A Comprehensive Guide to Building Advanced AI Applications with PostgreSQL and pgvector

Executive Summary

This report serves as a definitive, expert-level guide for integrating the pgvector extension into PostgreSQL to build sophisticated, AI-driven enterprise applications. It details a multi-layered architectural pattern that leverages vector embeddings for semantic discovery, entity resolution, and as a core component within an intelligent agent orchestrated by a Large Language Model (LLM). The analysis moves from foundational principles to advanced implementation, demonstrating how the convergence of relational and vector database capabilities within a single system can solve complex business problems.

The core of this report is a practical demonstration of a powerful hybrid architecture. This pattern uses vector search for what it excels at—understanding semantic nuance and resolving ambiguity in natural language—while relying on the proven strengths of a relational database for structured data storage, querying, and maintaining data integrity. The examples and implementations are grounded in a realistic business context, utilizing a detailed Enterprise Architecture (EA) database schema that tracks the intricate relationships between strategic objectives, organizational capabilities, projects, and risks.

The report systematically covers four key areas:

1. **pgvector Fundamentals:** Establishes a production-ready foundation for using pgvector, including installation, data modeling, embedding generation via the OpenAI API, and the implementation of high-performance indexed similarity searches.
2. **Semantic Schema Discovery:** Presents a novel application of semantic search to the database schema itself, enabling users to discover relevant tables using natural language queries, thereby bridging the gap between business terminology and technical implementation.
3. **High-Fidelity Entity Resolution:** Addresses the critical challenge of mapping

ambiguous user references to specific database entities, introducing a "Resolve, then Query" pattern that dramatically improves the reliability of AI-driven database interactions.
4. **LLM Orchestration with Function Calling:** Culminates in a capstone example that integrates all preceding concepts with OpenAI's function calling feature. This section demonstrates the construction of an intelligent agent capable of orchestrating vector searches and subsequent SQL queries to answer complex, multi-step business questions.

By combining the structured power of PostgreSQL with the semantic search capabilities of pgvector, organizations can unlock new paradigms for data interaction, moving beyond rigid dashboards and keyword searches to create truly conversational and intelligent applications. This report provides the architectural blueprints and production-grade code necessary to realize this transformative potential.

Part I: pgvector Fundamentals: From Installation to Indexed Queries

This foundational section provides a complete, hands-on guide to getting started with pgvector. It moves beyond trivial examples to establish a production-ready baseline, incorporating best practices from the outset to ensure scalability and performance.

1.1. Introduction: The RDBMS and Vector Database Convergence

The landscape of data management is undergoing a significant transformation, marked by the convergence of traditional Relational Database Management Systems (RDBMS) and specialized vector databases. Historically, workloads involving semantic search or similarity matching on high-dimensional vector data required a separate, dedicated vector database. This approach, while powerful, introduced architectural complexity, data synchronization challenges, and operational overhead. A prominent industry trend is now challenging this paradigm by augmenting established relational databases with native vector search capabilities.[1]

At the forefront of this movement is pgvector, a mature, open-source PostgreSQL extension

that integrates a powerful vector similarity search engine directly into the database kernel. This allows developers to store, index, and query high-dimensional vectors alongside their traditional structured data within a single, robust, ACID-compliant system.[3] The benefits of this unified approach are substantial:

- **Reduced Architectural Complexity:** Eliminates the need for a separate vector database, simplifying deployment, management, and data pipelines.
- **Unified Data Management:** Stores embeddings directly with their source data, ensuring consistency and simplifying data governance.
- **Powerful Hybrid Queries:** Enables queries that seamlessly combine traditional structured filtering (e.g., using WHERE clauses on timestamps, categories, or user IDs) with unstructured semantic search, a capability that is often difficult to achieve efficiently across separate systems.

By leveraging pgvector, organizations can build sophisticated AI applications on their existing PostgreSQL infrastructure, benefiting from decades of development in database reliability, security, and ecosystem tooling.

1.2. Environment Setup and Installation

A robust development environment is the first step toward building production-grade applications. This section details the setup of PostgreSQL with pgvector using Docker for reproducibility and the configuration of a dedicated Python environment.

PostgreSQL and pgvector Setup via Docker

Docker provides an isolated and consistent environment for running database services.

1. **Create a docker-compose.yml file:** This file defines the PostgreSQL service and specifies a Docker image that includes the pgvector extension.
   YAML
   ```
   version: '3.8'
   services:
     postgres:
       image: pgvector/pgvector:pg16
       container_name: postgres_with_pgvector
   ```

```
    environment:
      POSTGRES_DB: ai_database
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: yoursecurepassword
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

2. **Start the Container:** From the directory containing the docker-compose.yml file, run the following command:
   Bash
   ```
   docker-compose up -d
   ```

3. **Enable the Extension:** Connect to the running PostgreSQL instance and execute the CREATE EXTENSION command. This only needs to be done once per database.[6]
   Bash
   ```
   psql "postgresql://admin:yoursecurepassword@localhost:5432/ai_database" -c "CREATE EXTENSION IF NOT EXISTS vector;"
   ```

   This command makes the vector data type and its associated functions and operators available for use within the ai_database.

Python Environment Setup

A virtual environment is crucial for managing project dependencies and avoiding conflicts with system-wide packages.[7]

1. **Create and Activate a Virtual Environment:**
   Bash
   ```
   python3 -m venv venv
   source venv/bin/activate
   ```

2. **Install Necessary Libraries:** Install the required Python packages for database interaction, vector operations, and interacting with the OpenAI API.
   Bash

```
pip install psycopg2-binary openai pgvector SQLAlchemy python-dotenv
```

- psycopg2-binary: A robust PostgreSQL adapter for Python.
- openai: The official Python client for the OpenAI API.
- pgvector: Provides adapters for psycopg2 and SQLAlchemy to handle the vector data type seamlessly.
- SQLAlchemy: An optional but powerful Object-Relational Mapper (ORM) that can simplify database interactions.
- python-dotenv: For managing environment variables, such as API keys, securely.

1.3. Core Operations: Storing and Querying Embeddings

With the environment configured, the next step is to perform the fundamental operations of creating a vector-enabled table, generating embeddings, and executing similarity searches.

Creating a Vector-Enabled Table

The pgvector extension introduces the vector(n) data type, where n is the number of dimensions in the vector. It is a critical best practice to ensure this dimension matches the output of the embedding model you intend to use. For OpenAI's widely used text-embedding-ada-002 model, the dimension is **1536**.[8] Using an incorrect dimension will result in errors or data truncation.

SQL Example:

```
SQL
```

```sql
CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    content TEXT NOT NULL,
    embedding VECTOR(1536)
);
```

This SQL statement creates a table named documents with a standard primary key, a TEXT column to store the original content, and an embedding column of type VECTOR(1536).

Generating Embeddings with the OpenAI API

An embedding is a numerical vector representation of text that captures its semantic meaning. The following Python function encapsulates the process of generating an embedding using the OpenAI API. For security, the API key should be stored in an environment variable and loaded at runtime, never hard-coded in the source code.[10]

Complete Python Code:

Python

```
import os
from openai import OpenAI
from dotenv import load_dotenv

# Load environment variables from a.env file
load_dotenv()

# Initialize the OpenAI client
# The API key is read automatically from the OPENAI_API_KEY environment variable
try:
    client = OpenAI()
except Exception as e:
    print(f"Error initializing OpenAI client: {e}")
    # Handle the case where the API key is not set
    client = None

def get_openai_embedding(text: str, model: str = "text-embedding-ada-002") -> list[float] | None:
    """
    Generates an embedding for a given text using a specified OpenAI model.

