EDB POSTGRES AI

JPMorganChase

Building an AI Factory with EDB Postgres AI

Bilge Ince
Senior Machine Learning Engineer
February 2026

Speaker Introduction

Bilge Ince

Senior Machine Learning Engineer

Bilge Ince is a Machine Learning Engineer at EDB, working at the intersection of AI and PostgreSQL to build practical, production-ready AI systems for developers.

She holds a Master's in Computer Science from Istanbul Technical University and co-organizes Diva: Dive Into AI, a conference.

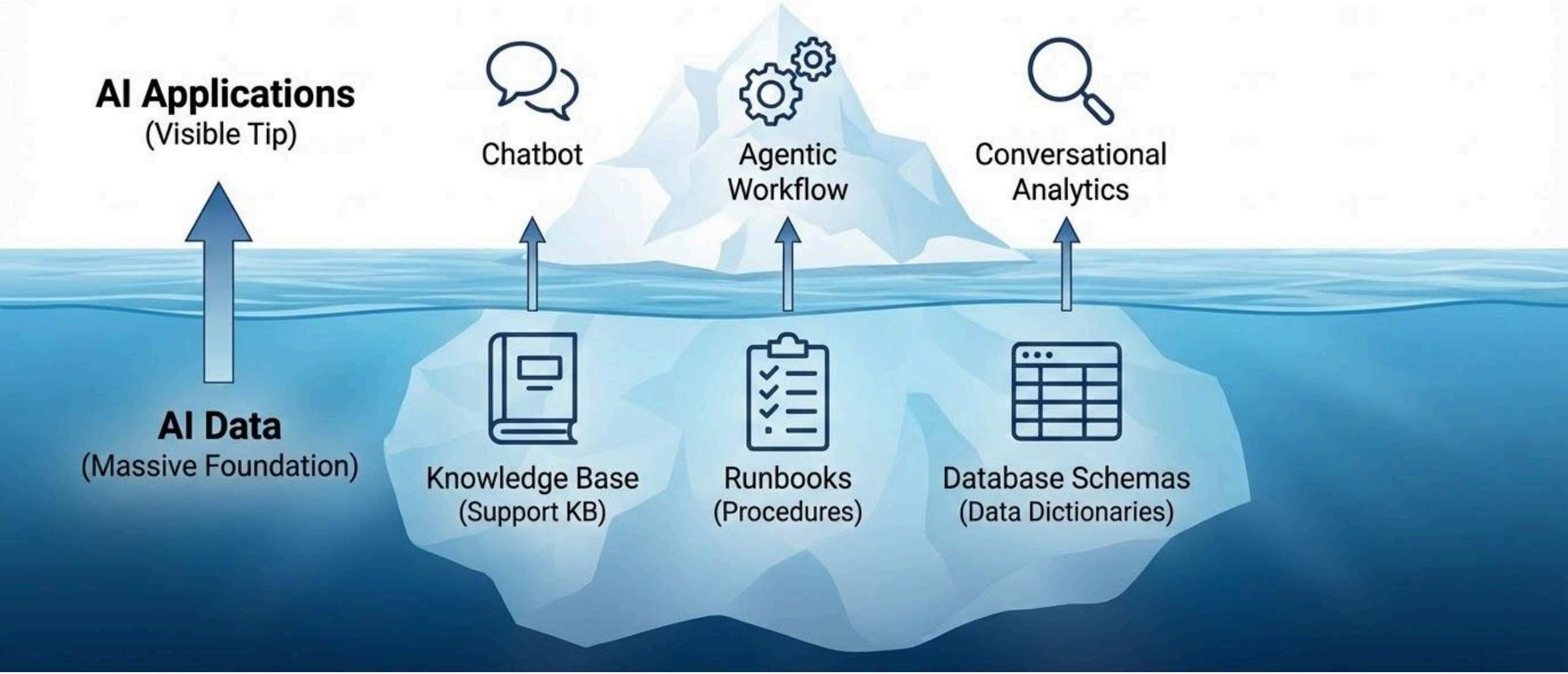
Connect on Linkedin

Agenda

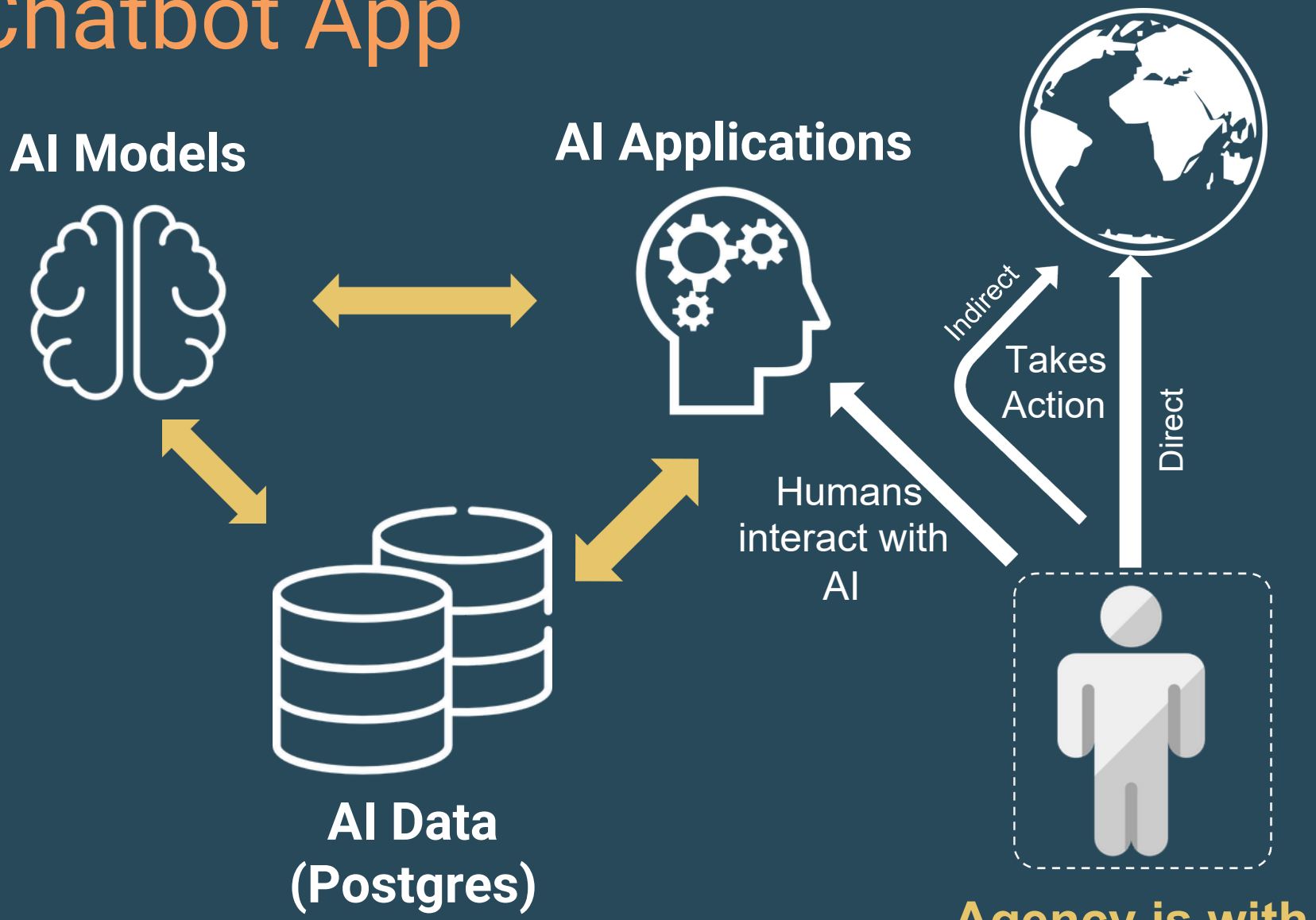
- Generative AI Applications and Knowledge
- Approaches to Maintaining Knowledge in a GenAI Application
- Vector Engine and the Benefits of Using Postgres as a Vector Database
- Ingesting Data into a Postgres Knowledge Base with EDB Postgres AI Pipelines
- The Future: Agentic AI & Unified Architecture

