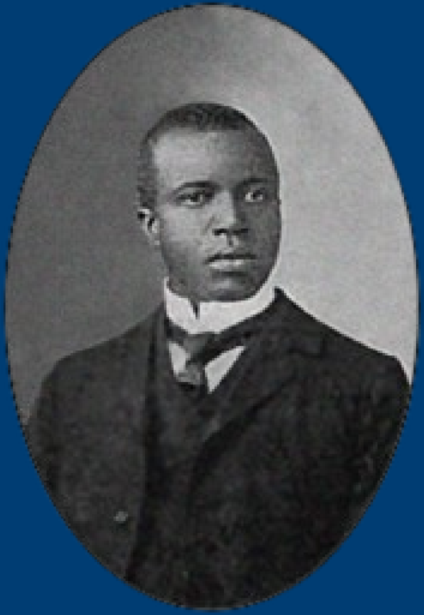
RAGtime with Postgres

AI power with pgvector and Retrieval-Augmented Generation

FOSSY 2025

Jimmy Angelakos

RAGtime?



- Scott Joplin: “King of Ragtime”
- Not about the music genre
- Retrieval-Augmented Generation

Hype vs Reality

- LLM, GPT, Vector DB, AGI, MCP, Langchain, AI Agents…
- Data lakes/Warehouses/Lakehouses, Serverless DBs…
- Skepticism
  - “As we learn about how the technology works, we realize that GenAI is nothing but statistically derived plagiarism” —Prof. Ulises A. Mejias (SUNY)
- Building practical tools is the answer

About me (Postgres nerd, not AI guru)

- Systems & Database Architect, based in Edinburgh, Scotland
- Open Source user & contributor (25+ years)
- PostgreSQL exclusively (17+ years), Contributor
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- Co-author, PostgreSQL 16 Administration Cookbook
- `pg_statviz` PostgreSQL extension

What we'll cover

- What?
- Why?
- How?

- Problem solving
- Building
- Tips
- Caveats

What is GenAI?

- GenAI: Generative (statistical) model which produces pictures, audio, video, etc.
- Deep generative model (as in *deep learning*)
  - Trained on large corpus of ~~copyrighted~~ works

What are LLMs?

- LLM: Large Language Model, a deep neural network for language-based tasks

- GPT: Generative Pre-trained Transformer

What is the problem with LLMs?

- How much time do you have? 🥁
- Seriously now… they can lie
  - Make stuff up when they don’t know the answer
  - “Hallucination” (politest term I could find)
- Their knowledge is frozen at the time of training
  - Gets progressively more stale
  - They don’t have access to your data
- Availability Heuristic – Einstellung Effect (“commonness bias”?)

You're my hallu—, hallucination...👩‍🎤

J How can I tune toast_compression_level in PostgreSQL?

You can tune the `toast_compression_level` parameter in PostgreSQL through several methods, depending on your version and scope requirements.

Setting the Parameter

System-wide configuration: Add to your `postgresql.conf` file:

```
toast_compression_level = 6
```

Then reload the configuration with `SELECT pg_reload_conf();` or restart PostgreSQL.

Reply to Claude...

