



**EDB**  
Postgres® for the AI Generation

# Integrating AI with Postgres

Bilge Ince  
MLE



**Prague PostgreSQL Developer Day**



Bilge INCE

MLE @ EDB

Organizer of Diva: Dive Into AI  
Muay Thai, Running ❤️



@abilgegunduz



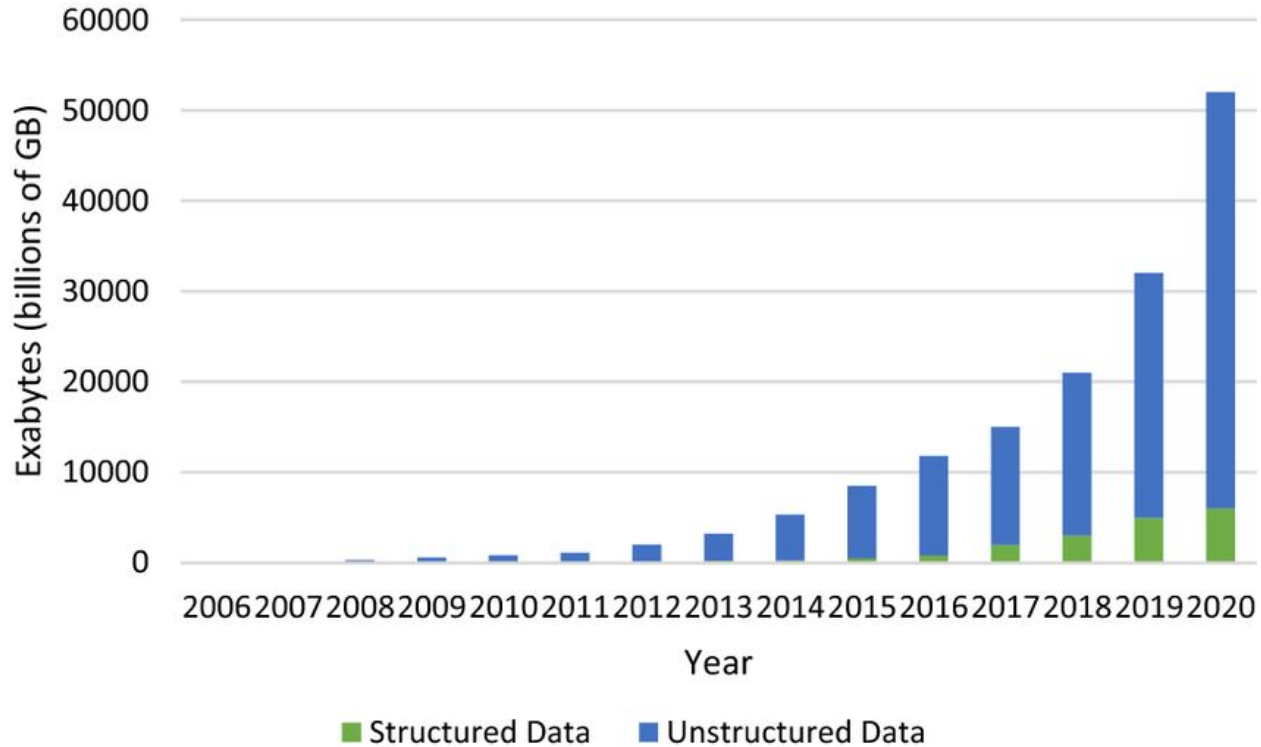
@bilge.bsky.social



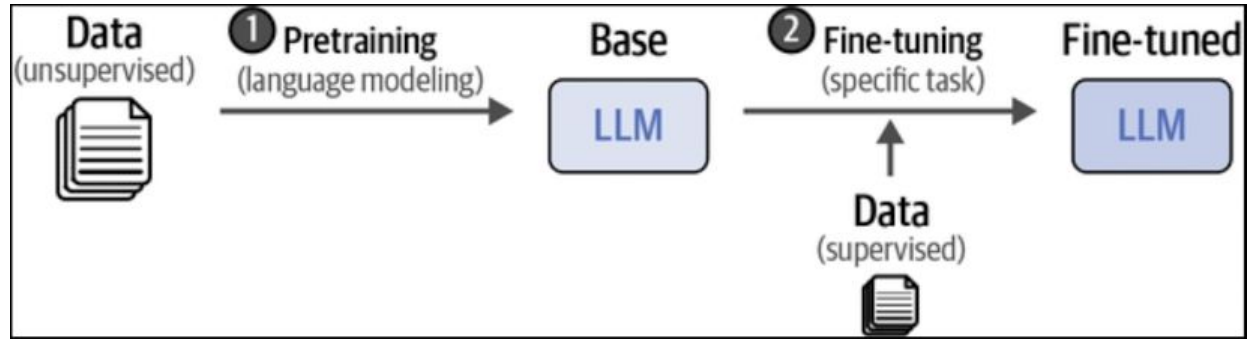
# What is PostgreSQL for an AI Engineer?