The Knowledge-First AI Stack

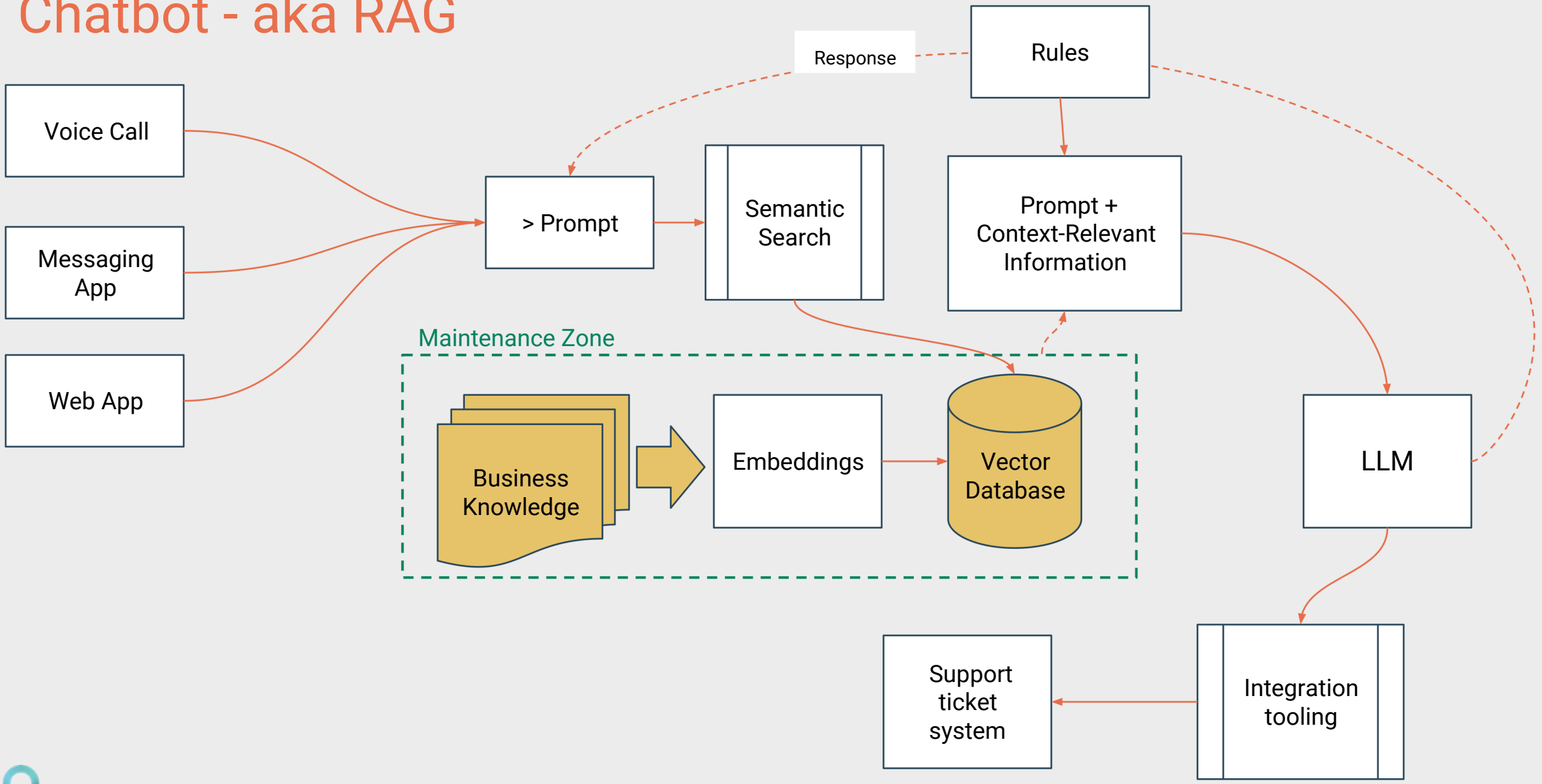
The Knowledge-First AI Stack



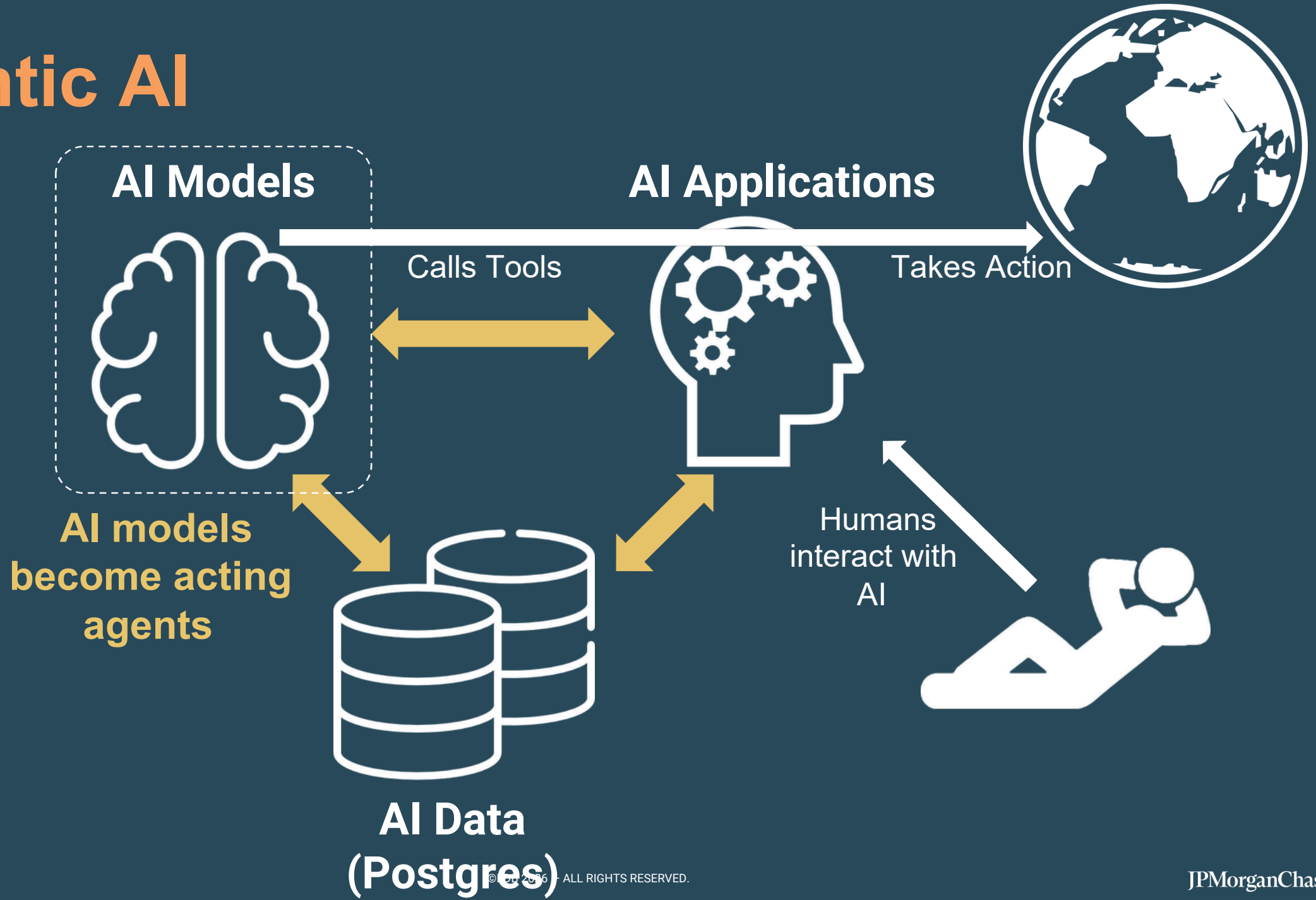
Typical Chatbot App



Chatbot - aka RAG



Agentic AI



Integrating Postgres in Agentic Workflows

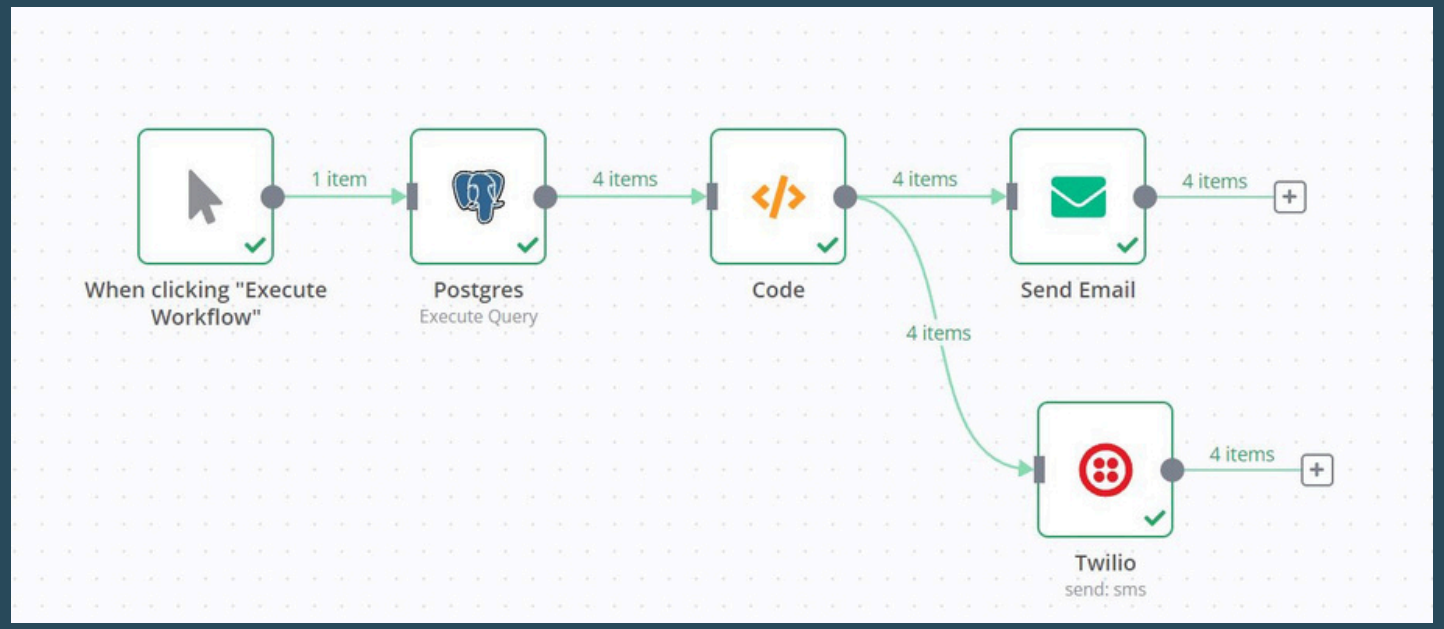
AI Agent Frameworks

Low-Code:

- N8N, LangFlow, Zapier, FlowiseAI, Dify

Python

- LlamaStack, Langchain, Pydantic AI, LangGraph

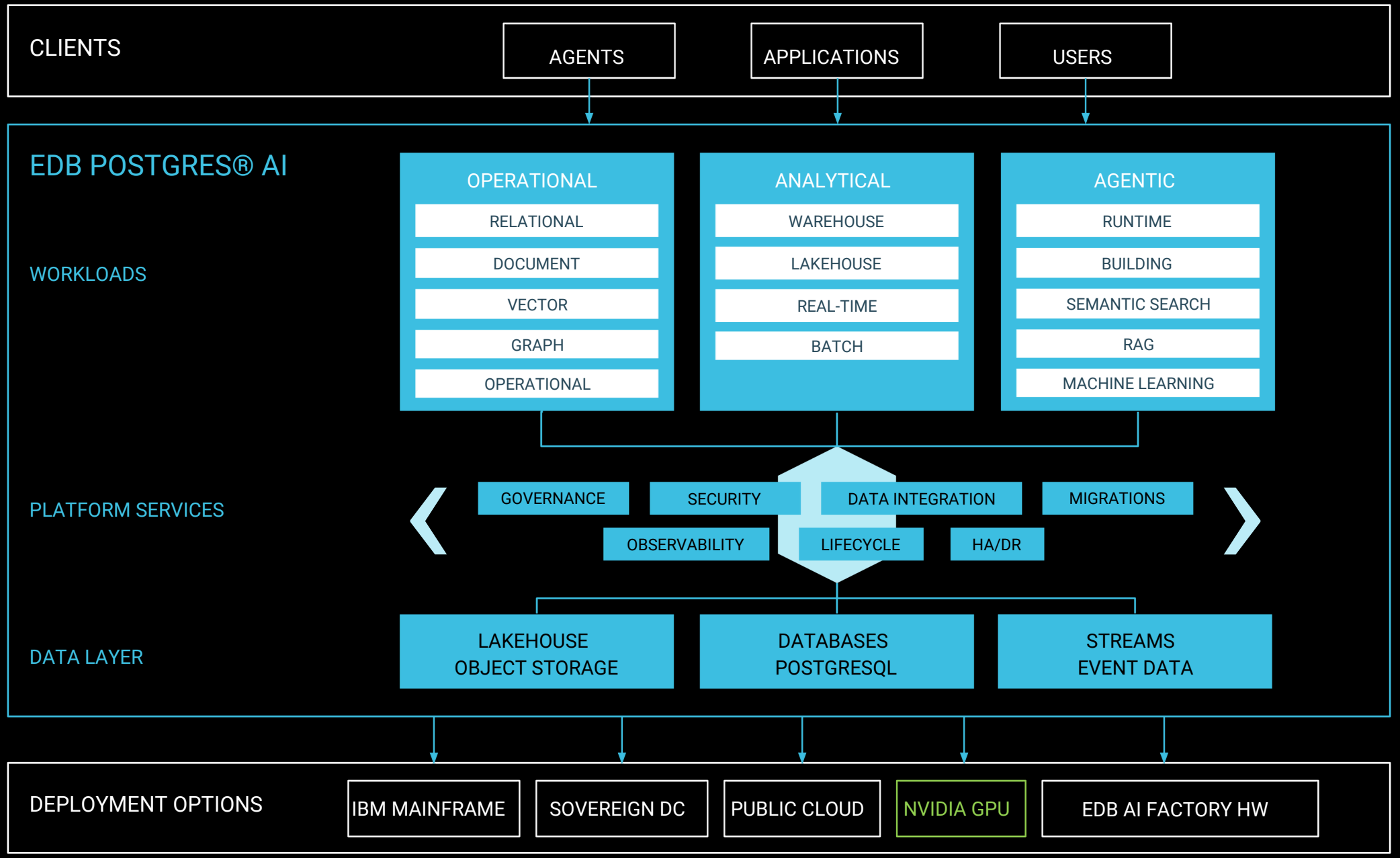


Agentic tools for Postgres

- Retrieving Schema
- Running DML & DDL
- Vector search & retrieval
- Index tuning
- Backup & Restore
- etc. etc.

