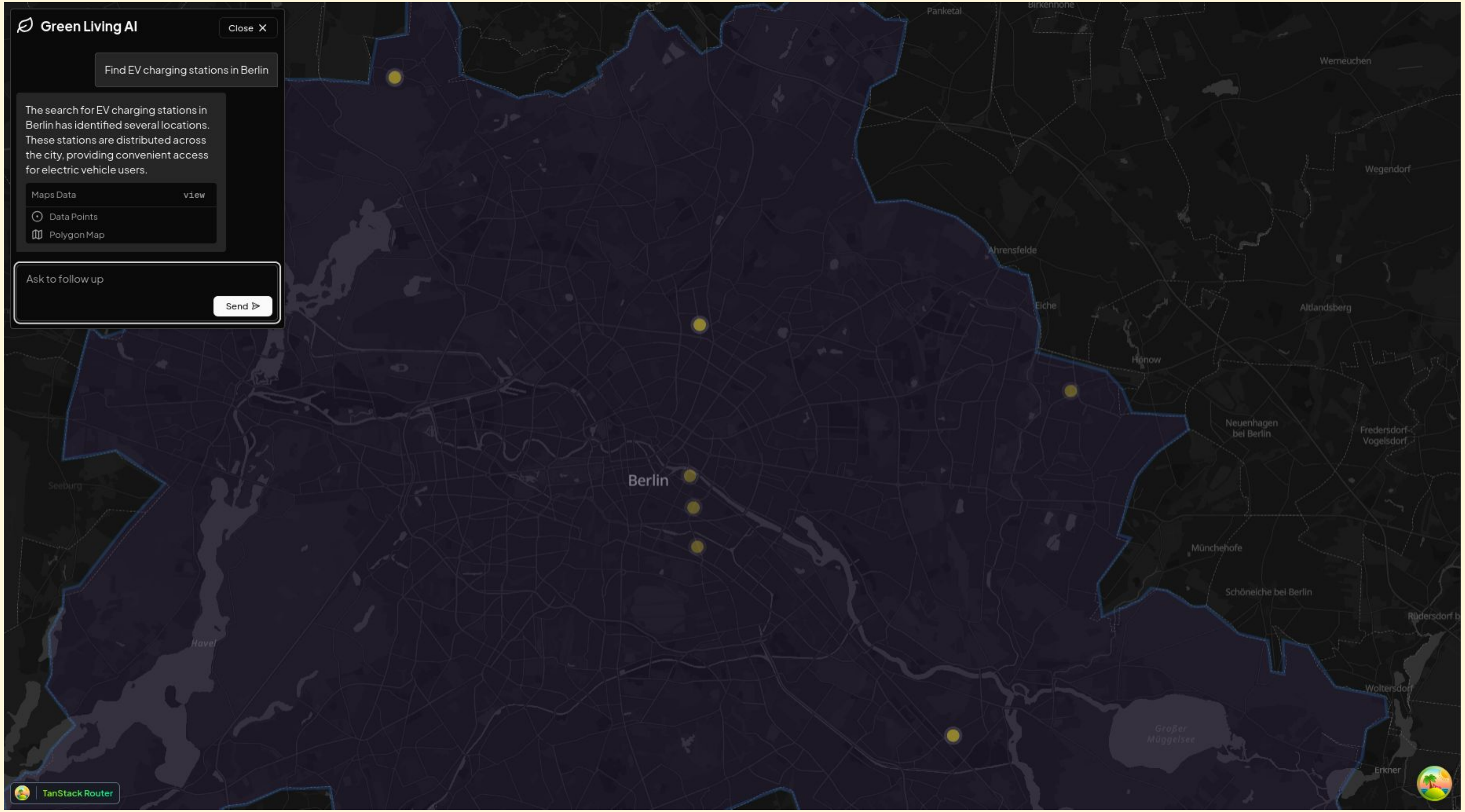
07

Application Demo

Web Application

We create web application to visualize our result in map based format.

User can write the prompt and we will query the data based on the data in our graph database.



THANK YOU