# Data in Years



# Traditional ML vs LLMs



# LLMs

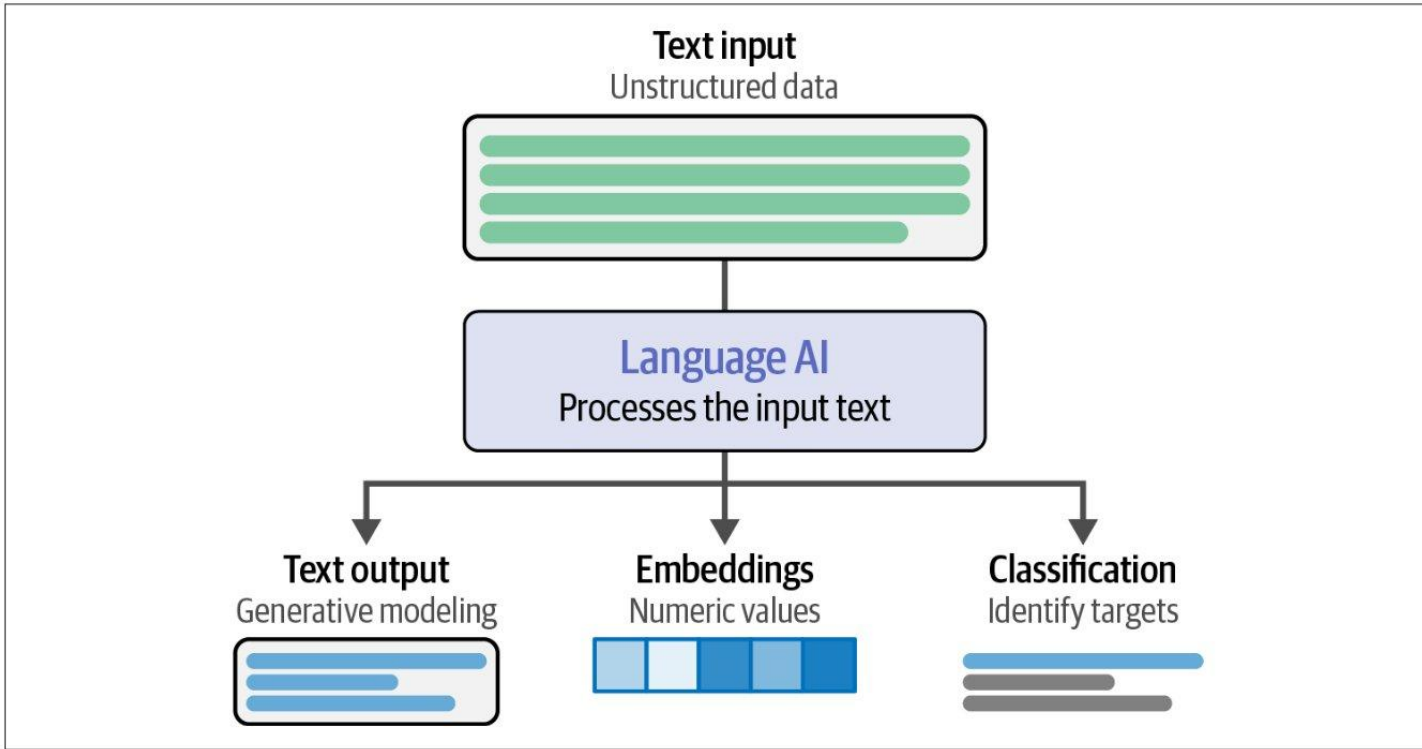


Fig: Hands on LLMs - Jay Alammar & Maarten Grootendorst



# LLMs

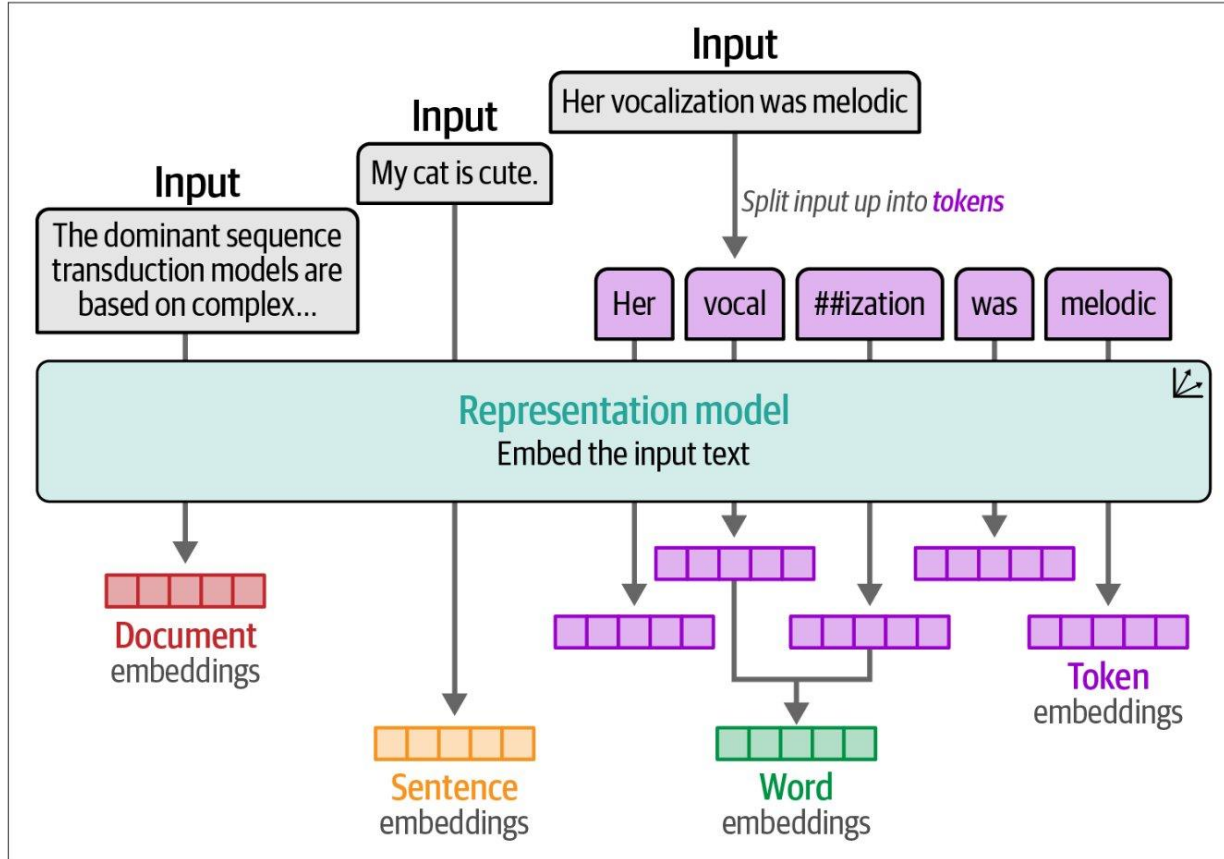
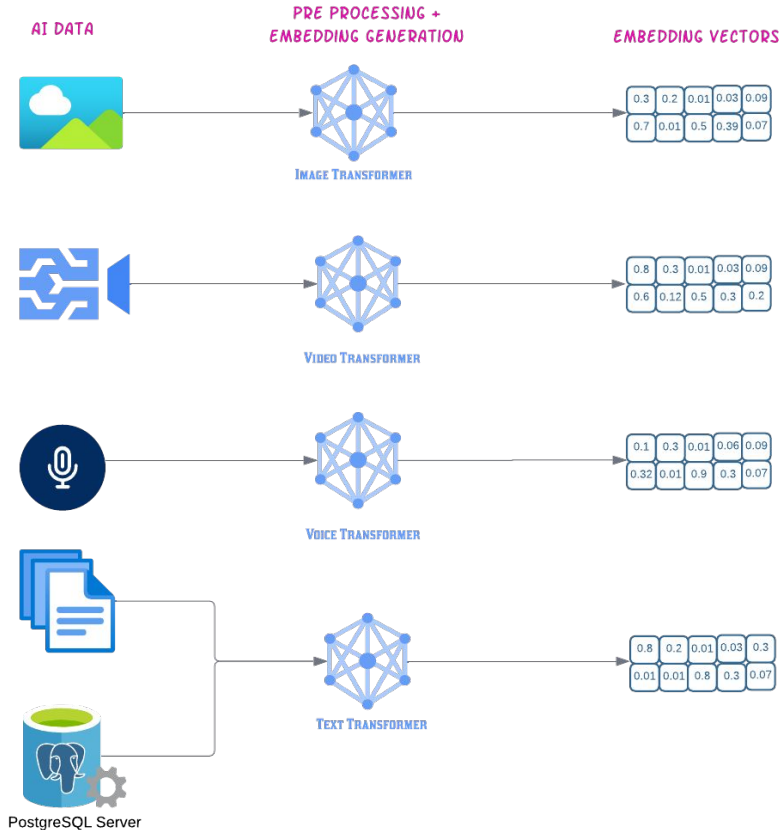