Claude Sonnet 4

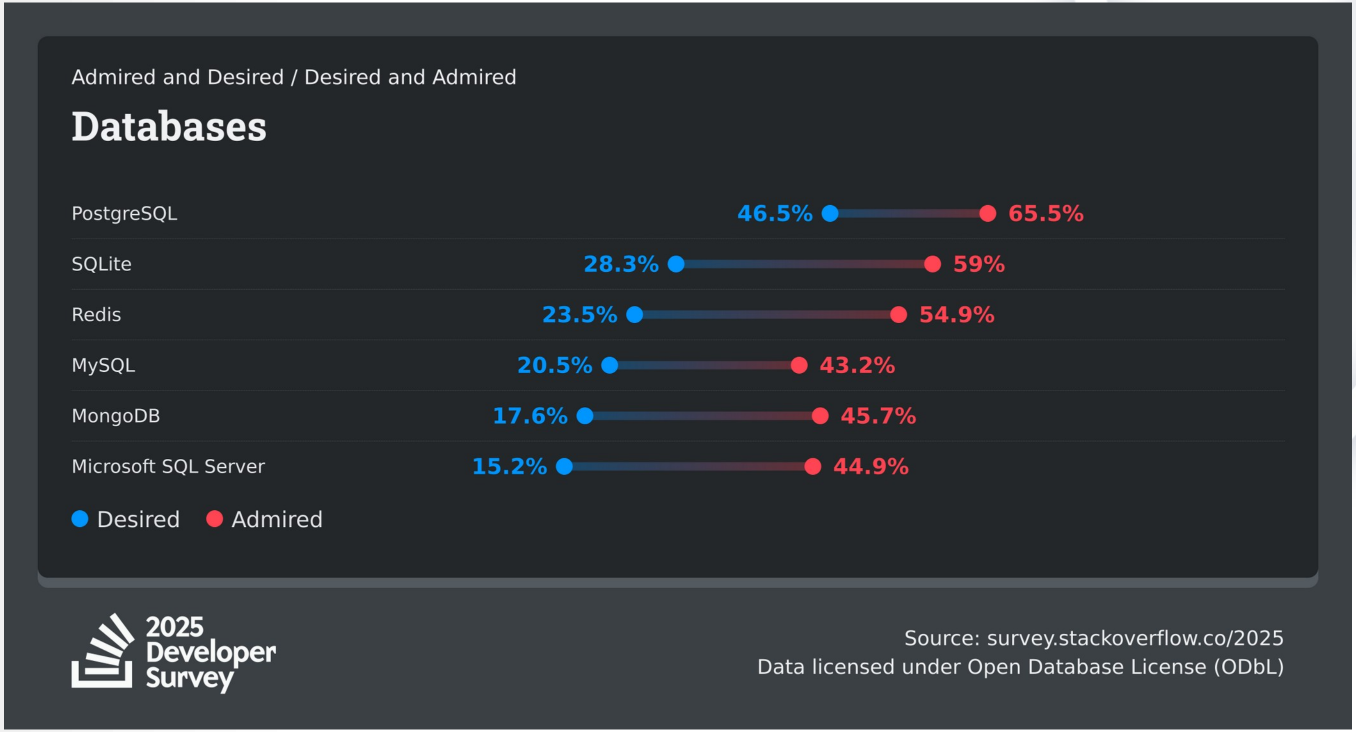
What is RAG?