    Args:
```

```
        text (str): The input text to embed.
        model (str): The embedding model to use.

    Returns:
        list[float] | None: The embedding vector as a list of floats, or None if an error occurs.
    """
    if not client:
        print("OpenAI client is not initialized. Please check your API key.")
        return None

    try:
        # Replace newlines with spaces, as recommended by OpenAI for embedding models
        text = text.replace("\n", " ")
        response = client.embeddings.create(input=[text], model=model)
        return response.data.embedding
    except Exception as e:
        print(f"An error occurred while generating embedding: {e}")
        return None

# Example usage:
# sample_text = "This is a test document for pgvector."
# embedding_vector = get_openai_embedding(sample_text)
# if embedding_vector:
#     print(f"Generated a {len(embedding_vector)}-dimensional vector.")
```

Inserting and Querying Vector Data

The following complete Python script demonstrates the end-to-end process: connecting to the database, inserting documents with their embeddings, and performing a similarity search to find the most relevant documents for a given query.

Complete Python Script (pgvector_basics.py):

```
Python

import psycopg2
```

```
from pgvector.psycopg2 import register_vector
import numpy as np

# --- Helper Functions (assuming get_openai_embedding is in this file or imported) ---

def get_db_connection():
    """Establishes a connection to the PostgreSQL database."""
    try:
        conn = psycopg2.connect(
            dbname="ai_database",
            user="admin",
            password="yoursecurepassword",
            host="localhost",
            port="5432"
        )
        return conn
    except psycopg2.OperationalError as e:
        print(f"Could not connect to the database: {e}")
        return None

def setup_database(conn):
    """Sets up the database by creating the table and enabling the extension."""
    with conn.cursor() as cur:
        cur.execute("CREATE EXTENSION IF NOT EXISTS vector;")
        cur.execute("""
        CREATE TABLE IF NOT EXISTS documents (
            id SERIAL PRIMARY KEY,
            content TEXT NOT NULL,
            embedding VECTOR(1536)
        );
        """)
        # Clear existing data for a clean run
        cur.execute("TRUNCATE documents RESTART IDENTITY;")
        conn.commit()
    print("Database setup complete.")

def insert_documents(conn, docs):
    """Generates embeddings and inserts documents into the database."""
    with conn.cursor() as cur:
        for doc_content in docs:
            embedding = get_openai_embedding(doc_content)
```

```
            if embedding:
                cur.execute(
                    "INSERT INTO documents (content, embedding) VALUES (%s, %s)",
                    (doc_content, np.array(embedding))
                )
    conn.commit()
    print(f"Inserted {len(docs)} documents.")

def similarity_search(conn, query_text, k=3):
    """Performs a similarity search for a given query."""
    query_embedding = get_openai_embedding(query_text)
    if not query_embedding:
        print("Could not generate query embedding.")
        return

    with conn.cursor() as cur:
        # Register the vector type with the connection
        register_vector(conn)

        print("\n--- Performing L2 (Euclidean) Distance Search ---")
        cur.execute(
            "SELECT id, content, embedding <-> %s AS distance FROM documents ORDER BY distance
LIMIT %s",
            (np.array(query_embedding), k)
        )
        results_l2 = cur.fetchall()
        for row in results_l2:
            print(f"ID: {row}, Distance: {row:.4f}, Content: '{row}'")

        print("\n--- Performing Cosine Distance Search ---")
        cur.execute(
            "SELECT id, content, 1 - (embedding <=> %s) AS similarity FROM documents ORDER BY
similarity DESC LIMIT %s",
            (np.array(query_embedding), k)
        )
        results_cosine = cur.fetchall()
        for row in results_cosine:
            print(f"ID: {row}, Similarity: {row:.4f}, Content: '{row}'")


if __name__ == "__main__":
```

```
    connection = get_db_connection()
    if connection:
        # 1. Setup the database
        setup_database(connection)

        # 2. Define documents to insert
        documents_to_insert =

        # 3. Insert documents
        insert_documents(connection, documents_to_insert)

        # 4. Perform similarity search
        user_query = "a pet on a carpet"
        print(f"\nSearching for documents similar to: '{user_query}'")
        similarity_search(connection, user_query, k=3)