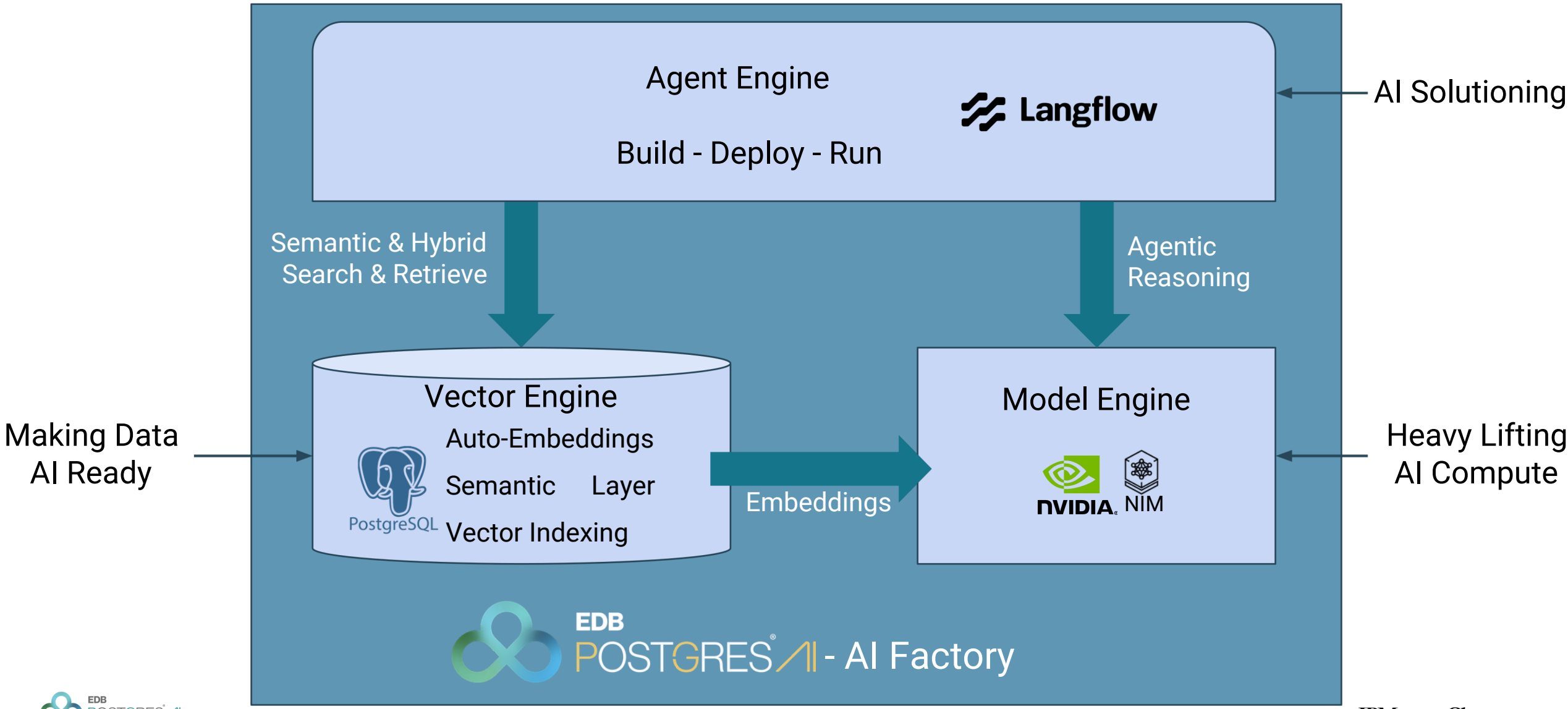
Tool Calling Standard

- MCP – Model Context Protocol
- Such as pg-airman-mcp

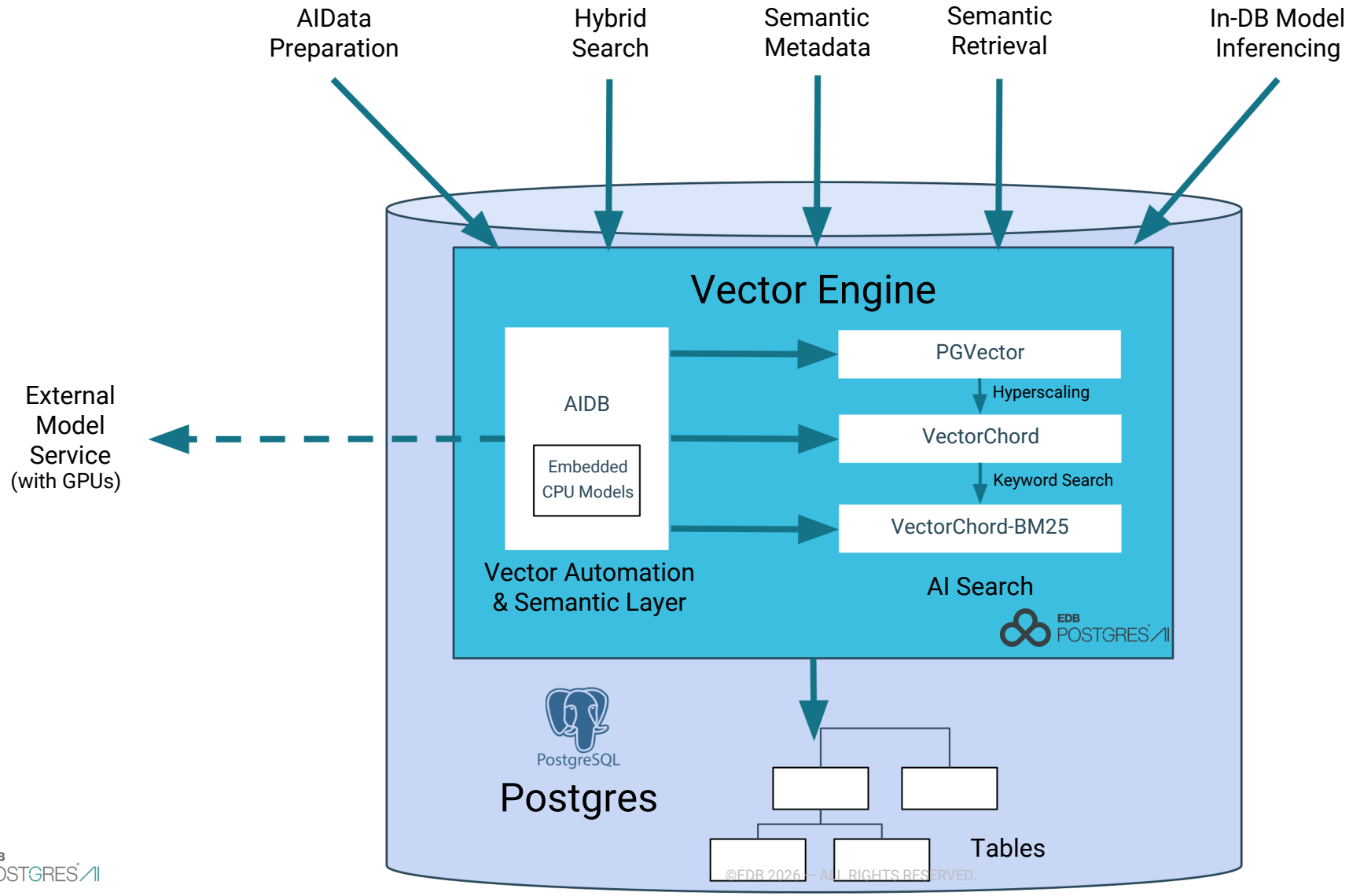


EDB POSTGRES AI

Double Click on AI Factory



EDB Vector Engine - Overview

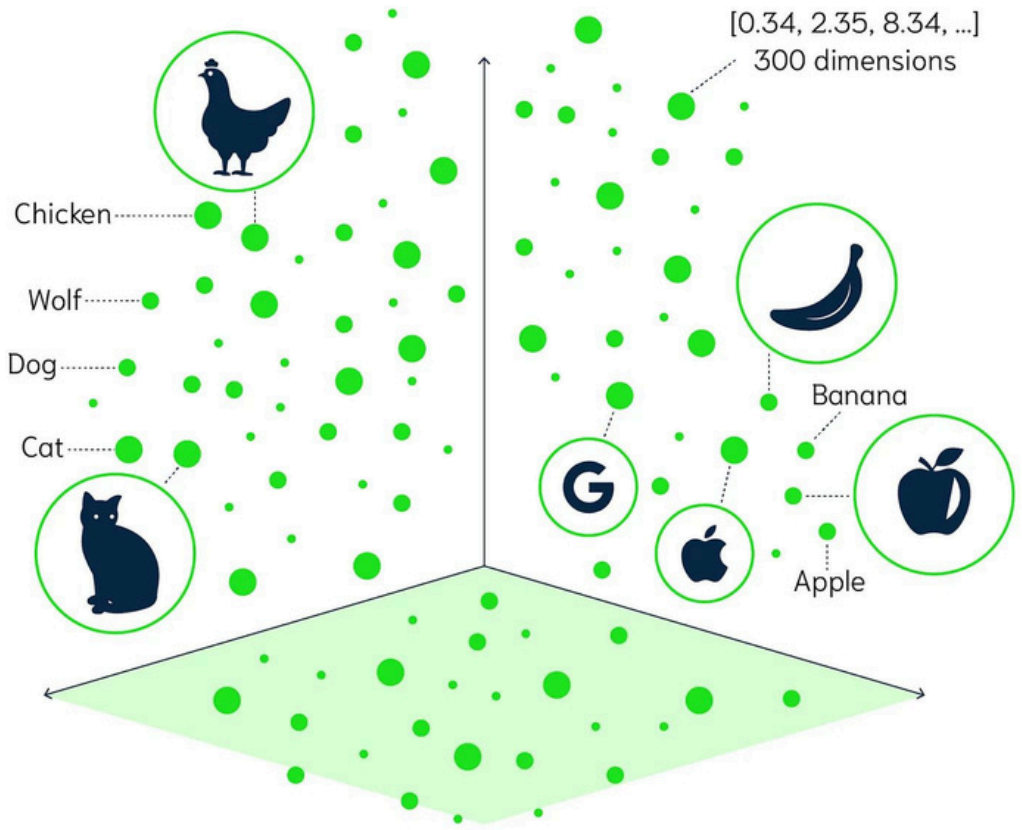


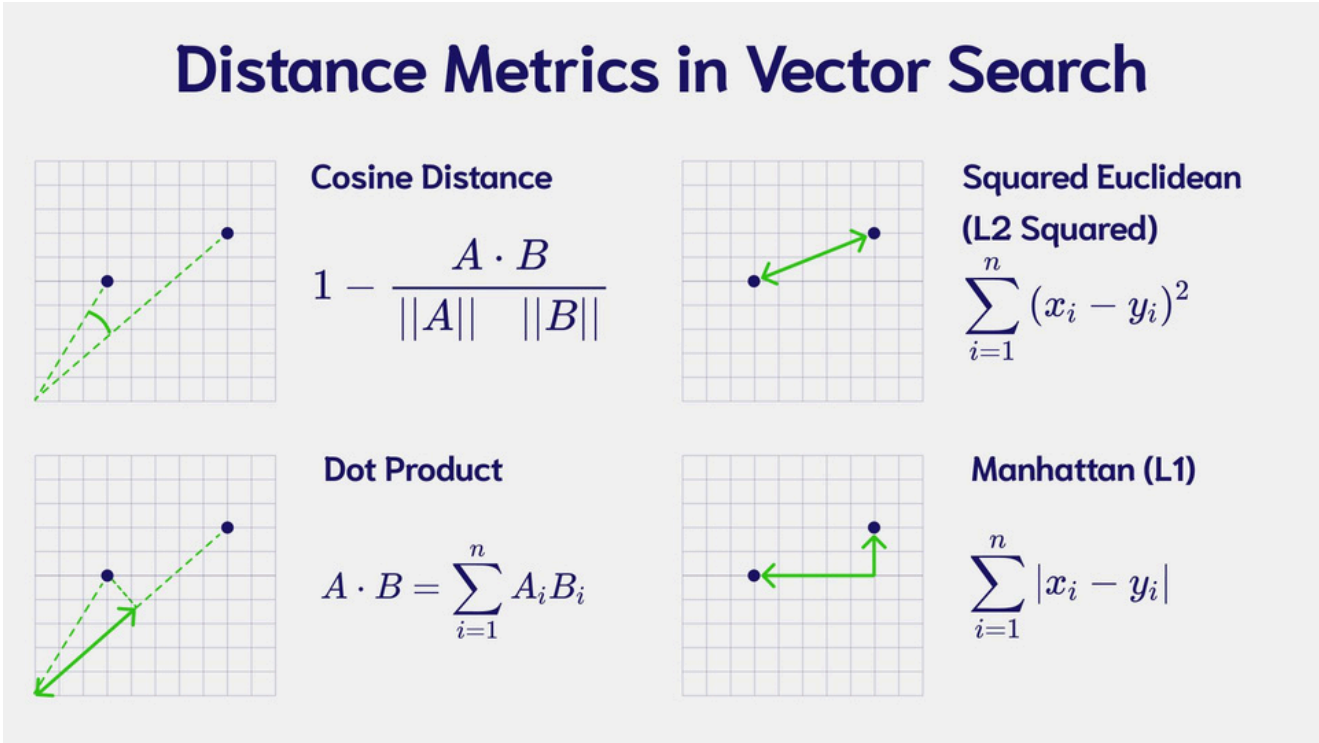
JPMorganChase

Vector Database 101

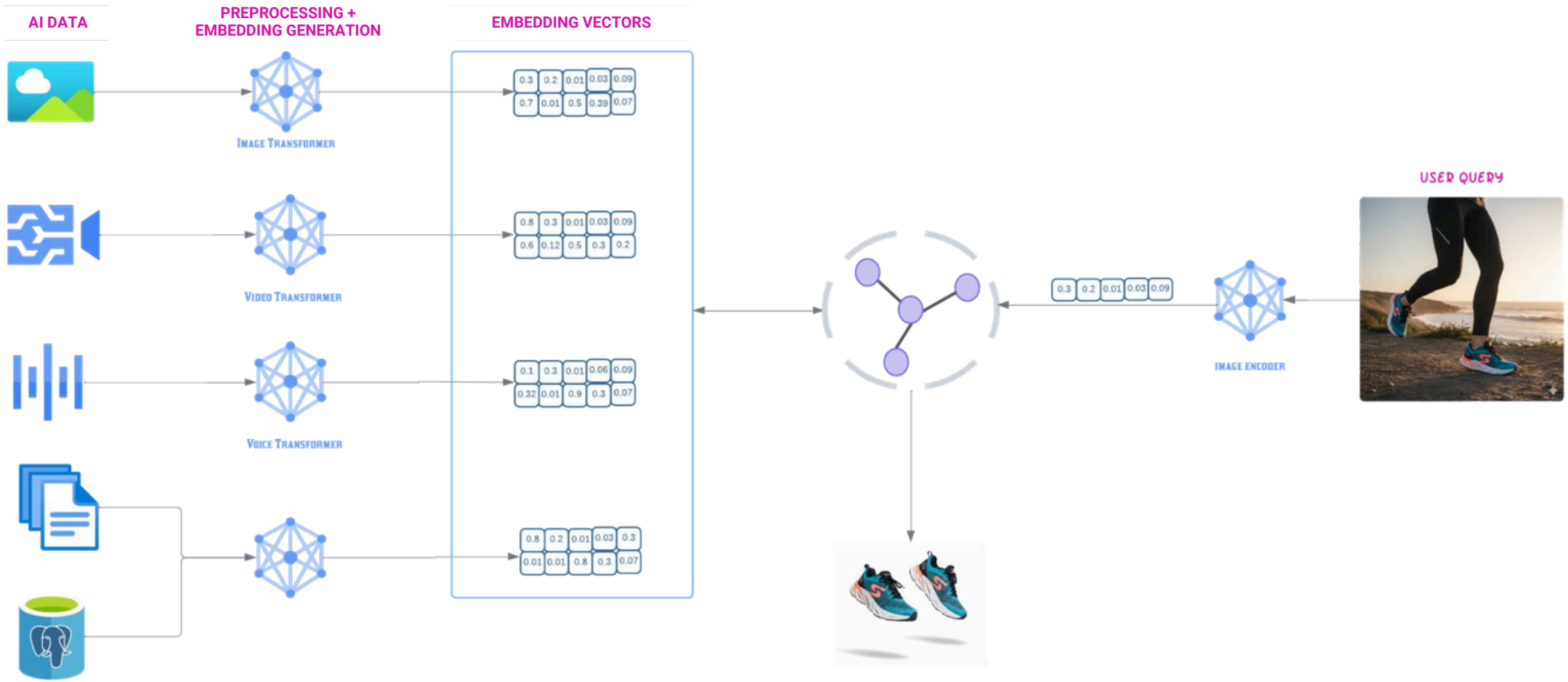
A **vector database** is a database adapted to store **embeddings** for semantic search.
Embeddings are dense, high-dimensional vectors that map unstructured data into a latent space, representing semantic relationships through geometrical proximity.

Distance Metrics in Vector Search





It’s Just SQL!



Find shoes similar to this embedding:

```
SELECT product_id FROM products_embedding ORDER BY
embedding <-> '[0.3,© ED0B 2.026 ,—A LL0 R.IGH0TS0 R1ES,ER VE0D..03, 0.09]'LIMIT 5;
```

pgvector

pgvector is an **extension** for PostgreSQL that enables vector similarity search within a PostgreSQL database.

- Exact and approximate nearest-neighbor search
- Single-precision, half-precision, binary, and sparse vectors
- L2 distance, inner product, cosine distance, L1 distance, Hamming distance, and Jaccard distance
- Any language with a Postgres client + ACID compliance, point-in-time recovery, JOINs, and all of the other great features of **PostgreSQL**

pgvector Search Syntax

```
-- Enable thepgvector extension
CREATE EXTENSION IF NOT EXISTS vector;

-- Create a table for storing items and their embeddings
CREATE TABLE items (
    id serial PRIMARY KEY,
    name text,
    embedding vector(1000) -- A vector with 1000 dimensions
);
```

Creating the extension and a table

```
INSERT INTO items (name, embedding) VALUES
('Apple', '[0.1, 0.2, 0.3]'),
('Banana', '[0.9, 0.8, 0.7]'),
('Orange', '[0.15, 0.25, 0.35]');
```

Inserting embeddings into the table (an AI model is needed to create the embeddings)

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops);
```

Creating a vector index on the embeddings for performance