- Baby don't hurt me

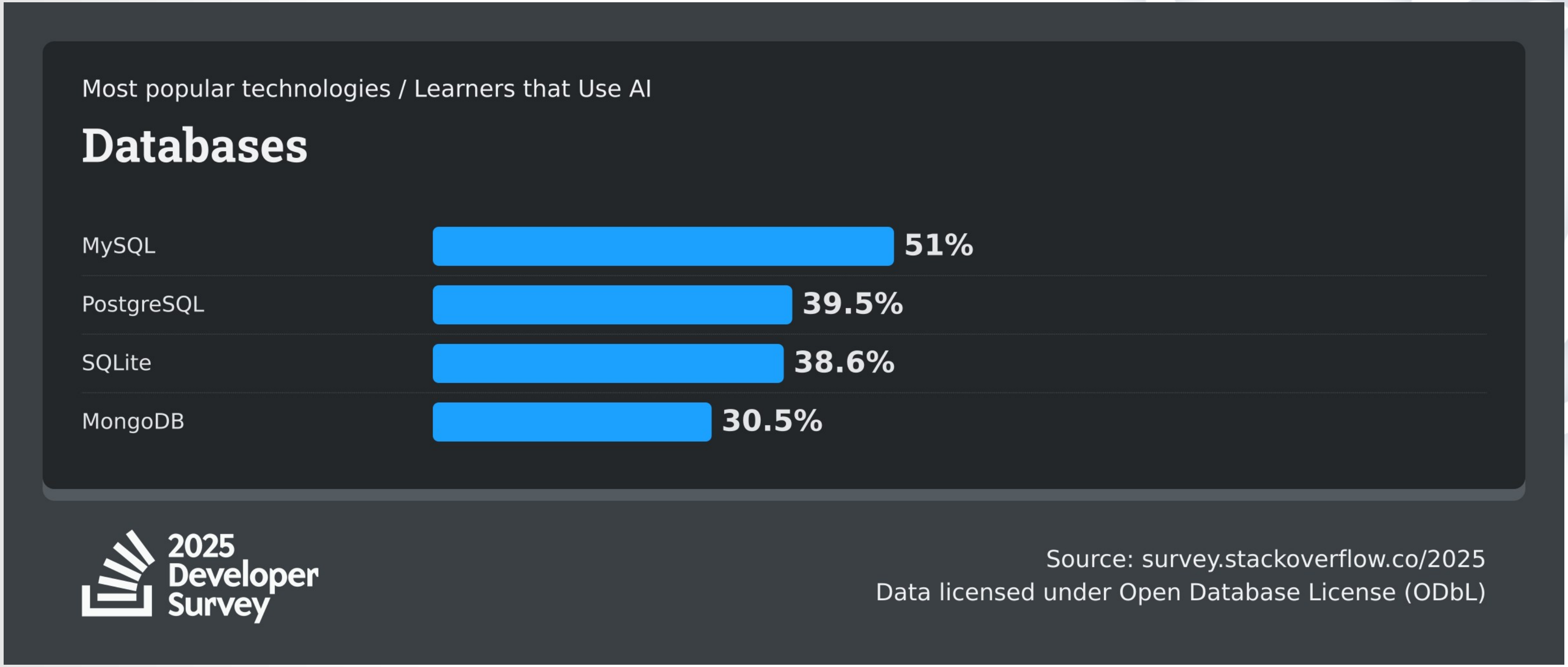
What is RAG?

- Retrieval-Augmented Generation
  - Incorporates information retrieval in the LLM response generation
  - Addresses the aforementioned problems
    - Reduces hallucinations
    - Data is fresh
  - Supplements information that was not available in the LLM training data

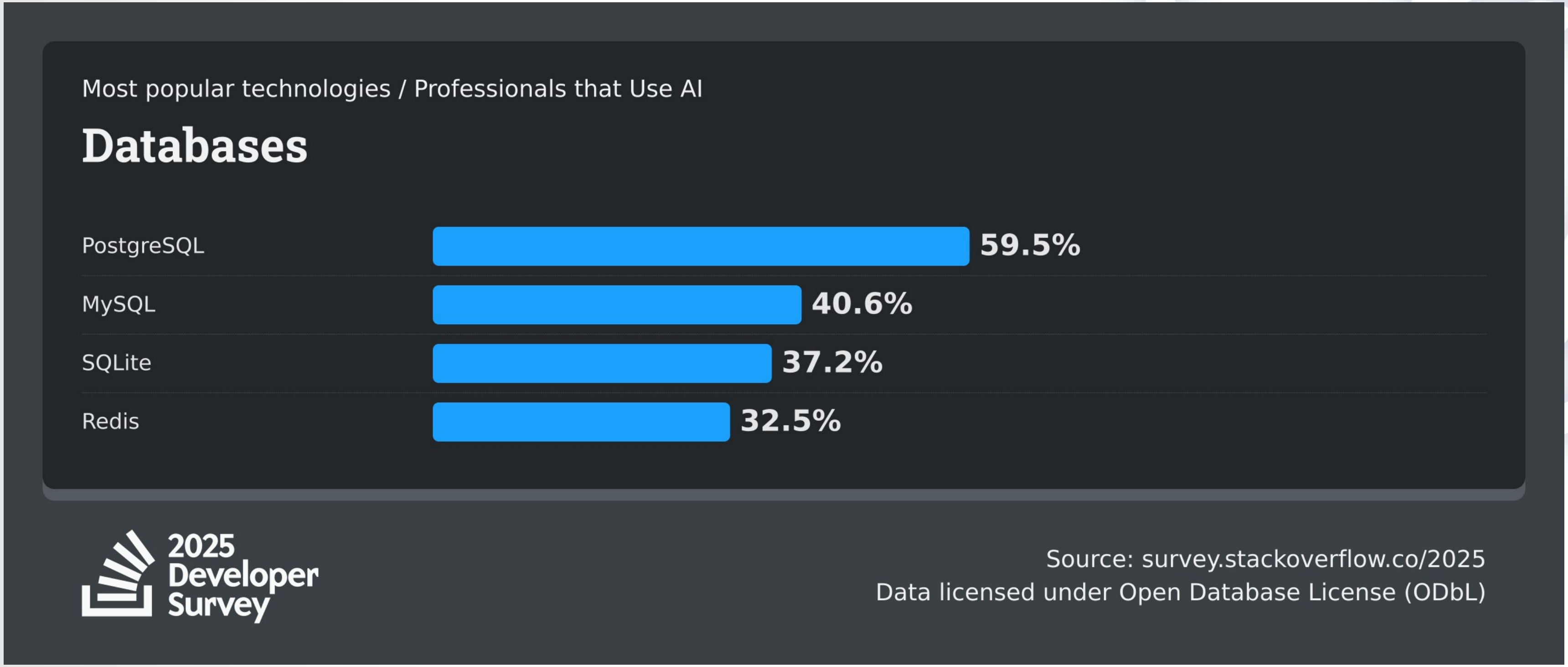
What can we use for information retrieval? 🤔