Fig: Hands on LLMs - Jay Alamar & Maarten Grootendorst

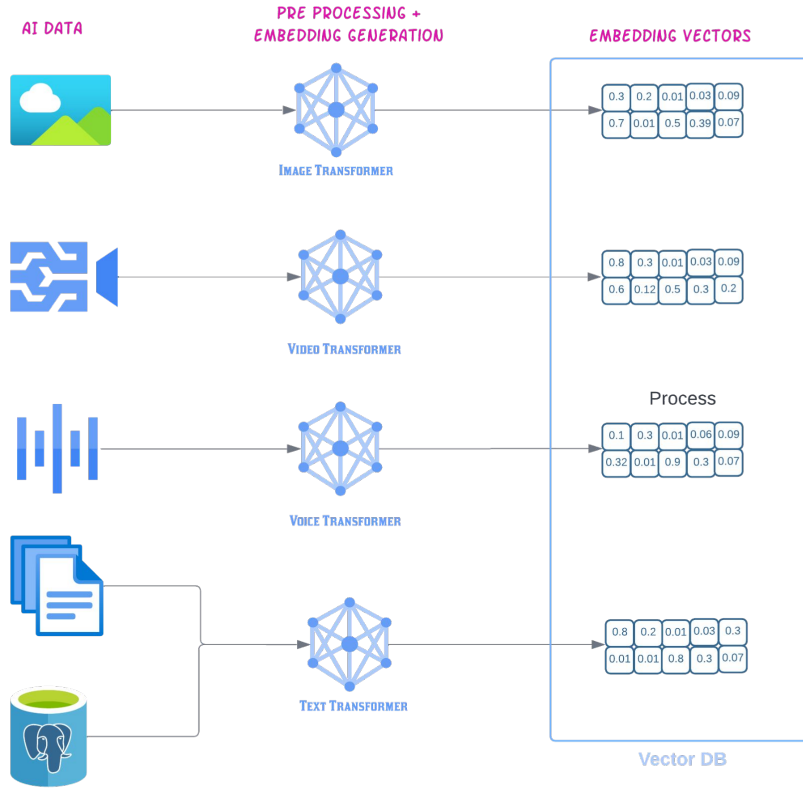


# How data & vectors are connected to each other?

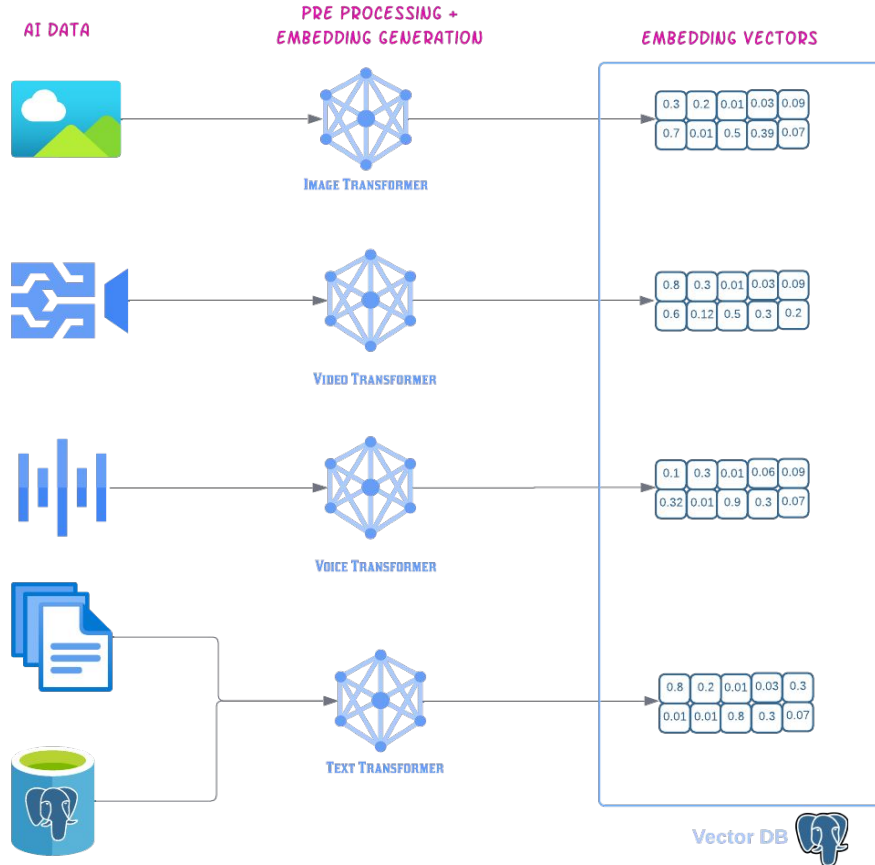




# How data & vectors are connected to each other?



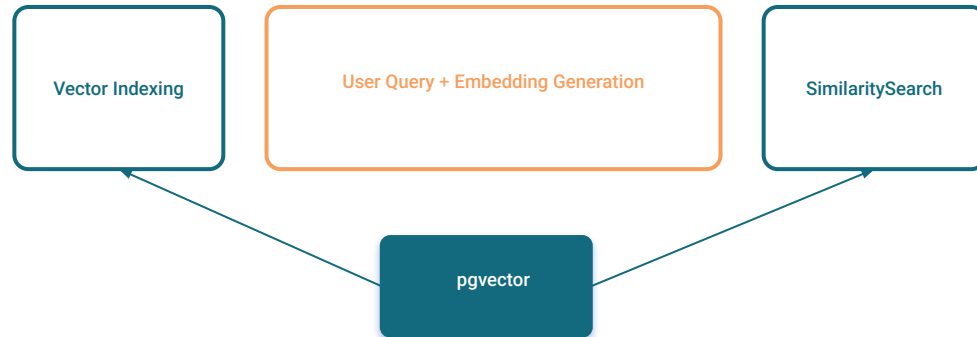
# How data & vectors are connected to each other?



What is PostgreSQL after pg vector extension  
for an AI Scientist?



# PgVector



# Similarity Search



0.3 0.5 0.01 0.08 0.09



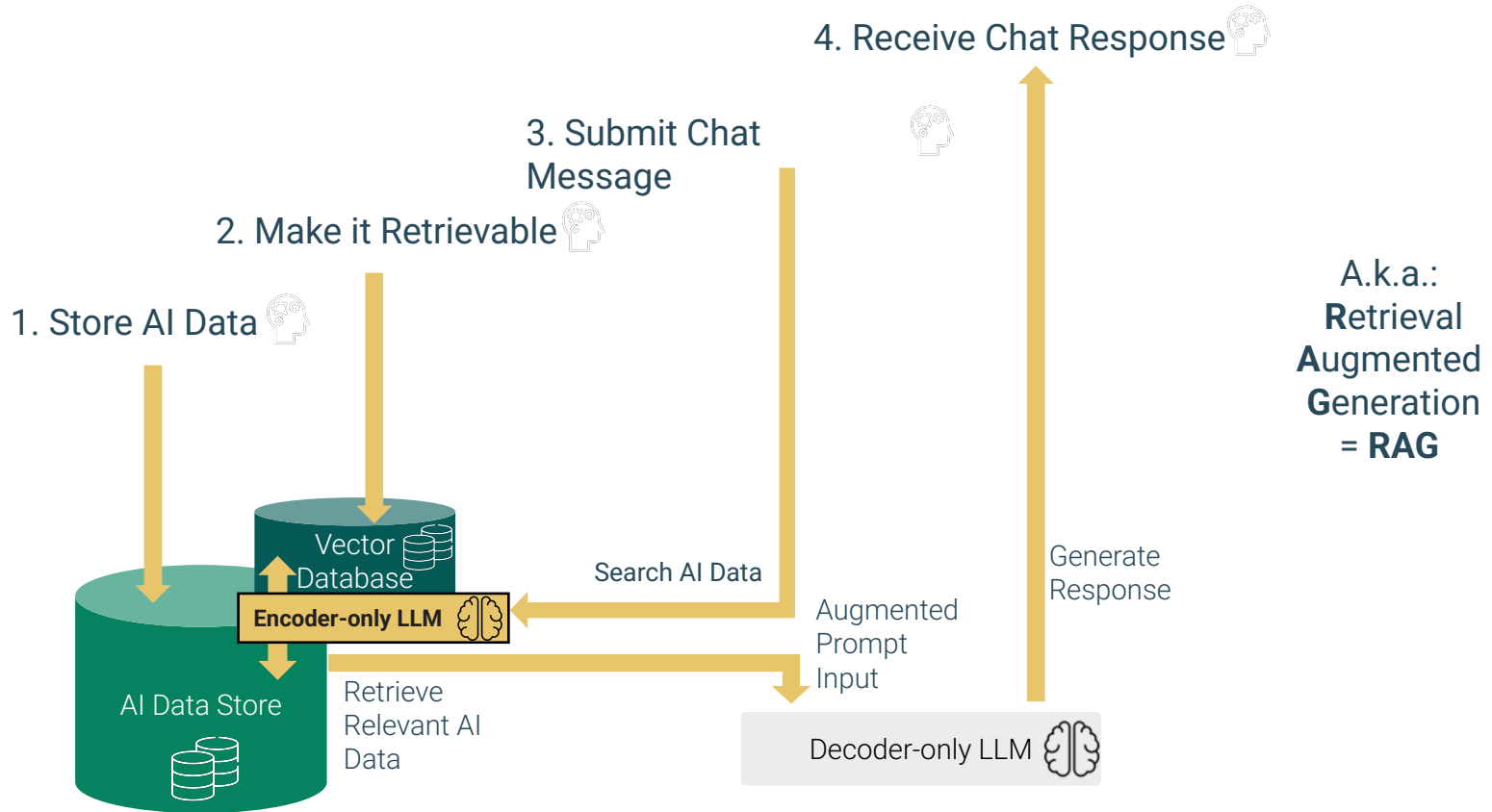
0.3 0.2 0.01 0.03 0.09

distance - Euclidean sq

$$(0.3 - 0.3)^2 + (0.5 - 0.2)^2 + (0.01 - 0.01)^2 + (0.08 - 0.03)^2 + (0.09 - 0.09)^2$$



# Chat Bots – The John Doe of Gen AI Applications



# Postgres is perfectly positioned as THE AI database

- Absolute **battle proof** Enterprise QoS

- In **community** distro but also **very vital** commercial Enterprise option ecosystem



- Perfect **extensibility** & customization

- With **AI relevant** languages & ecosystems: **Python, Rust**
- **Custom Data** Types
- Index & Table **Access Methods**

- Already houses the most valuable enterprise **business data**

- in **fully relational** manner



What brings aidb?