        # Close the connection
        connection.close()
```

1.4. Understanding Distance Metrics: L2 vs. Cosine Similarity

pgvector provides several operators to measure the "distance" or "similarity" between vectors. The two most common for semantic search are L2 (Euclidean) distance and Cosine distance.[12]

- **L2 Distance (<->)**: This is the straight-line or "as the crow flies" distance between the endpoints of two vectors in the high-dimensional space. It considers both the direction and the magnitude of the vectors. A smaller L2 distance means the vectors are closer and thus more similar.[3]
- **Cosine Distance (<=>)**: This metric measures the cosine of the angle between two vectors. It is concerned only with the orientation (direction) of the vectors, not their magnitude. The pgvector operator returns the *distance*, which is calculated as $1 - \text{cosine similarity}$. A smaller cosine distance (closer to 0) indicates a smaller angle and therefore higher similarity.[3]

A crucial aspect of working with text embeddings from modern language models is that the output vectors are typically **normalized**. Normalization scales a vector so that its length (magnitude) is 1. This process effectively places all vector endpoints on the surface of a unit

hypersphere, removing magnitude as a distinguishing factor and focusing purely on the semantic "direction."

When all vectors are normalized, a direct mathematical relationship emerges between L2 distance and cosine similarity. Minimizing the straight-line distance between two points on the surface of a sphere is equivalent to minimizing the angle between the vectors connecting those points to the origin. Because of this, for normalized vectors, a query ordered by L2 distance will yield the exact same ranking as a query ordered by cosine distance.[13]

Given this equivalence, **Cosine Distance (<=>)** is often the recommended metric for semantic search applications. Its conceptual model—measuring the angle between vectors of meaning—aligns more intuitively with the goal of finding text with similar context and intent, abstracting away the less relevant concept of vector magnitude.

1.5. Optimizing for Performance: An Introduction to Indexing

As the number of vectors in a table grows, performing a similarity search via a sequential scan becomes computationally prohibitive. A sequential scan must calculate the distance between the query vector and every single vector in the table, an operation with a time complexity of $O(N)$, where N is the number of rows. For millions of vectors, this can take seconds or even minutes.

To achieve sub-second query times on large datasets, it is essential to use an **Approximate Nearest Neighbor (ANN) index**. ANN indexes trade a small amount of recall accuracy for a massive gain in search speed by intelligently partitioning the vector space, allowing the search algorithm to explore only the most promising regions instead of the entire dataset.[15] pgvector supports two primary types of ANN indexes: IVFFlat and HNSW.

- **IVFFlat (Inverted File with Flat Quantization):** This index works by first partitioning the vector space into a predefined number of clusters, or lists, using a clustering algorithm like k-means. Each vector is then assigned to its nearest cluster centroid. At query time, the search is narrowed to only the cluster(s) closest to the query vector, dramatically reducing the number of distance calculations needed. The number of clusters to search is controlled by the probes parameter.[17] IVFFlat is generally faster to build and has a smaller memory footprint than HNSW, making it a solid choice for static or infrequently updated datasets. A good starting point for the lists parameter is rows / 1000 for up to 1 million rows, or sqrt(rows) for larger datasets.[17]
- **HNSW (Hierarchical Navigable Small World):** This index builds a multi-layered graph structure where nodes are vectors and edges connect a node to its nearest neighbors.

> The hierarchy allows for efficient searching, starting from a sparse top layer and navigating down to denser layers to quickly converge on the nearest neighbors. HNSW typically offers superior query performance and higher recall compared to IVFFlat, especially on dynamic datasets where data is frequently added or updated. It comes at the cost of longer build times and higher memory consumption, but is often the recommended choice for high-performance applications.[16]

To create an index, it is vital to use the operator class that corresponds to the distance metric you will use in your queries. Using a mismatched operator will prevent the query planner from utilizing the index.[14]

SQL Examples for Index Creation:

SQL

```
-- Create an HNSW index for use with Cosine Distance (<=>)
CREATE INDEX ON documents USING hnsw (embedding vector_cosine_ops);

-- Create an HNSW index for use with L2 Distance (<->)
CREATE INDEX ON documents USING hnsw (embedding vector_l2_ops);

-- Create an IVFFlat index with 100 lists for Cosine Distance
-- This should be created AFTER the table has been populated with data
CREATE INDEX ON documents USING ivfflat (embedding vector_cosine_ops) WITH (lists = 100);
```

For IVFFlat, the index should be created after the table contains a representative sample of data, as the clustering algorithm needs data to effectively partition the vector space. HNSW indexes do not have this requirement and can be created on an empty table.

Part II: Semantic Schema Discovery in an Enterprise Architecture Context

This section transitions from fundamental operations to a high-value, practical application: using semantic search to enable users to discover database tables through natural language

queries. This technique is particularly powerful in complex enterprise environments where the technical schema can be opaque to business users.

2.1. The Challenge: Bridging Business Language and Technical Schema

A common friction point in data-driven organizations is the gap between business concepts and their technical implementation in a database. A business analyst might need to find information about "strategic goals for the upcoming year" or "risks related to our IT infrastructure," but they are unlikely to know that this information resides in tables named sec_objectives and ent_risks, respectively. Traditional database discovery tools rely on exact name matching or manual documentation, which is often incomplete or outdated.

The solution is to create a semantic search layer over the database schema itself. By generating vector embeddings for the schema's metadata—including table names, column names, and, most importantly, rich business descriptions—we can create a searchable catalog. This allows a user's natural language query to be matched against the *meaning* and *context* of each table, rather than just its name.[18]

2.2. Curating a High-Quality Metadata Corpus

The effectiveness of a semantic search engine is directly proportional to the quality of the text corpus it searches over. Simply concatenating table and column names provides a weak semantic signal. The key to building a high-fidelity schema search engine is to curate a rich, descriptive document for each table that synthesizes its technical metadata with its intended business context.

The provided documentation for the Enterprise Architecture database offers a perfect opportunity to construct such a corpus. The DB as JSON.txt file contains the technical schema (table and column names) [21], while the LevelsDefinitions.csv file provides detailed, multi-level business descriptions for each major entity.[21] By combining these two sources, we can create a comprehensive description for each table that bridges the linguistic gap between user intent and technical implementation.

For example, to create the descriptive document for the ent_projects table, the process is as follows:

1. Identify the table name (ent_projects) and its columns (id, year, name, level, etc.) from the technical schema document.[21]
2. Look up the corresponding "Projects" business module in the business context document.[21]
3. Extract the descriptions for L1 ("Collections of Initiatives - Called Programs or Portfolios"), L2 ("Initiatives that deliver specific outcomes"), and L3 ("Primary output delivered from a project").
4. Synthesize this information into a single, coherent text document that will be used for embedding.

The resulting document for the ent_projects table would be structured like this:

> "Table Name: ent_projects. This table stores information about Projects. At Level 1, this represents Collections of Initiatives, also known as Programs or Portfolios. At Level 2, this represents Initiatives that deliver specific outcomes, which are usually composed of many projects. At Level 3, this represents the Primary output delivered from a project which is deployed in operations. The columns in this table include: id, year, quarter, name, level, parent_id, parent_year, start_date, end_date, status, budget, and progress_percentage."

This composite document creates an embedding that is semantically rich, capturing both the technical structure and the business purpose of the table, making it far more likely to match a user's natural language query.

2.3. Implementation: Building the Schema Search Engine

The implementation involves three steps: setting up a dedicated table to store the schema embeddings, writing a script to populate this table, and creating a query function to perform the search.

Database Setup

A new table is required to store the table names, their curated descriptions, and the corresponding embeddings.

SQL Schema:

SQL

```
CREATE TABLE schema_embeddings (
    table_name TEXT PRIMARY KEY,
    description TEXT,
    embedding VECTOR(1536)
);

-- Create an HNSW index for fast similarity search
CREATE INDEX ON schema_embeddings USING hnsw (embedding vector_cosine_ops);
```

Embedding and Storage Script

The following Python script automates the process of curating the metadata corpus, generating embeddings, and populating the schema_embeddings table.

Complete Python Script (embed_schema.py):

Python

```
import json
import csv
import psycopg2
from pgvector.psycopg2 import register_vector
import numpy as np
# Assume get_openai_embedding and get_db_connection are available from the previous section

def parse_schema_metadata(schema_file):
    """Parses the detailed database schema from the JSON file."""
    with open(schema_file, 'r') as f:
        data = json.load(f)
```