What can we use for AI? (Learners)



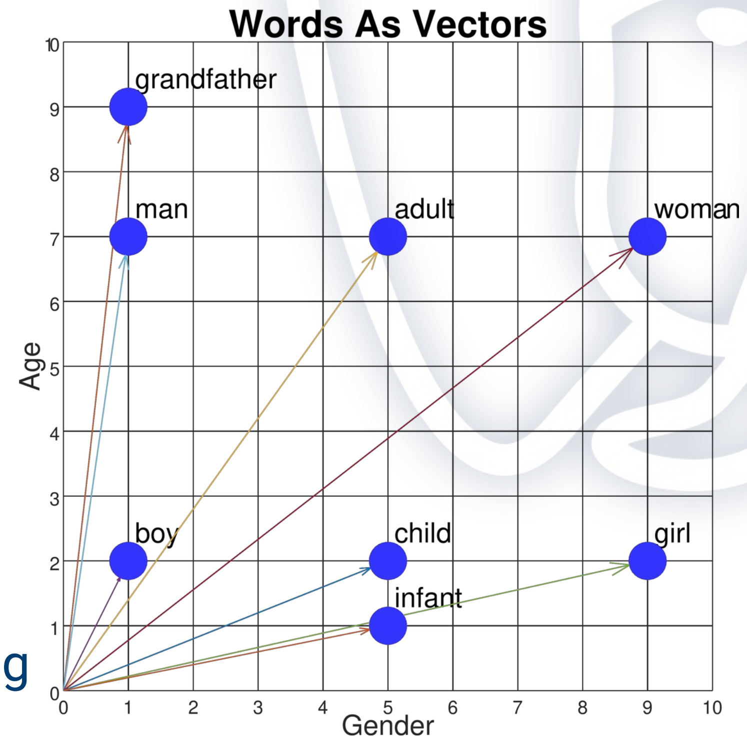
What can we use for AI? (Professionals)



How does RAG work? (i)

- The secret sauce: vector embeddings
  - aka word embeddings
  - Words encoded into vectors of real numbers
- Words clustered together by meaning
  - Distributional semantics

https://github.com/touretzkyds/WordEmbeddingDemo



How does RAG work? (ii)

- "Indexing"
  - Convert the text input using an *embedding model*
  - Store the embeddings generated in a database
- "Retrieval"
  - User query fetches the most relevant documents
- "Augmentation"
  - Prompt engineering by feeding in the retrieved documents
- "Generation"
  - Craft the response with the LLM

How does RAG work in Postgres? (i)

- `pgvector` extension is your friend
  - Adds **`vector`** data type
  - Store it as a simple column in a table
- Now the good stuff is inside your database
  - You can store it alongside the original document and other metadata

How does RAG work in Postgres? (ii)

```
CREATE EXTENSION pgvector;
CREATE TABLE documents (
   content text
   embedding vector(384)
);
```

How does RAG work in Postgres? (iii)

```
SELECT content
FROM documents
ORDER BY
   (1 - embedding <-> 'my query'::vector);
```

- L2 (Euclidean) distance between two points or vectors

Okay, but isn’t this going to be slow?