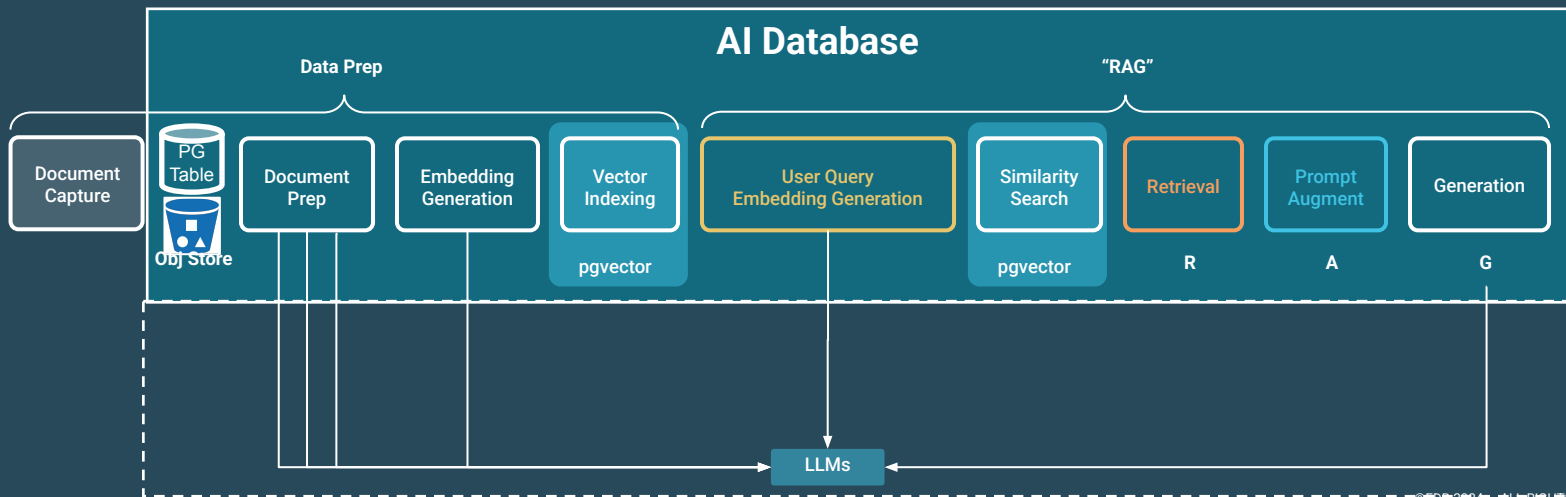


# Building GenAI applications with EDB Postgres AI

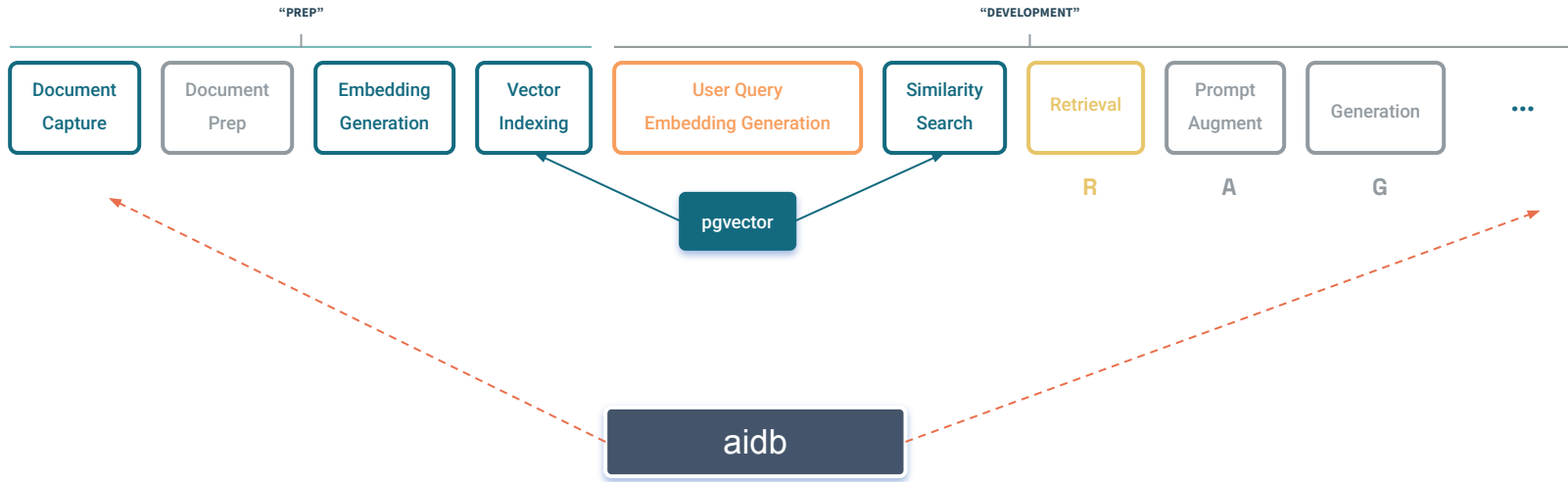
BEYOND VECTOR SUPPORT

- 1 Postgres as GenAI Retriever & Generator:**  
Automating document (and other modalities) prep, embedding generation & vector indexing, providing a simple semantic retriever interface, and even chat completion in database