```
-- Search for the closest match using L2 distance
SELECT name, embedding
FROM items
ORDER BY embedding <-> '[0.12, 0.22, 0.32]'
LIMIT 1;
```

Running a semantic search (an AI model is needed to create the search term embeddings)

pgvector at Scale

- HNSW
  - good recall
  - build & updates expensive
  - memory-intensive for build & query
  - max **2000 vector** dimensions
- IVF
  - Slow for high recall with large volumes

Vector Index Extensions

FurtherVectorindexalternatives for Postgres?

- Relevant innovations:
  - ◯Disk-based algorithms
  - ◯ Quantization
- Open options:
  - pgvectorscale
  - **VectorChord**
- All these can run on existing PGVECTOR columns

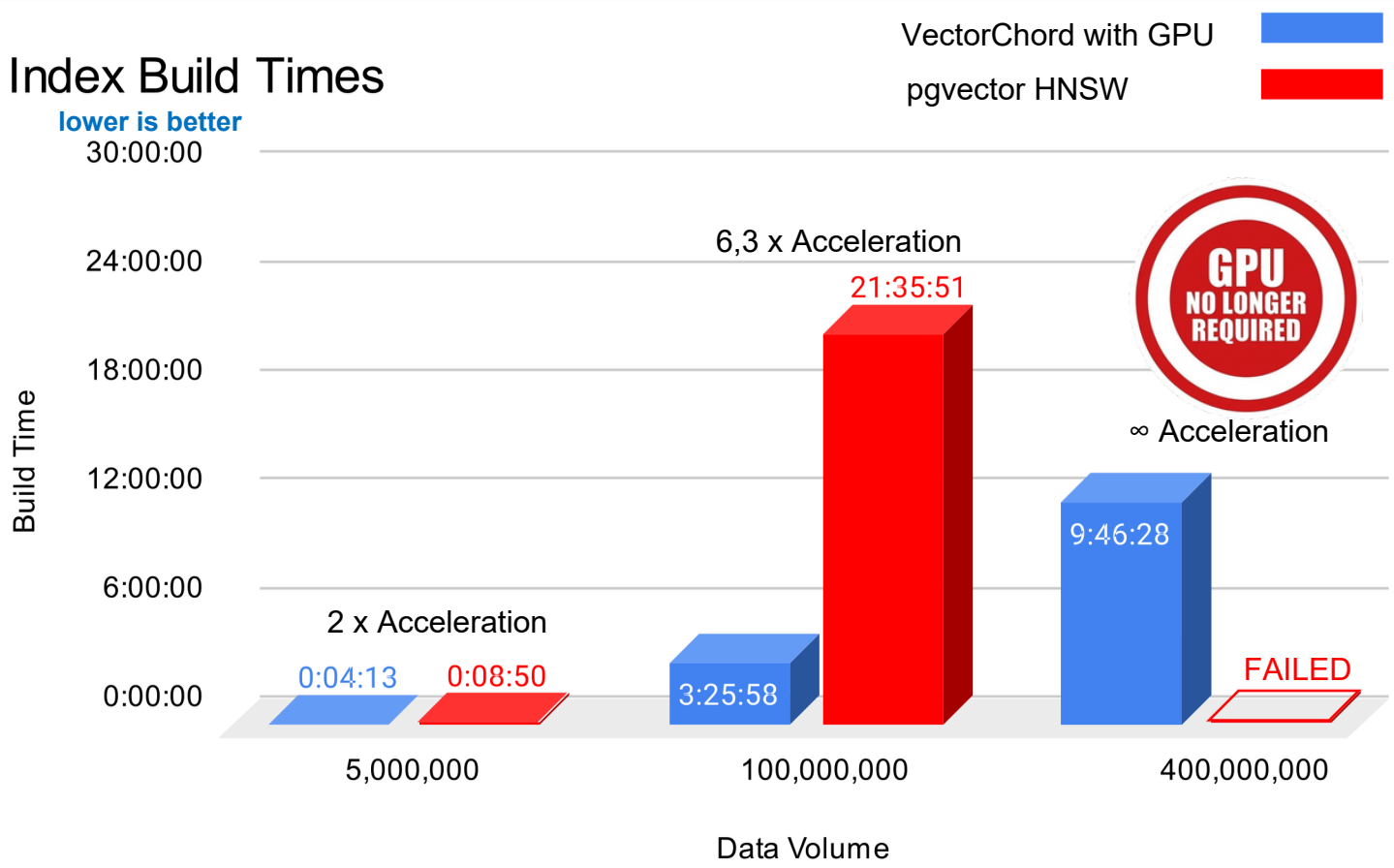
Vectorchord at Scale

- RabitQ
  - Dimension size 60,000+
  - 5x faster queries, 16x higher insert throughput
  - Fully compatible with pgvector
  - No manual parameter tuning needed
  - disk-based algorithms + quantization

pgvector HNSW vs VectorChord

Environment: AWS i7i.16xlarge

- Intel Xeon PLATINUM 8559c
-64 CPU cores
-512GB RAM
-GPU-build on external H100
-PG dotdifference
shared_buffers='128GB'
max_worker_processes = '64'
max_parallel_maintenance_workers = '64'
max_parallel_workers = '64'
maintenance_work_mem = '128GB'

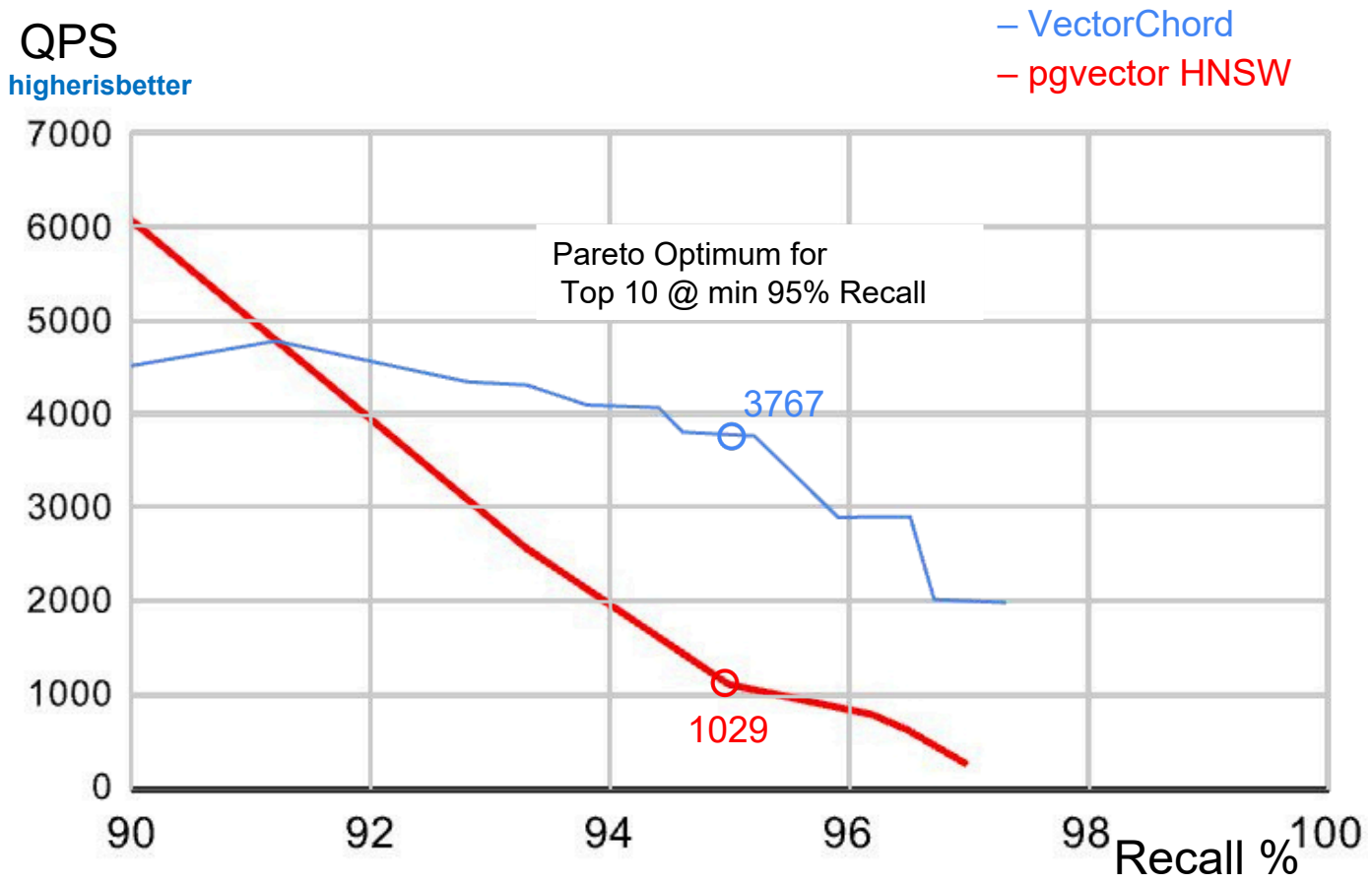


pgvector HNSW vs VectorChord – Parallel Query

Environment: AWS i7i.16xlarge

- Intel Xeon PLATINUM 8559c
-64 CPU cores
-512GB RAM
-GPU-build on external H100
-PG dotdifference
shared_buffers='128GB'
max_worker_processes = '64'

max_parallel_maintenance_workers = '64'
max_parallel_workers = '64'
maintenance_work_mem = '128GB'



VectorChord parameters:

- nprob 20-200
- epsilon 1,1.5, 1.9
- lists 160000
- sampling 256

pgvector parameters:

- m16
- ef_construction 200
- ef_search 10-800

Vectorchord Search Syntax

```
-- Enable thepgvector extension
CREATE EXTENSION IF NOT EXISTS vchord;

-- Create a table for storing items and their embeddings
CREATE TABLE items (
    id serial PRIMARY KEY,
    name text,
    embedding vector(1000) -- A vector with 1000 dimensions
);
```

Creating the extension and a table

```
INSERT INTO items (name, embedding) VALUES
('Apple', '[0.1, 0.2, 0.3]'),
('Banana', '[0.9, 0.8, 0.7]'),
('Orange', '[0.15, 0.25, 0.35]');
```

Inserting embeddings into the table (an AI model is needed to create the embeddings)

```
CREATE INDEX ON items USING vchordrq (embedding vector_l2_ops);
```

Creating a vector index on the embeddings for performance

```
-- Search for the closest match using L2 distance
SELECT name, embedding
FROM items
ORDER BY embedding <-> '[0.12, 0.22, 0.32]'
LIMIT 1;
```

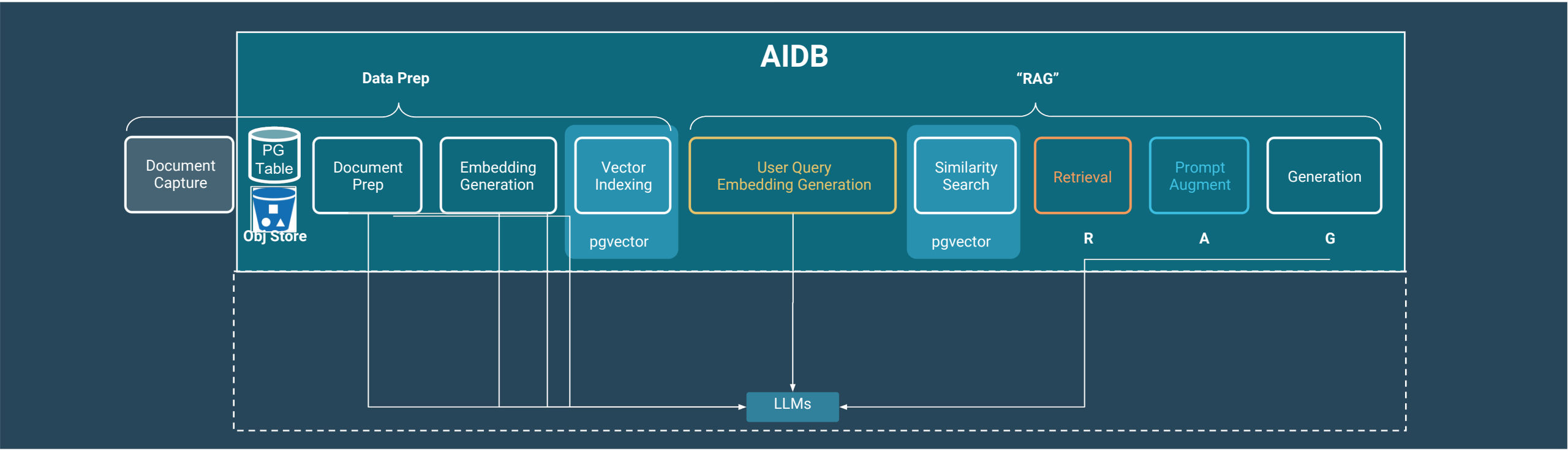
Running a semantic search (an AI model is needed to create the search term embeddings)

Enable GenAI applications with Postgres as **AI Database (AIDB)**

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.



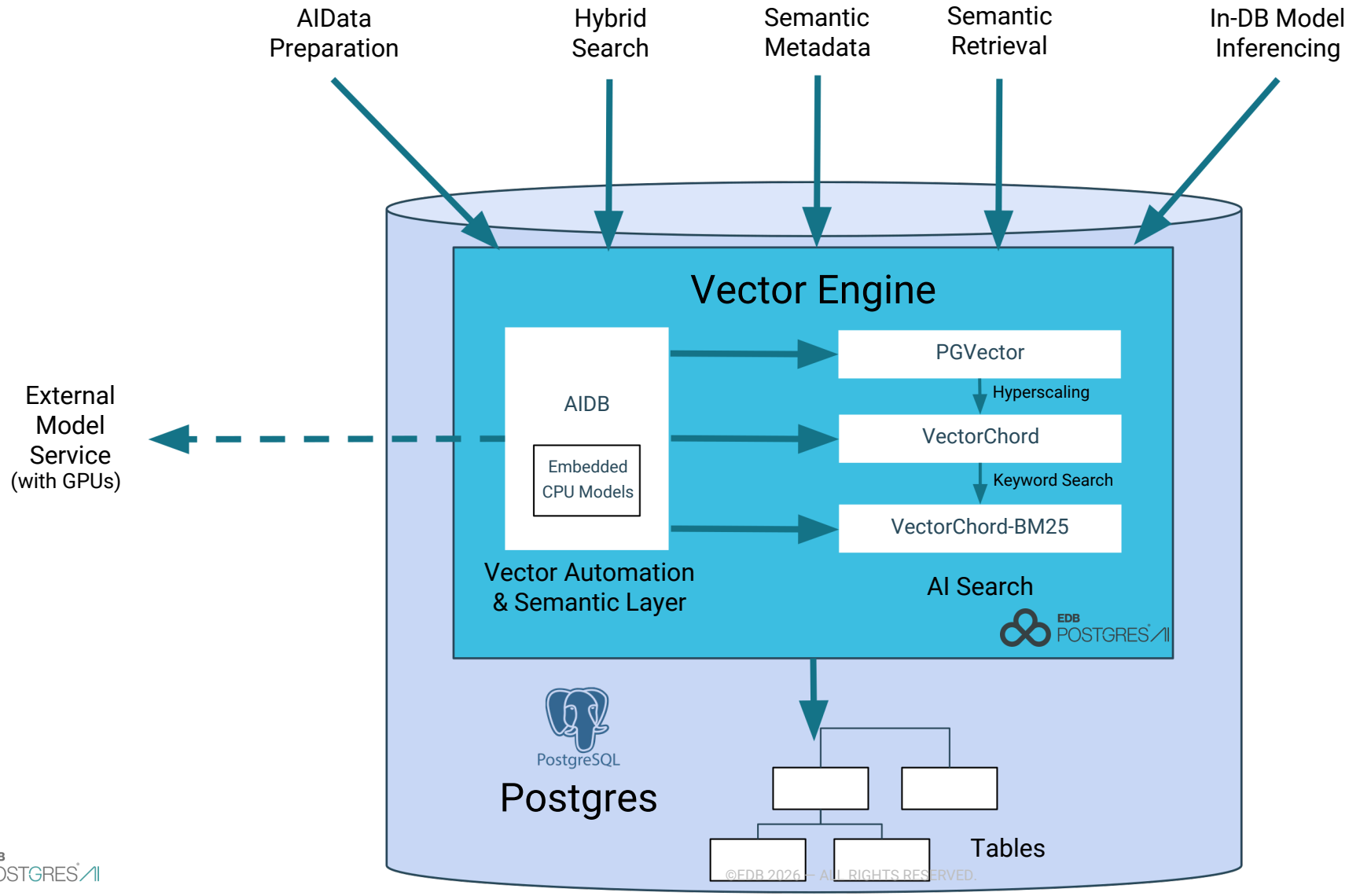
Implementation with aidb

132 linesofstatementswithout,vs **2** with aidb