```
    schema_info = {}
    # The actual schema data is nested inside the JSON structure
    for item in data['execute_sql']:
        table_name = item['table_name']
        if table_name not in schema_info:
            schema_info[table_name] = {'columns': set()}
        schema_info[table_name]['columns'].add(item['column_name'])

    # Convert sets to sorted lists for consistent output
    for table in schema_info:
        schema_info[table]['columns'] = sorted(list(schema_info[table]['columns']))

    return schema_info

def parse_business_context(context_file):
    """Parses the business context from the LevelsDefinitions.csv file."""
    context = {}
    with open(context_file, 'r', encoding='utf-8') as f:
        reader = csv.reader(f)
        next(reader)  # Skip header
        current_module = ""
        for row in reader:
            module, level, desc = row
            if module:
                current_module = module.strip()
                if current_module not in context:
                    context[current_module] = {}
            if level and desc:
                context[current_module][level.strip()] = desc.strip()
    return context

def map_module_to_table(module_name):
    """Maps business module names to database table names."""
    # This mapping is based on an analysis of the schema and business context.
    # It may need to be adjusted based on deeper domain knowledge.
    mapping = {
        "Objectives": "sec_objectives",
        "Policy Execution Tools": "sec_policy_tools",
        "Citizens": "sec_citizens",
        "Businesses": "sec_businesses",
        "Government Stakeholders": "sec_gov_entities",
```

```
        "Admin Records": "sec_admin_records",
        "Process Transactions": "sec_data_transactions",
        "Capability Core": "ent_capabilities",
        "Risks": "ent_risks",
        "Performance": "sec_performance",
        "Organization": "ent_org_units",
        "Process": "ent_processes",
        "IT Systems": "ent_it_systems",
        "Culture Health": "ent_culture_health",
        "Change Architecture": "ent_change_adoption",
        "Projects": "ent_projects",
        "Vendors": "ent_vendors"
    }
    return mapping.get(module_name)

def generate_and_store_schema_embeddings(conn, schema_info, business_context):
    """Generates and stores embeddings for the database schema."""
    with conn.cursor() as cur:
        register_vector(cur)
        cur.execute("CREATE EXTENSION IF NOT EXISTS vector;")
        cur.execute("""
        CREATE TABLE IF NOT EXISTS schema_embeddings (
            table_name TEXT PRIMARY KEY,
            description TEXT,
            embedding VECTOR(1536)
        );
        """)
        cur.execute("CREATE INDEX IF NOT EXISTS schema_hnsw_idx ON schema_embeddings USING
hnsw (embedding vector_cosine_ops);")
        cur.execute("TRUNCATE schema_embeddings;")

        for module, levels in business_context.items():
            table_name = map_module_to_table(module)
            if not table_name or table_name not in schema_info:
                continue

            # Construct the rich description
            description = f"Table Name: {table_name}. This table stores information about {module}. "
            for level, desc in levels.items():
                description += f"At {level}, this represents {desc}. "
```

```
            columns = ", ".join(schema_info[table_name]['columns'])
            description += f"The columns in this table include: {columns}."

            # Generate embedding
            embedding = get_openai_embedding(description)
            if embedding:
                print(f"Embedding table: {table_name}")
                cur.execute(
                    "INSERT INTO schema_embeddings (table_name, description, embedding) VALUES (%s,
%s, %s)",
                    (table_name, description, np.array(embedding))
                )
        conn.commit()

def find_relevant_tables(conn, user_query, top_k=3):
    """Finds the most relevant tables for a natural language user query."""
    query_embedding = get_openai_embedding(user_query)
    if not query_embedding:
        print("Failed to generate embedding for query.")
        return

    with conn.cursor() as cur:
        register_vector(cur)
        cur.execute(
            "SELECT table_name, 1 - (embedding <=> %s) AS similarity FROM schema_embeddings ORDER
BY similarity DESC LIMIT %s",
            (np.array(query_embedding), top_k)
        )
        results = cur.fetchall()
        return results

if __name__ == "__main__":
    # File paths to the provided schema and context documents
    schema_file_path = 'DB as JSON.txt'
    context_file_path = 'LevelsDefinitions.csv'

    # 1. Parse metadata
    db_schema = parse_schema_metadata(schema_file_path)
    biz_context = parse_business_context(context_file_path)

    # 2. Connect to DB and populate embeddings
```

```
    connection = get_db_connection()
    if connection:
        generate_and_store_schema_embeddings(connection, db_schema, biz_context)
        print("\nSchema embeddings have been generated and stored.")

        # 3. Test the search function
        test_query = "Find me information about projects planned for 2027"
        print(f"\nSearching for tables related to: '{test_query}'")
        relevant_tables = find_relevant_tables(connection, test_query, top_k=3)

        if relevant_tables:
            print("\nTop matching tables:")
            for table, similarity in relevant_tables:
                print(f"- {table} (Similarity: {similarity:.4f})")

        connection.close()
```

This implementation directly provides the functionality requested by the user: it takes a natural language query, generates an embedding, performs a vector search against the curated schema descriptions, and returns the top-K most relevant table names. This semantic discovery layer empowers users to interact with the database on their own terms, significantly lowering the barrier to data access.

Part III: High-Fidelity Entity Resolution with Vector Embeddings

While semantic schema discovery helps users find the right table, a more granular and common challenge is mapping a user's ambiguous reference to a specific entity—a single row—within that table. This process, known as entity resolution, is critical for building applications that can translate natural language commands into precise, executable database operations.

3.1. The Problem: From Fuzzy Mentions to Concrete IDs

Consider a user interacting with an AI assistant for the Enterprise Architecture database. The user might ask, "What is the budget for our digital transformation initiatives?" The LLM may correctly identify that this query relates to the ent_projects table, but it still faces a significant hurdle. The phrase "digital transformation initiatives" is a fuzzy, human-readable description. To query the database accurately, the system needs the concrete, non-ambiguous primary key for that project, which, according to the schema, is a composite of id and year (e.g., ('proj-dt-001', 2024)).

Attempting to solve this with traditional SQL LIKE clauses (WHERE name LIKE '%digital transformation%') is brittle and prone to error. It fails to account for synonyms, alternate phrasings ("initiatives for DT"), or related but distinct projects. Vector embeddings provide a robust solution by enabling a search based on semantic meaning rather than lexical similarity.[4]

This creates an architectural pattern that can be termed "Resolve, then Query." The first step is not to generate a complex SQL query, but to resolve the ambiguous entity mention into a concrete identifier. This separation of concerns dramatically improves the reliability of the overall system. The fuzzy semantic matching is delegated to vector search, a tool perfectly suited for the task. Once a precise identifier is obtained, the system can proceed to construct a much simpler and more reliable structured query.

3.2. Creating an Entity Embedding Catalog

To enable entity resolution, we must first create a searchable catalog of embeddings for the key entities in our database. For this example, we will focus on the ent_projects table. The strategy is to generate an embedding for the most descriptive text field of each project—its name—and store this embedding alongside its composite primary key.

Database Setup

A new table is created to store the project embeddings and their corresponding primary keys.

SQL Schema:

SQL