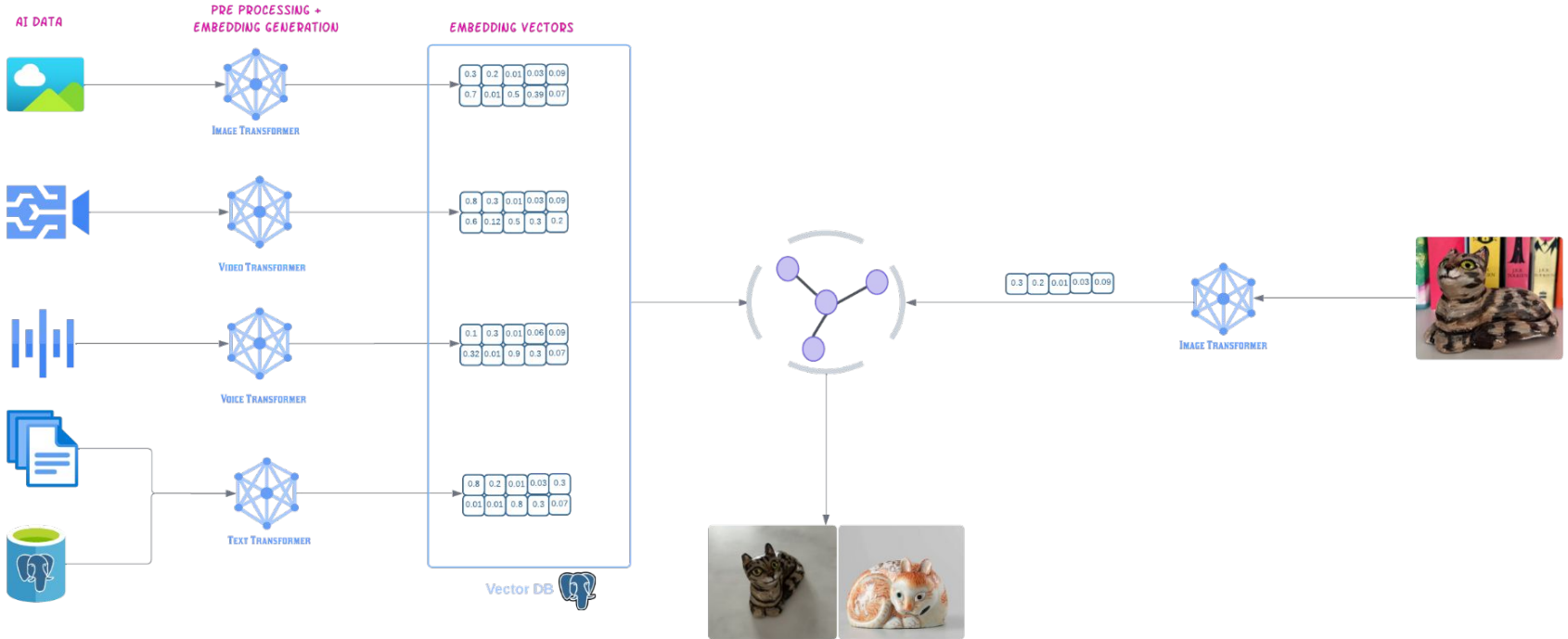
- 2 Enabling Sovereign AI for enterprises:**  
Runs with either, embedded LLMs (in PG memory), external model provider of your choice, or EDB Postgres AI platform hosted models.