```
SELECT aidb.create_pipeline(
    name => 'products',
    model_name => 'my_model',
    source_table => 'products',
    source_data_column => 'description',
    source_data_type => 'Text',
    source_key_column => 'id'
    );

SELECT aidb.run_pipeline('products');
```

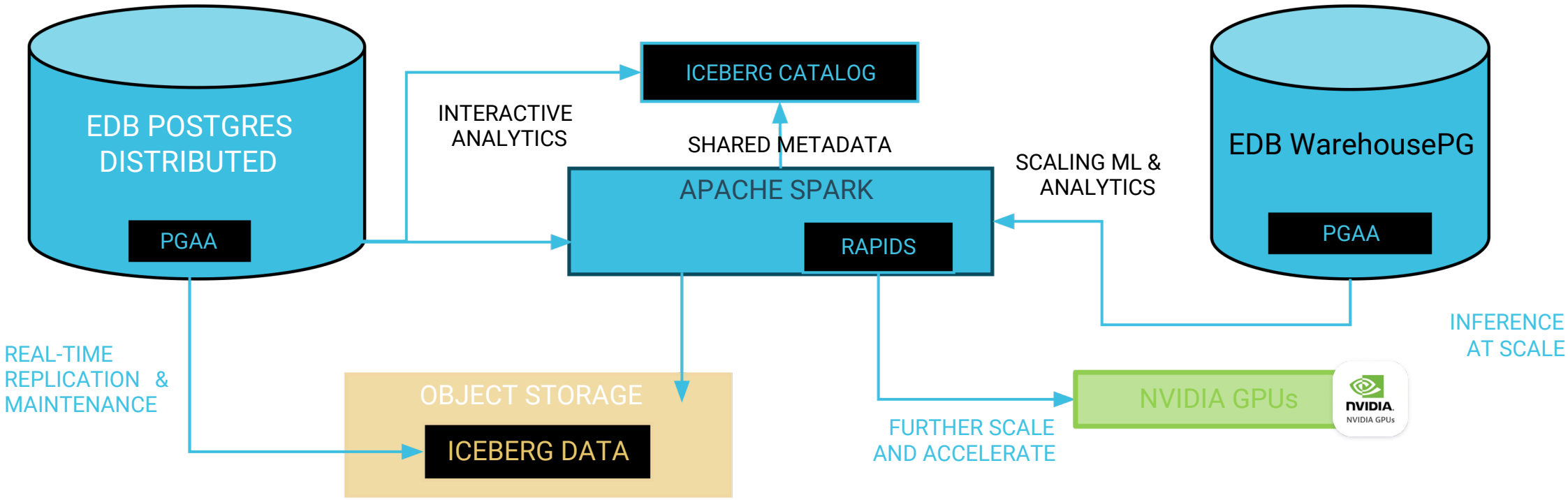

JPMorganChase

EDB Vector Engine - Overview

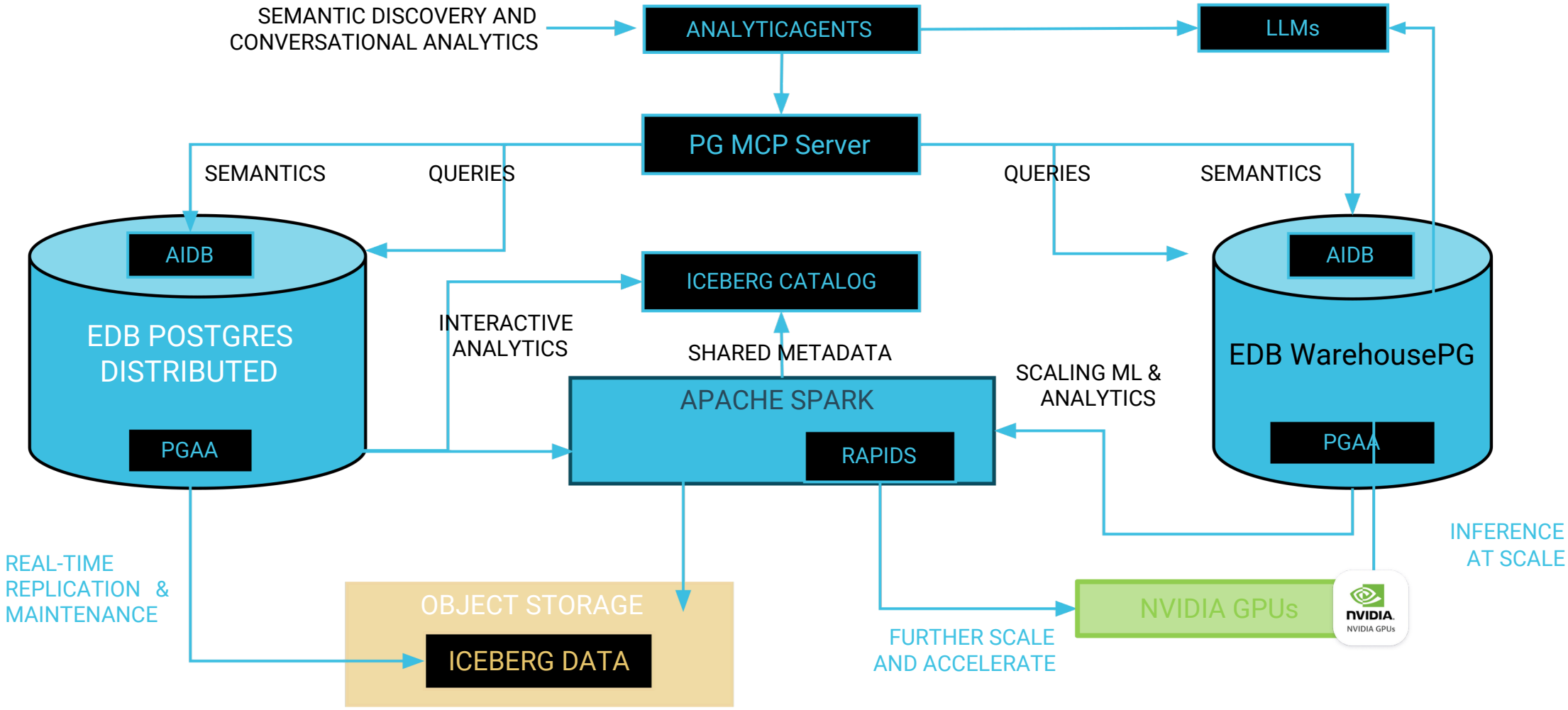


JPMorganChase

Converged Analytics



Agentic Analytics: **Reference Architecture**



OPPORTUNITY: UNIFIED QUERY ENGINE

MOVE FROM DATABASE TO DATA PLATFORM IN POSTGRES
- COMBINE STRUCTURED / UNSTRUCTURED & DIVERSE WORKLOADS

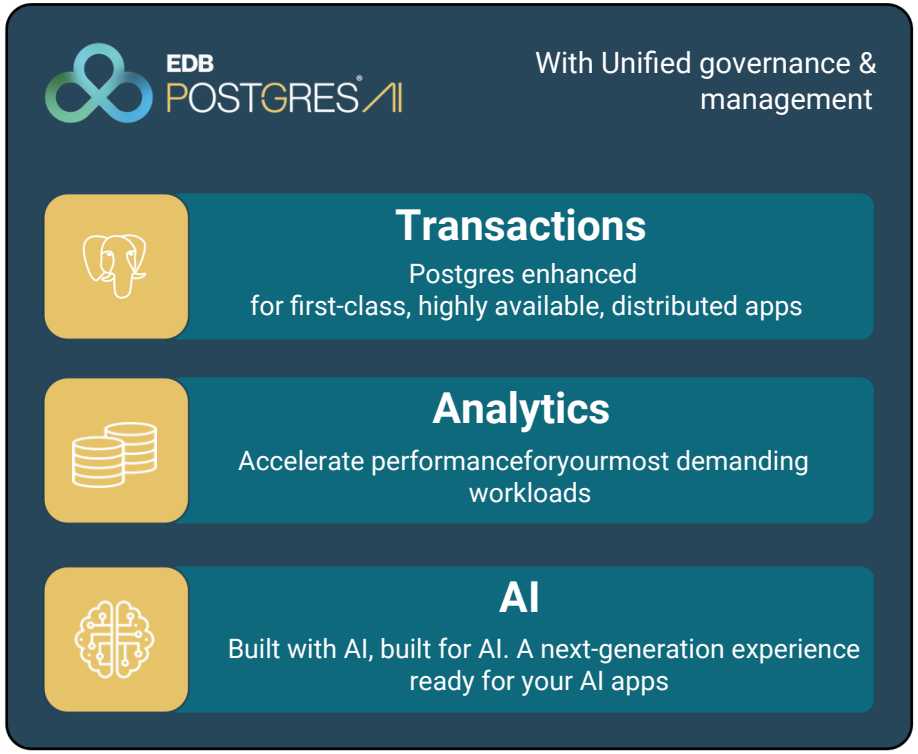
PRIOS
- RESILIENCY
- CONTROL & SECURITY
- ADDRESS COMPLEXITY
- PORTABILITY

Challenges

- Too many query engines
- Governance gaps
- Legacy silos and modern data stack fragmentation

EDB Postgres AI Solution

- **One platform** for querying across all data
- **Familiar Postgres interface**, no new skill sets required
- **Real-time data processing:** improve efficiency, flexibility, scalability, and security



- Building on Postgres' industry renowned transactional workload support
- Postgres is extended for analytics workloads with significantly faster performance than traditional
- Data warehouse quality of service on cheaper storage without the need for expensive ETL.
- Reduced complexity with a one stop Postgres solution for transactions, data warehouse and advanced analytics for AI-infused applications.

Thank You

Please complete the survey for EDB Days Track 3 now.

EDB POSTGRES AI

JPMorganChase