```sql
CREATE TABLE project_embeddings (
    project_id VARCHAR,
    project_year INT,
    project_name TEXT,
    embedding VECTOR(1536),
    PRIMARY KEY (project_id, project_year)
);

-- Create an HNSW index for fast lookups
CREATE INDEX ON project_embeddings USING hnsw (embedding vector_cosine_ops);
```

Embedding and Storage Script

The following script reads all entries from the ent_projects table, generates an embedding for each project's name, and populates the project_embeddings catalog. In a production system, this process would be automated to run periodically or triggered by updates to the ent_projects table to keep the catalog synchronized.

Complete Python Script (embed_entities.py):

Python

```python
import psycopg2
from pgvector.psycopg2 import register_vector
import numpy as np
# Assume get_openai_embedding and get_db_connection are available

def populate_project_embeddings(conn):
    """
    Reads from ent_projects, generates embeddings for project names,
    and stores them in the project_embeddings table.
    """
```

```
    with conn.cursor() as cur:
        # Setup the embeddings table
        cur.execute("""
        CREATE TABLE IF NOT EXISTS project_embeddings (
            project_id VARCHAR,
            project_year INT,
            project_name TEXT,
            embedding VECTOR(1536),
            PRIMARY KEY (project_id, project_year)
        );
        """)
        cur.execute("CREATE INDEX IF NOT EXISTS proj_hnsw_idx ON project_embeddings USING hnsw (embedding vector_cosine_ops);")
        cur.execute("TRUNCATE project_embeddings;")

        # Fetch all projects from the source table
        # NOTE: In a real database, you would fetch from the actual 'ent_projects' table.
        # For this example, we will insert some dummy data into a temporary 'ent_projects' table.
        cur.execute("""
        CREATE TABLE IF NOT EXISTS ent_projects (
            id VARCHAR, year INT, name TEXT, PRIMARY KEY (id, year)
        );
        TRUNCATE ent_projects;
        INSERT INTO ent_projects (id, year, name) VALUES
            ('proj-dt-001', 2024, 'Digital Transformation Program'),
            ('proj-cloud-002', 2024, 'Cloud Migration Initiative'),
            ('proj-sec-003', 2023, 'Cybersecurity Overhaul Project'),
            ('proj-data-004', 2024, 'Data Governance Framework Implementation');
        """)
        conn.commit()

        cur.execute("SELECT id, year, name FROM ent_projects;")
        projects = cur.fetchall()

        print(f"Found {len(projects)} projects to embed.")

        for project_id, project_year, project_name in projects:
            if not project_name:
                continue

            embedding = get_openai_embedding(project_name)
            if embedding:
```

```
            print(f"Embedding project: {project_name}")
            cur.execute(
                """
                INSERT INTO project_embeddings (project_id, project_year, project_name, embedding)
                VALUES (%s, %s, %s, %s)
                """,
                (project_id, project_year, project_name, np.array(embedding))
            )
    conn.commit()
    print("Project embeddings catalog populated successfully.")

if __name__ == "__main__":
    connection = get_db_connection()
    if connection:
        populate_project_embeddings(connection)
        connection.close()
```

3.3. Implementation: Real-time Entity Linking Function

With the catalog in place, we can now implement the core entity resolution function. This function serves as a "semantic pre-processor" in our application architecture. It takes a fuzzy text query from the user and returns the precise, structured identifier needed for subsequent database operations.

Complete Python Function (resolve_entities.py):

Python

```
import psycopg2
from pgvector.psycopg2 import register_vector
import numpy as np
# Assume get_openai_embedding and get_db_connection are available

def resolve_project_entity(conn, query_text: str, similarity_threshold: float = 0.75):
    """
```

```
    Resolves a natural language query to a specific project entity.

    Args:
        conn: Active database connection.
        query_text (str): The user's description of the project.
        similarity_threshold (float): The minimum similarity score to consider a match.

    Returns:
        tuple | None: A tuple containing (project_id, project_year, project_name) or None if no match is found.
    """
    query_embedding = get_openai_embedding(query_text)
    if not query_embedding:
        return None

    with conn.cursor() as cur:
        register_vector(cur)
        # We calculate similarity (1 - distance) and filter by it
        cur.execute(
            """
            SELECT project_id, project_year, project_name, 1 - (embedding <=> %s) AS similarity
            FROM project_embeddings
            WHERE 1 - (embedding <=> %s) > %s
            ORDER BY similarity DESC
            LIMIT 1;
            """,
            (np.array(query_embedding), np.array(query_embedding), similarity_threshold)
        )
        result = cur.fetchone()

        if result:
            return (result, result, result) # (id, year, name)
        else:
            return None

if __name__ == "__main__":
    # Assume populate_project_embeddings has been run
    connection = get_db_connection()
    if connection:
        user_mention = "digital transformation initiatives"
        print(f"Resolving entity for user mention: '{user_mention}'")
```

```
    resolved_entity = resolve_project_entity(connection, user_mention)

    if resolved_entity:
        project_id, project_year, project_name = resolved_entity
        print("\n--- Entity Resolved ---")
        print(f"  Matched Project Name: '{project_name}'")
        print(f"  Database PK (id):   {project_id}")
        print(f"  Database PK (year): {project_year}")
    else:
        print("\nCould not resolve the entity to a specific project.")