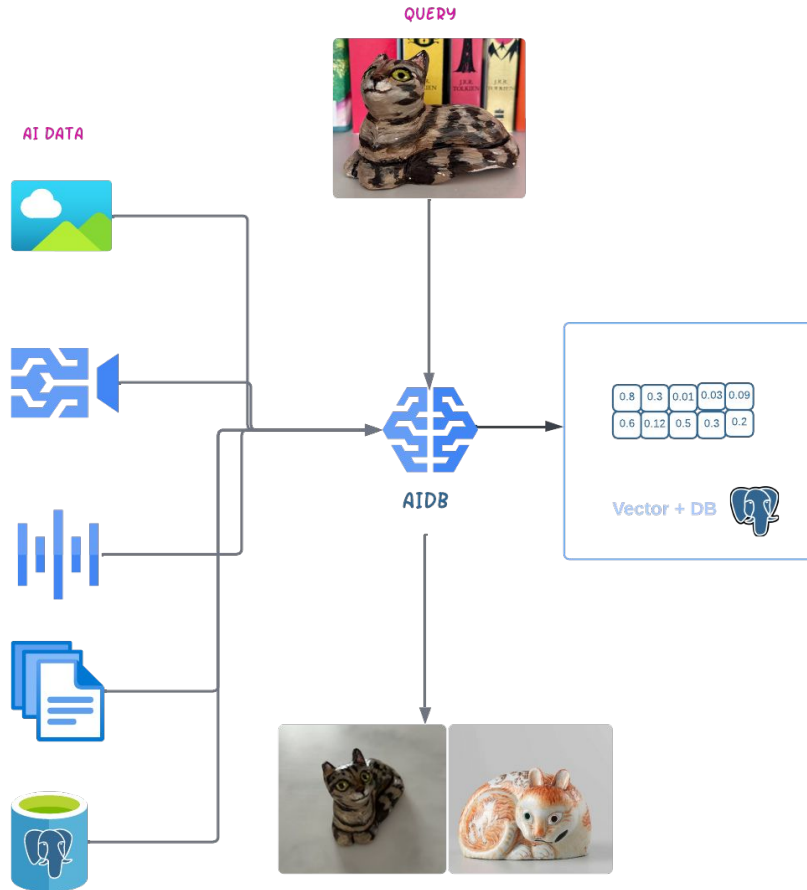
# aidb



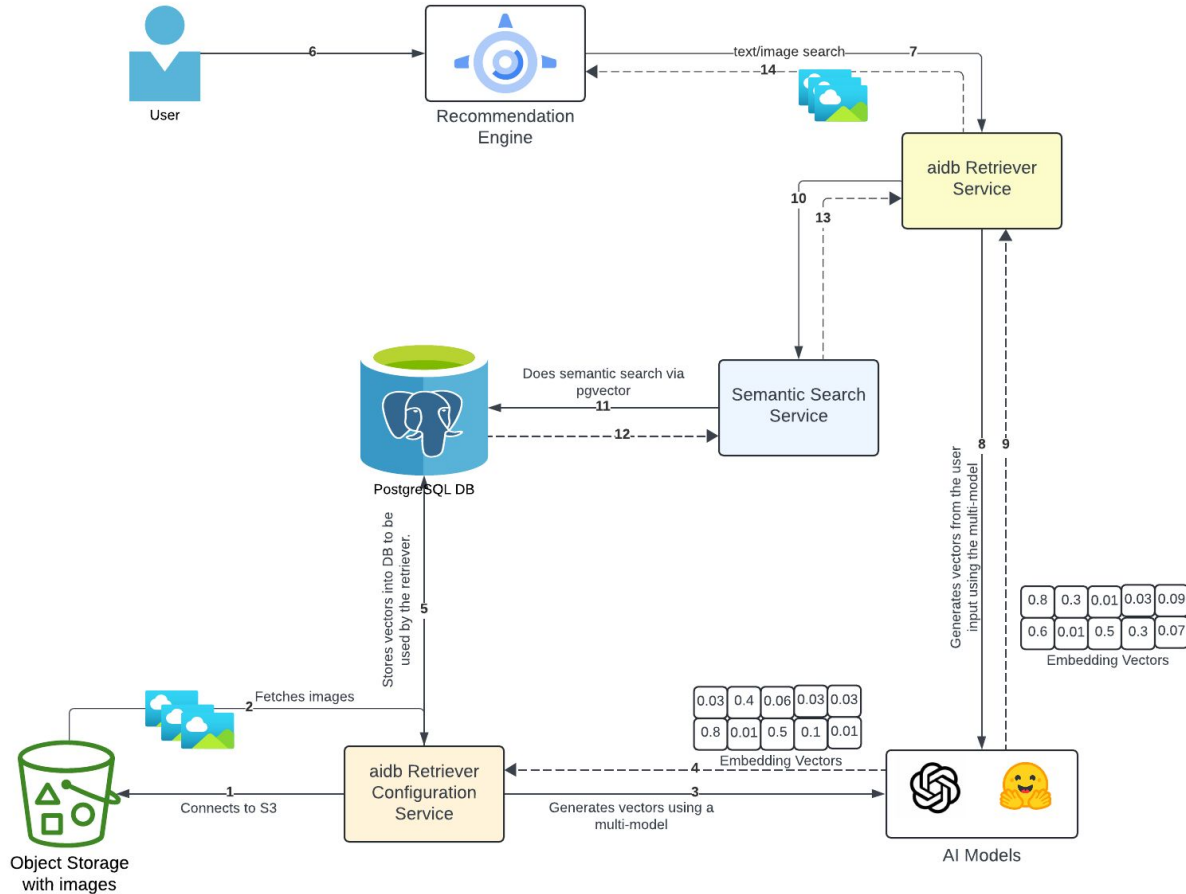
# A recommendation engine with pgvector



# A recommendation engine with aidb




# A recommendation engine as an app



# Demo

localhost

Google Wikipedia Twitter LinkedIn My Apps Dashboard | EDB

 [Products](#) [Solutions](#) [Resources](#) [Company](#)

## Recommendation Engine

Powered by EDB Postgres and AIDB


Select a Category:

Enter search term:


Or upload an image to search:

Drag and drop file here   
Limit 200MB per file • JPG, JPEG, PNG

**2go Active Gear USA Men Pack of 3 White Socks**



**2go Active Gear USA Men Pack of Two Cushion Socks**



# Implementation with pgvector

132 lines of code without, vs 5 line of function with ai db

```
function_start_time = time.time()

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
model_loading_end = time.time()

fetch_start = time.time()
cursor = conn.cursor()
cursor.execute("SELECT id, gender, mastercategory, subcategory, articletype, basecolour, season, year, usage, productdisplayname FROM products;")
result = cursor.fetchall()
fetch_end = time.time()

batch_size = batch
total_rows_inserted = 0
total_image_processing_time = 0

for i in range(0, 50, batch_size):
    batch_ids = [row[0] for row in result[i:i+batch_size]]
    inputs, valid_paths = load_images_batch(batch_ids, base_path, processor, tag)
    if inputs is not None:
        image_processing_start_time = time.time()
        outputs = model(*inputs)
        image_processing_end_time = time.time()
        embeddings = outputs.image_embeds
        image_processing_time = image_processing_end_time - image_processing_start_time
        total_image_processing_time += image_processing_time

        embeddings_list = embeddings.detach().cpu().numpy().tolist()

        with conn.cursor() as cursor:
            for idx, embedding in enumerate(embeddings_list):
                row = result[i + idx]
                image_path = valid_paths[idx]
                cursor.execute(
                    """INSERT INTO products_emb
                    (id, gender, mastercategory, subcategory, articletype, basecolour, season, year, usage, productdisplayname, image_path, embedding) """
                    "VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)",
                    (row[0], row[1], row[2], row[3], row[4], row[5], row[6], row[7], row[8], row[9], image_path, embedding)
                )
            total_rows_inserted += 1

function_end_time = time.time()
total_time = function_end_time - function_start_time
print(f"Total Rows: {total_rows_inserted}")
print(f"Total function execution time: {total_time} seconds")
print(f"Model loading time: {model_loading_end - model_loading_start} seconds")