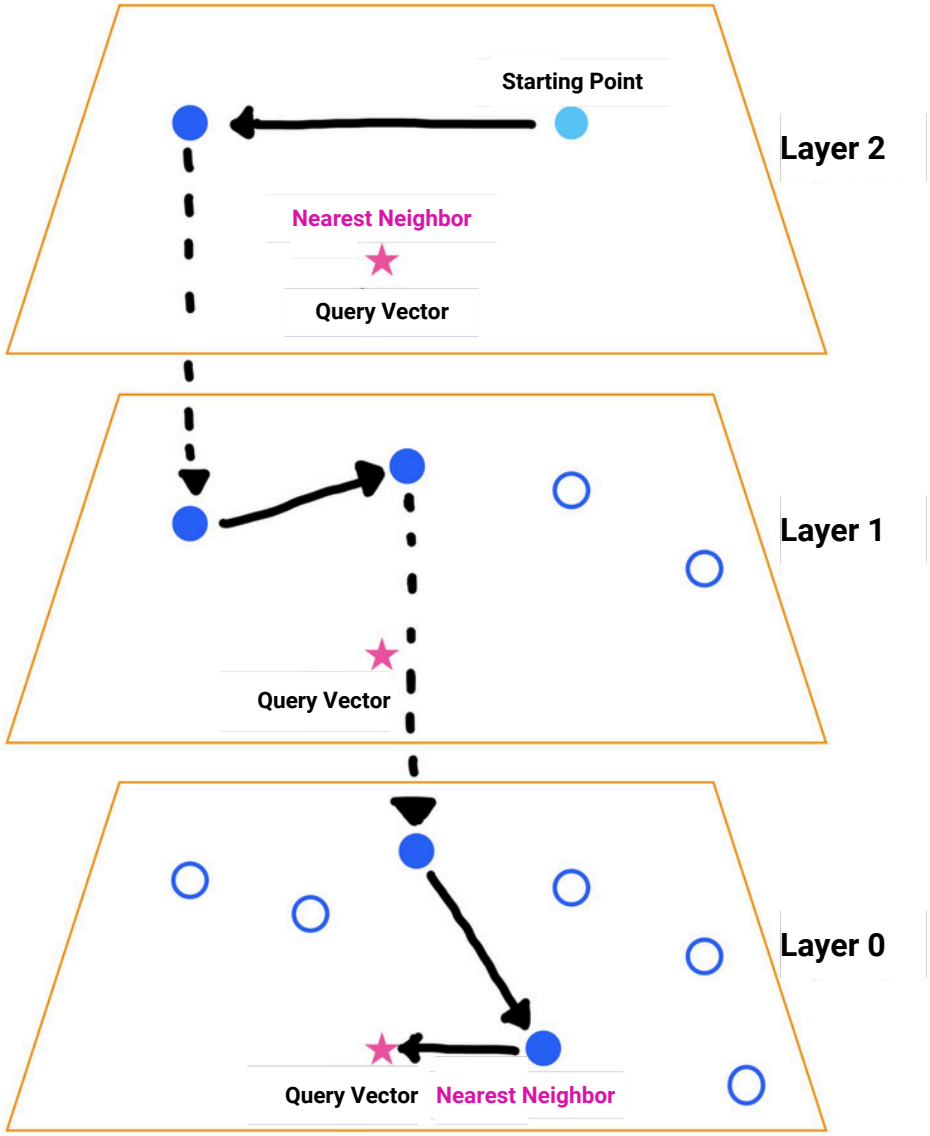
Tune Your Index Creation Parameters

HNSW

Before index build:

m -> max number of connection per layer

- A reasonable range of m: from 5 to 48
- Higher m = better recall = higher index size in memory



Tune Your Index Creation Parameters

HNSW

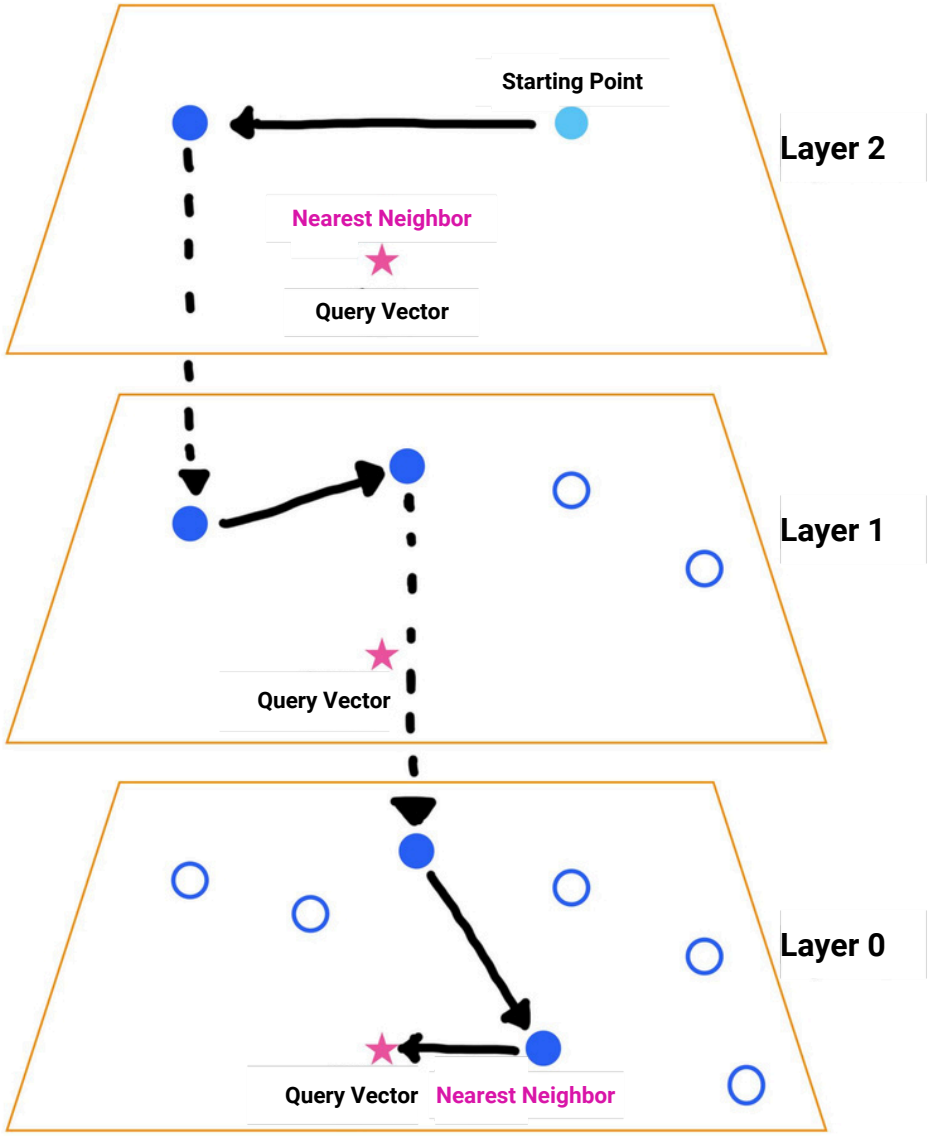
Before index build:

m -> max number of connection per layer

- A reasonable range of m: from 5 to 48
- Higher m = better recall = higher index size in memory

ef_construct -> size of the dynamic candidate list of nodes on each layer:

- `ef_construct => m*2`
- Higher ef_construct leads to little extra performance in exchange for significantly longer construction time



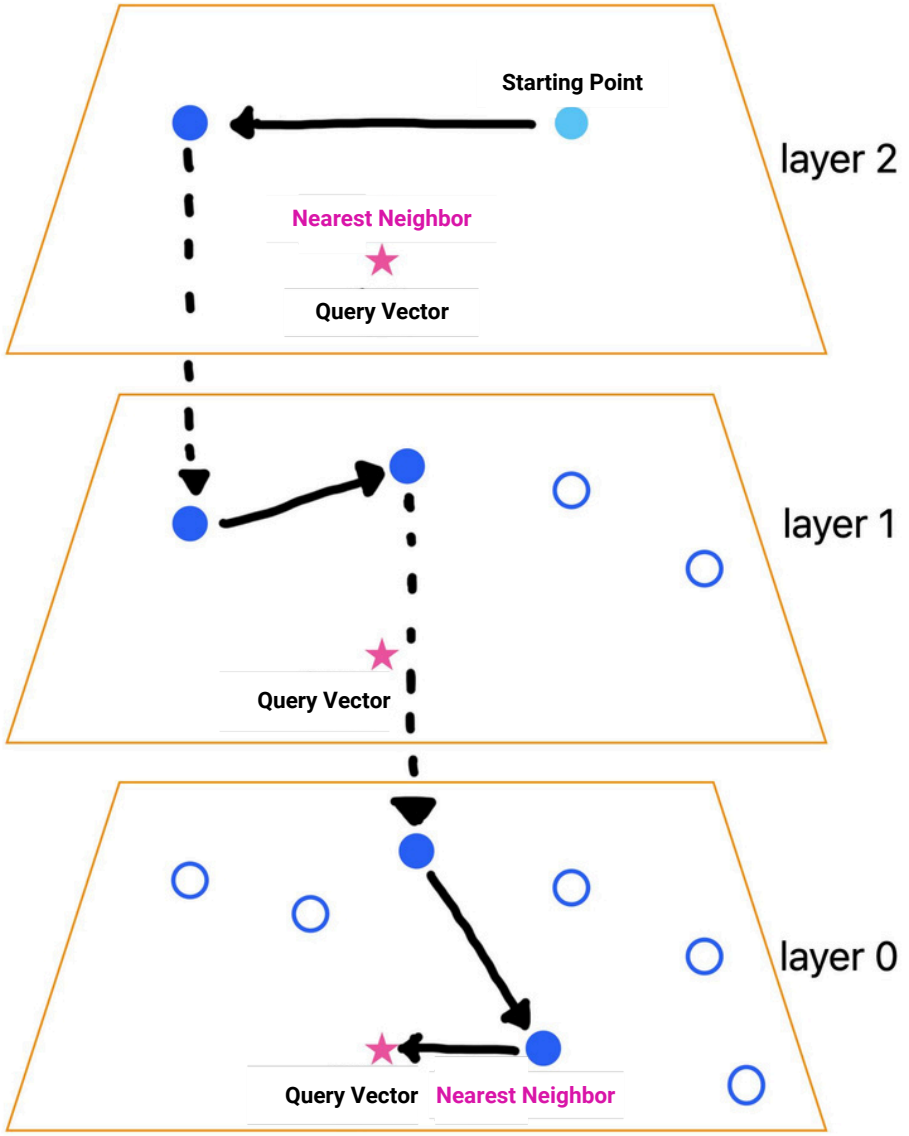
Tune Your Index Creation Parameters

HNSW

Query time:

ef_search -> size of the dynamic list of nodes during search

- Limitation on maximum number of records returned
- Limitation on accuracy



Focus on RAM

- For HNSW, your total RAM needs to be bigger than:
  - The size of your base vectors
  - Plus the size of your HNSW index (which can be 2x–3x the size of your base vectors!)
  - Plus your PostgreSQL shared_buffers (usually 25% of total RAM)

- For IVFFlat, “build time” is the most crucial time for memory needs:
  - Before you create your IVFFlat index, temporarily boost your `maintenance_work_mem` for that session

Prepare Your Vectors Before You Insert

Normalize your vectors when cosine is needed.

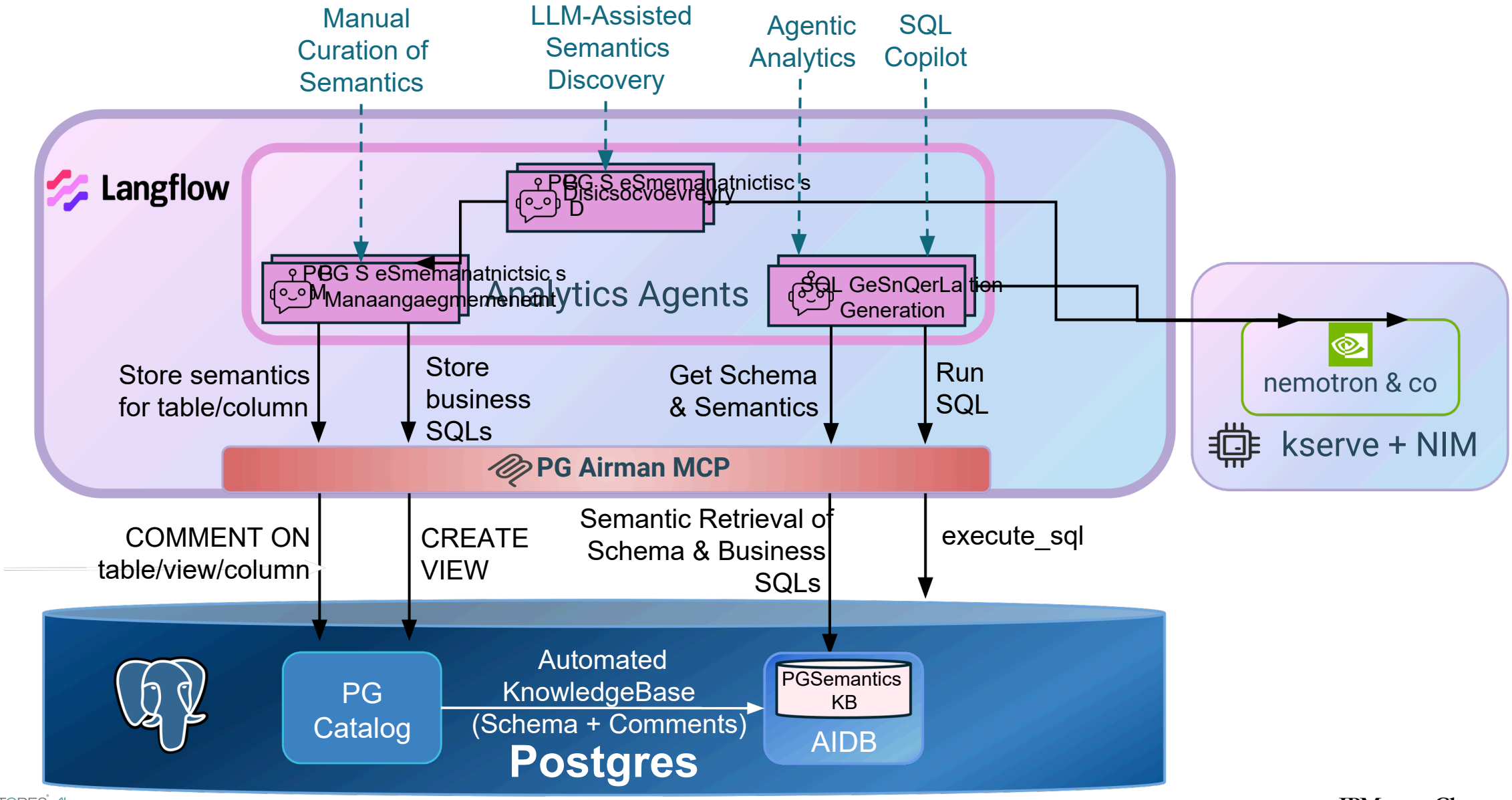
```
-- If your vectors are normalized, this:
SELECT * FROM items ORDER BY embedding <#> '[0.3, 0.4, ...]' LIMIT 5;