    connection.close()
```

This function is a critical architectural component. It encapsulates the semantic resolution logic, providing a clean interface that other parts of the application can use. By invoking this function first, an AI agent can obtain the precise identifiers needed to confidently construct accurate SQL queries, thereby avoiding the pitfalls of generating SQL based on ambiguous user input.

Part IV: Architecting Intelligent Database Interactions with LLM Function Calling

This capstone section integrates all the preceding components into a single, cohesive, and powerful system: an LLM-powered agent that can intelligently query the Enterprise Architecture database. By leveraging OpenAI's function calling feature, the LLM transitions from a simple text generator to a sophisticated orchestration engine, deciding when to use specialized tools—like our semantic search functions—to answer complex user questions.

4.1. The Function Calling Paradigm: An Orchestration Engine

Function calling is a feature of advanced LLMs that allows them to interact with external tools and APIs. It operates as a multi-step orchestration loop, transforming the LLM into a planner that can reason about which actions to take to fulfill a user's request.[22] The process consists of three main stages:

1. **LLM as Planner:** The application sends the user's prompt to the LLM, along with a list of available "tools." Each tool is described by a function schema that specifies its name, purpose, and parameters. The LLM analyzes the user's intent and, instead of generating a direct answer, returns a structured JSON object. This object contains a request to call a specific function with the arguments it deems necessary to proceed.
2. **Application as Executor:** The application code is responsible for parsing this JSON response. It identifies the requested function and its arguments, executes the corresponding Python function (e.g., our semantic_entity_search function), and captures the return value. This step grounds the LLM's plan in reality by executing code and retrieving real data.
3. **LLM as Synthesizer:** The application makes a second call to the LLM. This time, it appends the result of the function call to the conversation history. The LLM now has the specific data it requested and uses this new context to synthesize a final, coherent, and data-driven natural language response for the user.

This loop effectively delegates tasks to the component best suited for them. The LLM handles natural language understanding and planning, while the application code handles execution and data retrieval.

4.2. Defining the semantic_search Tool for OpenAI

To make our semantic search capabilities available to the LLM, we must define them according to the JSON schema required by the OpenAI tools parameter. A well-defined schema is critical, as the descriptions provided for the function and its parameters are the primary instructions the LLM uses to decide when and how to call the tool.[25]

The following schema defines a versatile semantic_entity_search function that can resolve different types of entities within our database.

Complete JSON Schema for the Tool:

JSON

```
tools_schema =,
                    "description": "The type of entity to search for. Must be one of 'project', 'capability', or
```

```
'risk'."
                }
            },
            "required": ["query_text", "entity_type"]
        }
    }
]
```

Breakdown of the Schema Fields:

- name: semantic_entity_search. This is the function name the LLM will request to call.
- description: This is a crucial, high-level instruction. It tells the LLM what the function does ("Searches for specific enterprise entities...") and provides examples, guiding it to recognize when this tool is appropriate.
- parameters: This object defines the function's arguments.
  - properties: Each key within this object is an argument.
    - query_text: The description here guides the LLM on what kind of text to extract from the user's prompt (e.g., "digital transformation initiatives").
    - entity_type: The enum constraint is very powerful. It forces the LLM to choose from a valid list of options, preventing it from hallucinating non-existent entity types and ensuring the call to our backend function will succeed.
  - required: This list informs the LLM that both query_text and entity_type must be provided for the function call to be valid.

4.3. Implementing the Python Tool

The semantic_entity_search function in our Python code is the concrete implementation of the tool described in the schema. It acts as a router, dispatching the request to the appropriate entity resolution function based on the entity_type parameter.

Python

```
# This function would be part of our main application logic.
# It relies on the entity resolution functions defined in Part III.
```

```
def semantic_entity_search(query_text: str, entity_type: str):
    """
    Backend implementation of the tool. Routes the search to the correct
    entity resolution function.
    """
    print(f"\n: Running semantic_entity_search(query_text='{query_text}', entity_type='{entity_type}')")

    connection = get_db_connection()
    if not connection:
        return json.dumps({"error": "Database connection failed."})

    resolved_entity = None
    if entity_type == "project":
        # We would have similar functions for 'capability' and 'risk'
        resolved_entity = resolve_project_entity(connection, query_text)
    # elif entity_type == "capability":
    #     resolved_entity = resolve_capability_entity(connection, query_text)
    # elif entity_type == "risk":
    #     resolved_entity = resolve_risk_entity(connection, query_text)
    else:
        return json.dumps({"error": f"Unknown entity type: {entity_type}"})

    connection.close()

    if resolved_entity:
        # Return a structured JSON string for the LLM to process
        entity_id, entity_year, entity_name = resolved_entity
        return json.dumps({
            "status": "success",
            "entity_type": entity_type,
            "name": entity_name,
            "id": entity_id,
            "year": entity_year
        })
    else:
        return json.dumps({"status": "not_found"})
```

4.4. The Complete Orchestration Flow: A Full Example

This final, end-to-end script demonstrates the complete agentic loop, combining the LLM's planning capabilities with our pgvector-powered tools and traditional SQL queries to answer a complex business question.[28]

Scenario: A user asks, *"What are the main risks associated with our digital transformation projects?"*

Complete Python Orchestration Script (llm_agent.py):

Python

```
import os
import json
import psycopg2
from openai import OpenAI
from dotenv import load_dotenv

# --- Assume all previous helper functions are available ---
# get_db_connection(), resolve_project_entity(), etc.

# Load environment variables
load_dotenv()
client = OpenAI()

# --- Tool Definition and Implementation ---

tools_schema =, "description": "The type of entity to search for."}
            },
            "required": ["query_text", "entity_type"]
        }
    }
]

def semantic_entity_search(query_text: str, entity_type: str):
    """Backend implementation of the tool."""
    print(f"\n: Running semantic_entity_search(query_text='{query_text}', entity_type='{entity_type}')")
```

```
    conn = get_db_connection()
    if not conn: return json.dumps({"error": "Database connection failed."})

    resolved_entity = None
    if entity_type == "project":
        resolved_entity = resolve_project_entity(conn, query_text)
    else:
        conn.close()
        return json.dumps({"error": f"Search for entity type '{entity_type}' is not implemented."})

    if resolved_entity:
        project_id, project_year, project_name = resolved_entity

        # Now, perform a structured SQL query using the resolved ID
        print(f": Found project '{project_name}'. Now querying for associated risks.")
        with conn.cursor() as cur:
            # This SQL query is a placeholder representing the logic to join projects to risks.
            # The actual join path in the full schema might be more complex (e.g., Project -> Capability ->
Risk).
            # For this example, we'll assume a direct or simplified link for clarity.
            # Let's create some dummy risk data for the demo.
            cur.execute("""
            CREATE TABLE IF NOT EXISTS ent_risks (id VARCHAR, year INT, name TEXT, risk_description
TEXT, PRIMARY KEY (id, year));
            TRUNCATE ent_risks;
            INSERT INTO ent_risks (id, year, name, risk_description) VALUES
                ('risk-001', 2024, 'Vendor Lock-in', 'Over-reliance on a single cloud provider could lead to
increased costs and reduced flexibility.'),
                ('risk-002', 2024, 'Data Migration Failure', 'Risk of data loss or corruption during the migration
from on-premise to cloud infrastructure.'),
                ('risk-003', 2024, 'Low User Adoption', 'Employees may resist new workflows and tools, leading
to unrealized project benefits.');