print(f"Fetching time: {fetch_end - fetch_start} seconds")
```



# Implementation with aidb

132 lines of code without, vs 2 line of function with aidb

```
cur.execute(f"""
    SELECT aidb.create_s3_retriever(
        '{retriever_name}',
        'public',
        'clip-vit-base-patch32',
        'img',
        '{s3_bucket_name}',
        '',
        '{s3_endpoint}'
    );
""")
cur.execute(f"SELECT aidb.refresh_retriever('{retriever_name}');")
```





# Implementation with pgvector

70 lines of code without, vs 1 line of function with aidb

```
CREATE OR REPLACE FUNCTION generate_embeddings_clip_bytea(
    bytes_data bytea)
RETURNS SETOF vector
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000
AS $BODY$
```

```
import sys
import os
path = '{}/lib/python{}/./site-packages'.format(
    os.environ['VIRTUAL_ENV'],
    sys.version_info.major,
    sys.version_info.minor
)
sys.path.append(path)
from PIL import Image
from transformers import CLIPModel, CLIPProcessor
import numpy as np
from io import BytesIO # Import BytesIO to handle bytea input

# Define the model and processor outside the loop to avoid reloading them for each image
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

# Convert the bytea data to a bytes-like object and load the image
img_bytes = BytesIO(img_bytea)
img = Image.open(img_bytes)

# Process the image and calculate embeddings
inputs = processor(text=[tag], images=img, return_tensors="pt")
outputs = model(**inputs)
embedding = outputs.image_embeds

# Convert embeddings to a list to store in the database
embeddings_list = embedding.tolist()

return embeddings_list
$BODY$;
```

```
query = text(
    "SELECT public.generate_embeddings_clip_bytea(:bytes_data, 'person':text);"
)
with engine.connect() as connection:
    vector_result = connection.execute(query, {"bytes_data": bytes_data})
    data = [
        {"generate_embeddings_clip_bytea": row["generate_embeddings_clip_bytea"]}
        for row in vector_result.mappings().all()
    ]

# If you expect a single embedding or a single row, extract it
if data:
    # If there's only one row, return the first row's data
    return data[0]
```

```
query = text(
    """"SELECT id, productDisplayname, image_path FROM products_emb
    ORDER BY (embedding <=> :vector_result) LIMIT 2;""")
)
if isinstance(
    vector_result, list
): # If it's a list, format it as a string that PostgreSQL understands
    vector_result = "[" + ",".join(map(str, vector_result)) + "]"

with engine.connect() as connection:
    result = connection.execute(query, {"vector_result": vector_result})
    data = [
        {
            "id": row["id"],
            "name": row["productdisplayname"],
            "image_path": row["image_path"],
        }
        for row in result.mappings().all()
    ]
```