-- ...is a faster way to get the same result as this:
SELECT * FROM items ORDER BY embedding <=> '[0.3, 0.4, ...]' LIMIT 5;
```

Consider Quantization

pgvectorsupportsvector(float32) but also halfvec (float16).

```
-- Half the memory, almost all the accuracy!
    CREATE TABLE product_embeddings (
        product_id BIGINT REFERENCES products(id),
        embedding halfvec(384)
    );
```

Agentic Analytics: **Architecture Double Click**

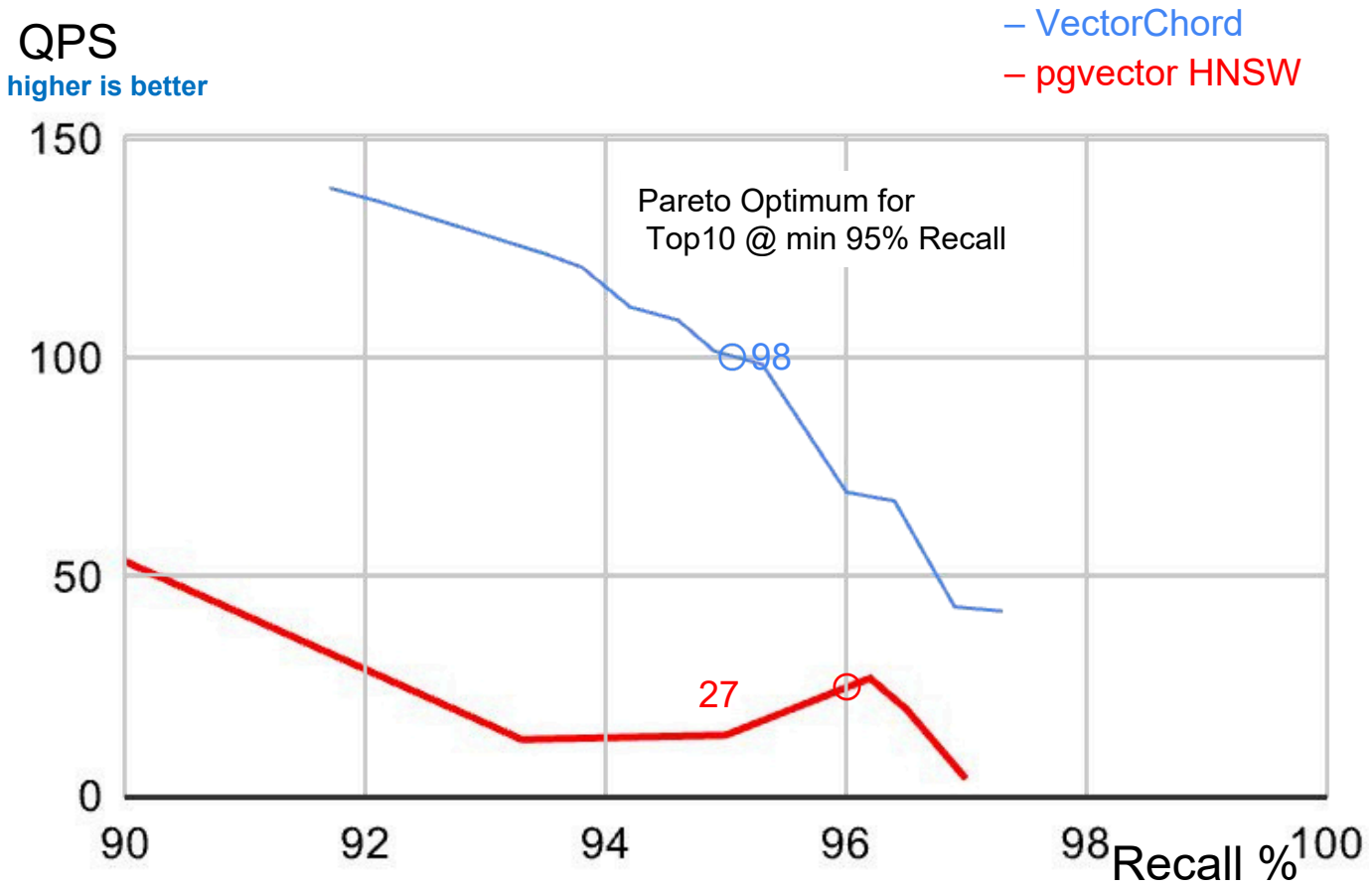


JPMorganChase

VectorChord vs pgvector HNSW – Single Query

Environment: AWS i7i.16xlarge

- Intel Xeon PLATINUM 8559c
- 64 CPU cores
- 512GB RAM
- GPU-build on external H100
- PG dot difference
  shared_buffers = '128GB'
  max_worker_processes = '64'
  max_parallel_maintenance_workers = '64'
  max_parallel_workers = '64'
  maintenance_work_mem = '128GB'



VectorChord parameters:

- nprob 20-200
- epsilon 1,1.5, 1.9
- lists 160000
- sampling 256

pgvector parameters:

- m16
- ef_construction 200
- ef_search 10-800