            -- Dummy join table
            CREATE TABLE IF NOT EXISTS jt_project_risks (project_id VARCHAR, project_year INT, risk_id
VARCHAR, risk_year INT);
            TRUNCATE jt_project_risks;
            INSERT INTO jt_project_risks (project_id, project_year, risk_id, risk_year) VALUES
                ('proj-dt-001', 2024, 'risk-001', 2024),
                ('proj-dt-001', 2024, 'risk-002', 2024),
                ('proj-dt-001', 2024, 'risk-003', 2024);
            """)
```

```
            conn.commit()

            cur.execute("""
              SELECT r.name, r.risk_description
              FROM ent_risks r
              JOIN jt_project_risks j ON r.id = j.risk_id AND r.year = j.risk_year
              WHERE j.project_id = %s AND j.project_year = %s;
            """, (project_id, project_year))
            risks = cur.fetchall()
            conn.close()

            if risks:
                return json.dumps([{"name": r, "description": r} for r in risks])
            else:
                return json.dumps({"status": "success", "message": "No risks found for this project."})
    else:
        conn.close()
        return json.dumps({"status": "not_found"})

def run_conversation(user_prompt: str):
    """Main orchestration loop for the LLM agent."""
    print(f"--- Starting Conversation for Prompt: '{user_prompt}' ---")
    messages = [{"role": "user", "content": user_prompt}]

    # --- Step 1: Initial LLM Call (Planner) ---
    print("\n: Sending user prompt and tools to LLM...")
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=messages,
        tools=tools_schema,
        tool_choice="auto",
    )
    response_message = response.choices.message
    messages.append(response_message)

    # --- Step 2: Application Execution ---
    if response_message.tool_calls:
        print(": LLM requested a tool call. Executing...")
        tool_call = response_message.tool_calls
        function_name = tool_call.function.name
        function_args = json.loads(tool_call.function.arguments)
```

```
        if function_name == "semantic_entity_search":
            function_response = semantic_entity_search(
                query_text=function_args.get("query_text"),
                entity_type=function_args.get("entity_type")
            )

            # --- Step 3: Second LLM Call (Synthesizer) ---
            print("\n: Sending tool response back to LLM for final answer...")
            messages.append(
                {
                    "tool_call_id": tool_call.id,
                    "role": "tool",
                    "name": function_name,
                    "content": function_response,
                }
            )

            final_response = client.chat.completions.create(
                model="gpt-4o",
                messages=messages,
            )
            print("\n--- Final AI Response ---")
            print(final_response.choices.message.content)
        else:
            print(f"Error: LLM called an unknown function '{function_name}'")
    else:
        print("\n--- Final AI Response (No Tool Call) ---")
        print(response_message.content)

if __name__ == "__main__":
    # Ensure the project embeddings table is populated before running
    conn = get_db_connection()
    if conn:
        populate_project_embeddings(conn)
        conn.close()