# Implementation with aidb

70 lines of code without, vs 1 line of function with aidb

```
CREATE OR REPLACE FUNCTION generate_embeddings_clip_bytes
RETURNS SETOF vector
LANGUAGE SQL
COST 100
VOLATILE PARALLEL UNSAFE
ROWS 1000
AS $$
import sys
import os
path = "{0}/lib/python{1}-0/site-packages".format(
    os.environ["VIRTUAL_ENV"],
    sys.version_info.major,
    sys.version_info.minor
)
sys.path.append(path)
try
    cur.execute(
        f"""SELECT data from
aidb.retrieve_via_s3('{st.session_state.retriever_name}', 5, '{st.session_state.s3_bucket_name}', '{image_name}', '{st.session_state.s3_endpoint}');"""
    )
# Before the model and processor outside the loop to avoid reloading them for each image
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
# Convert the bytes data to a bytes-like object and load the image
img_bytes = BytesIO(img_bytes)
img = Image.open(img_bytes)
# Process the image and calculate embeddings
inputs = processor(text=[tag], images=img, return_tensors="pt")
outputs = model(**inputs)
embedding = outputs.image_embeds
# Convert embeddings to a list to store in the database
embeddings_list = embedding.tolist()
return embeddings_list
$$

query = text("SELECT public.generate_embeddings_clip_text(:text_query);")
with engine.connect() as connection:
    vector_result = connection.execute(query, {"text_query": text_query})
    data = [
        ("embedding": row["generate_embeddings_clip_text"])
        for row in vector_result.mappings().all()
    ]
# If you expect a single embedding or a single row, extract it
if data:
    # If there's only one row, return the first row's data
    return data[0]

if isinstance(
    vector_result, list
): # If it's a list, format it as a string that PostgreSQL understands
    vector_result = "[" + ",".join(map(str, vector_result)) + "]"
with engine.connect() as connection:
    result = connection.execute(query, {"vector_result": vector_result})
    data = [
        {
            "id": row["id"],
            "name": row["productdisplayname"],
            "image_path": row["image_path"],
        }
        for row in result.mappings().all()
    ]
}
```



# Implementation with pgvector

55 lines of code without, vs 1 line of function with aidb

```
CREATE OR REPLACE FUNCTION generate_embeddings_clip_text(text_query text)
RETURNS float[] AS
$$
import sys
import os
path = '{}/lib/python{}/{}/site-packages'.format(
    os.environ['VIRTUAL_ENV'],
    sys.version_info.major,
    sys.version_info.minor
)
sys.path.append(path)
import torch
from transformers import CLIPProcessor, CLIPModel

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

inputs = processor(text=[text_query], return_tensors="pt")
inputs = {k: v for k, v in inputs.items()}

with torch.no_grad():
    text_embeddings = model.get_text_features(**inputs).cpu().numpy().tolist()

return text_embeddings[0]
```

```
query = text("SELECT public.generate_embeddings_clip_text(:text_query);")
with engine.connect() as connection:
    vector_result = connection.execute(query, {"text_query": text_query})
    data = [
        {"embedding": row["generate_embeddings_clip_text"]}
        for row in vector_result.mappings().all()
    ]

# If you expect a single embedding or a single row, extract it
if data:
    # If there's only one row, return the first row's data
    return data[0]
```

```
query = text(
    """SELECT id, productDisplayname, image_path FROM products_emb
    ORDER BY (embedding <=> :vector_result) LIMIT 2;"""
)
if isinstance(
    vector_result, list
):
    # If it's a list, format it as a string that PostgreSQL understands
    vector_result = "[" + ",".join(map(str, vector_result)) + "]"

with engine.connect() as connection:
    result = connection.execute(query, {"vector_result": vector_result})
    data = [
        {
            "id": row["id"],
            "name": row["productdisplayname"],
            "image_path": row["image_path"],
        }
        for row in result.mappings().all()
    ]
```



# Implementation with aidb

55 lines of code without, vs 1 line of function with aidb

```
CREATE OR REPLACE FUNCTION generate_embeddings_clip_text(text_query text)
RETURNS float[] AS
$$
import sys
import os
path = '{}\Lib\python().()\/site-packages'.format(
    os.environ['VIRTUAL_ENV'],
    sys.version_info.major,
    sys.version_info.minor
)

sys.path.append(path)
import torch
from transformers import CLIPProcessor, CLIPModel

model = CLIPModel.from_pretrained('openai/clip-vit-large-patch14')
processor = CLIPProcessor.from_pretrained('openai/clip-vit-large-patch14')

inputs = processor(text=[text_query], return_tensors="pt")
inputs = {k: v for k, v in inputs.items()}

with torch.no_grad():
    text_embeddings = model.get_text_features(**inputs).cpu().numpy().tolist()

return text_embeddings[]
$$ LANGUAGE plpython3u;
```

```
query = text|
"SELECT id, productdisplayname, image_path FROM products_emb ORDER BY (embedding <> vector_result) LIMIT 2|"

-- instantiate
vector_result, list
k: if it isn't a list, format it as a string that postgresql understands
vector_result = "0" + "0" * (len(longest_str) - len(vector_result)) + "0"

with engine.connect() as connection:
result = connection.execute(query, {"vector_result": vector_result})
data = [
    {
        "id": row["id"],
        "name": row["productdisplayname"],
        "image_path": row["image_path"],
    }
    for row in result.mappings().all()
]
```

```
query = text|
""SELECT id, productdisplayname, image_path FROM products_emb

with engine.connect() as connection:
result = connection.execute(query, {"vector_result": vector_result})
data = [
    {
        "id": row["id"],
        "name": row["productdisplayname"],
        "image_path": row["image_path"],
    }
    for row in result.mappings().all()
]
```



# Summary

- PostgreSQL was only a relational DB before pgvector for an AI Engineer.
- Data and vector are better together.
- pgVector brought vector capabilities like semantic search in PostgreSQL and fulfilled the Vector DB needs as well as relational DB.
  - Also it's hard to install and it's complicated for someone who don't know AI and PostgreSQL.
- aidb brings simplicity and hides complexity without compromising from capabilities.



Thank You!