    run_conversation(user_prompt="What are the main risks associated with our digital
transformation projects?")
```

This script perfectly illustrates the power of the hybrid architecture. The LLM acts as an

intelligent "semantic glue," delegating tasks to the components best suited for them. It uses its natural language understanding to plan the query, invokes pgvector via function calling to resolve the fuzzy semantic part of the query, and then the application uses the precise result of that resolution to execute a high-performance, structured SQL query. Finally, the LLM synthesizes the raw data from the SQL query into a polished, human-readable answer. This robust pattern creates a system that is more powerful, reliable, and efficient than one that attempts to force a single component to handle every task.

Conclusion

This report has detailed a comprehensive journey from the foundational principles of pgvector to the architecture of a sophisticated, AI-powered database agent. The analysis demonstrates a clear and powerful pattern for building modern enterprise applications by integrating vector search capabilities directly within a PostgreSQL database.

The core architectural principles that emerge are clear:

1. **Unify Data Management:** By using pgvector, organizations can eliminate the complexity of maintaining separate vector and relational databases, simplifying data pipelines and ensuring transactional consistency.
2. **Delegate Tasks Appropriately:** The most robust AI systems are not monolithic. They delegate tasks to the components best suited for them. Vector search is used for semantic and fuzzy matching tasks like discovery and entity resolution. Traditional SQL is used for high-performance filtering, joining, and aggregation of structured data. The LLM serves as the intelligent orchestrator, using its reasoning capabilities to plan and sequence these tasks.
3. **Embrace the "Resolve, then Query" Pattern:** For applications that translate natural language to database operations, introducing an explicit entity resolution step via vector search dramatically increases reliability. By resolving ambiguous user mentions into concrete database identifiers *before* generating complex SQL, the system avoids the brittleness of LIKE clauses and the risk of LLM hallucination in WHERE conditions.

The integration of pgvector within the mature PostgreSQL ecosystem represents more than just a new feature; it signals a shift toward hybrid data platforms that are inherently AI-ready. By following the patterns and implementations detailed in this guide, developers and architects can move beyond traditional keyword-based systems and build a new class of applications that understand intent, resolve ambiguity, and provide truly intelligent access to enterprise data.

Appendix: Enterprise Architecture Data Model Analysis

This appendix provides the essential business context for the database schema used throughout the report's examples. Understanding this model is key to writing meaningful queries and interpreting the system's behavior.

A. Overview of the Business Data Model

The database schema represents a comprehensive model for an enterprise, designed to track and manage the complex interplay between strategy, operations, and change. It is structured around two primary domains: **Enterprise (ent_)** entities, which describe the internal workings of the organization, and **Sector (sec_)** entities, which describe the external context and high-level strategic drivers.

The model links strategic **Objectives** [21] to operational **Capabilities** [21], which define what the organization does. These capabilities are, in turn, supported by three pillars:

- **People:** Modeled in the ent_org_units table.
- **Processes:** Modeled in the ent_processes table.
- **Technology:** Modeled in the ent_it_systems table.

This structure allows the organization to analyze gaps, such as identifying which IT systems support a critical business capability. Furthermore, the model tracks **Projects** (ent_projects) as the primary agents of change and **Risks** (ent_risks) as potential impediments to achieving objectives or sustaining capabilities.[21] A defining feature of the schema is its temporal nature; most core tables use a composite primary key of (id, year), enabling the system to maintain a historical record and track the evolution of the enterprise architecture over time.[21]

B. Key Entity Relationships Table

The connectors.csv file [21] defines the explicit relationships (the "verbs") that connect the core entities (the "nouns" from components.csv [21]). The following table decodes the most

significant of these relationships into a human-readable format, providing a quick reference for understanding the business logic embedded in the database structure.

Source Entity (From)	Target Entity (To)	Relationship (Label)	Business Implication
8: Objectives	1: Policy Tools	Realized Via	Strategic objectives are achieved through the implementation of policy tools.
8: Objectives	7: Performance	Cascaded Via	High-level objectives are broken down into measurable performance indicators (KPIs).
6: Data Transaction	7: Performance	Measured By	The volume and efficiency of data transactions are measured by performance metrics.
7: Performance	8: Objectives	Aggregates To	Performance metrics roll up to indicate progress against strategic objectives, forming a feedback loop.
10: Capability	12: Organization	Role Gaps	A capability may have a dependency on specific organizational

			roles; a lack of these roles creates a "role gap."
10: Capability	13: Processes	Knowledge Gaps	A capability requires specific knowledge, which is typically embedded in processes; a lack of this knowledge is a "knowledge gap."
10: Capability	14: IT Systems	Automation Gaps	A capability can be enhanced or enabled by IT systems; a lack of sufficient automation is an "automation gap."
16: Projects Outputs	12: Organization	Close Gaps	The deliverables (outputs) of projects are intended to fill identified gaps in organizational roles.
16: Projects Outputs	13: Processes	Close Gaps	Project outputs are intended to close gaps in process knowledge.
16: Projects Outputs	14: IT Systems	Close Gaps	Project outputs, such as new software, are

			intended to close automation gaps.
16: Projects Outputs	17: Change Architecture	Adoption Risks	The successful implementation of project outputs faces risks related to their adoption within the organization's change architecture.
14: IT Systems	15: Vendor SLA	Depends On	The functionality or reliability of an IT system depends on services provided by external vendors, governed by Service Level Agreements (SLAs).

Works cited

1. What is pgvector, and How Can It Help Your Vector Database? - EDB Postgres AI, accessed October 27, 2025, https://www.enterprisedb.com/blog/what-is-pgvector
2. GenAI and LLMswith pgvector for PostgreSQL with Tessell, accessed October 27, 2025, https://www.tessell.com/genai-and-llms-with-pgvector
3. Uncovering the Power of Vector Databases with pgvector: An Introduction, accessed October 27, 2025, https://www.datascienceengineer.com/blog/post-what-is-pgvector
4. Turning PostgreSQL Into a Vector Database With pgvector | Tiger Data, accessed October 27, 2025, https://www.tigerdata.com/learn/postgresql-extensions-pgvector
5. Using Pgvector With Python | Tiger Data, accessed October 27, 2025, https://www.tigerdata.com/learn/using-pgvector-with-python
6. pgvector for Python developers, accessed October 27, 2025, https://pamelafox.github.io/my-py-talks/pgvector-python/

7. Getting Started with Vector Databases with `pgvector` and Python. 1 ..., accessed October 27, 2025, https://anas-rz.medium.com/getting-started-with-vector-databases-with-pgvector-and-python-1-n-65effe0bfdd6
8. Storing OpenAI embeddings in Postgres with pgvector - Supabase, accessed October 27, 2025, https://supabase.com/blog/openai-embeddings-postgres-vector
9. PostgreSQL as Vector database: Create LLM Apps with pgvector | by Rakesh | Tessell DBaaS | Medium, accessed October 27, 2025, https://medium.com/tessell-dbaas/postgresql-as-vector-database-create-llm-apps-with-pgvector-64677de48fc2
10. Simplifying RAG with PostgreSQL and PGVector | by Levi Stringer - Medium, accessed October 27, 2025, https://medium.com/@levi_stringer/rag-with-pg-vector-with-sql-alchemy-d08d96bfa293
11. Vector Embeddings with OpenAI in Python | CodeSignal Learn, accessed October 27, 2025, https://codesignal.com/learn/courses/understanding-embeddings-and-vector-representations-3/lessons/vector-embeddings-with-openai-in-python-pgvector
12. Inspecting Distances and Similarity Scores in pgvector | CodeSignal Learn, accessed October 27, 2025, https://codesignal.com/learn/courses/storing-and-managing-embeddings-in-postgresql-with-pgvector/lessons/inspecting-distances-and-similarity-scores-in-pgvector
13. Key vector database concepts for understanding pgvector - TimescaleDB - TigerData, accessed October 27, 2025, https://docs.tigerdata.com/ai/latest/key-vector-database-concepts-for-understanding-pgvector/
14. pgvector/pgvector: Open-source vector similarity search for Postgres - GitHub, accessed October 27, 2025, https://github.com/pgvector/pgvector
15. The Ultimate Guide to using PGVector | by Intuitive Deep Learning | Medium, accessed October 27, 2025, https://medium.com/@intuitivedl/the-ultimate-guide-to-using-pgvector-76239864bbfb
16. Understanding and Managing Indexes in pgvector | CodeSignal Learn, accessed October 27, 2025, https://codesignal.com/learn/courses/indexing-optimization-and-scaling-pgvector/lessons/understanding-and-managing-indexes-in-pgvector
17. pgvector, a guide for DBA – Part2 indexes - dbi services, accessed October 27, 2025, https://www.dbi-services.com/blog/pgvector-a-guide-for-dba-part2-indexes/
18. Building Production-Grade Semantic Search with Vector Databases | Folder IT, accessed October 27, 2025,

https://folderit.net/building-production-grade-semantic-search-with-vector-databases/
19. Semantic search | Supabase Docs, accessed October 27, 2025, https://supabase.com/docs/guides/ai/semantic-search
20. Best Approaches for Vectorizing Relational Databases for Natural Language Querying, accessed October 27, 2025, https://community.openai.com/t/best-approaches-for-vectorizing-relational-databases-for-natural-language-querying/1108727
21. DB as JSON.txt
22. How to use function calling with Azure OpenAI in Azure AI Foundry Models - Microsoft Learn, accessed October 27, 2025, https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/function-calling
23. Understanding OpenAI Function Calling - .NET - Microsoft Learn, accessed October 27, 2025, https://learn.microsoft.com/en-us/dotnet/ai/conceptual/understanding-openai-functions
24. How to Build An AI Agent with Function Calling and GPT-5 | Towards Data Science, accessed October 27, 2025, https://towardsdatascience.com/how-to-build-an-ai-agent-with-function-calling-and-gpt-5/
25. OpenAI Function Calling Tutorial: Generate Structured Output ..., accessed October 27, 2025, https://www.datacamp.com/tutorial/open-ai-function-calling-tutorial
26. Function calling with the Gemini API | Google AI for Developers, accessed October 27, 2025, https://ai.google.dev/gemini-api/docs/function-calling
27. Getting started with OpenAI function calling | by Chas Sweeting - Medium, accessed October 27, 2025, https://medium.com/@chassweeting/tips-and-gotchas-working-with-openai-function-calling-e429276d8752
28. An introduction to function calling and tool use - Apideck, accessed October 27, 2025, https://www.apideck.com/blog/llm-tool-use-and-function-calling
29. Understanding Function Calling in LLMs - Zilliz blog, accessed October 27, 2025, https://zilliz.com/blog/harnessing-function-calling-to-build-smarter-llm-apps