- Isn’t this like searching with **`LIKE`**?
- No, `pgvector` offers indexes
  - `IVFFlat`
  - `HNSW`

```
CREATE INDEX ON documents
USING ivfflat (embedding vector_l2_ops)
WITH (lists = 100);
```

Problem Solving

The problem

- You have a large legacy codebase
- What if your code experts
  - have left the organization?
  - are too busy with other work?
  - have forgotten what they did 10 years ago?

A possible solution

- Create a RAG system to be your “code expert”
- Index your code repositories’ content
- Create an interface to query said system
  - Web chatbot
  - Slackbot

Let’s build a solution

A real-world RAG application workflow

- Retrieve code from repositories
- Generate embeddings and into database
- Enter natural language query into interface
- Optimize query
- Use optimized query to retrieve code snippets from database
- Add code snippet context to original query to augment prompt
- Send engineered prompt to LLM
- Format and display response
- Bonus points: keep conversation history for each interaction

Clone repositories locally

- Write a Python script that clones the repos

```
import os
from github import Github
github_token = os.getenv('GITHUB_TOKEN')
g = Github(github_token)
user = g.get_user()
repos = user.get_repos(affiliation='owner')
for repo in repos:
    ...
```

Prepare your database

```
CREATE TABLE source_embeddings (
  repo_name text not null,
  file_path text not null,
  content text not null,
  semantic_embedding vector(384),
  code_embedding vector(256),
  created_at timestamptz DEFAULT now(),
  git_commit text,
  PRIMARY KEY(repo_name, file_path)
);
```

Index your repositories (i)

- Write a Python script that creates the embeddings in the DB for each file
  - Dual indexing strategy (semantic and code embeddings)

```
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModel
semantic_model = SentenceTransformer(
    'sentence-transformers/all-MiniLM-L6-v2', device='cpu',
    model_kwargs={'attn_implementation': 'eager'})
code_model = AutoModel.from_pretrained(
    'Salesforce/codet5p-110m-embedding', trust_remote_code=True,
    attn_implementation="eager")
code_tokenizer = AutoTokenizer.from_pretrained(
    'Salesforce/codet5p-110m-embedding', trust_remote_code=True)
```

Index your repositories (ii)

- Preprocess code for better embedding quality
  - Normalize whitespace within each line
  - Remove comments
  - Replace string literals with a placeholder
  - Remove consecutive blank lines

```
lines = text.split('\n')
processed_lines = []
for line in lines:
    line = re.sub(r'\s+', ' ', line)
    line = re.sub(r'#.*$', '', line)
    processed_lines.append(line.strip())
processed_text = '\n'.join(processed_lines)
processed_text = re.sub(r'"[^"]*"', '"STR"', processed_text)
processed_text = re.sub(r"'[^']*'", "'STR'", processed_text)
processed_text = re.sub(r'\n\s*\n\s*\n', '\n\n', processed_text))
```

Generate embeddings (i)

- Chunking is an imprecise art (trial and error)
- For semantic, split code into overlapping chunks

```
def chunk_code_for_semantic(
    text: str,
    chunk_size: int = 1000,
    overlap: int = 200) -> List[str]:
    ...
```

Generate embeddings (ii)

- For CodeT5+ we use smaller chunks to match its maximum context window

```
def chunk_code_for_codet5(
    text: str,
    chunk_size: int = 512) -> List[str]:
    ...
```

Generate embeddings (iii)

- Tokenize the input

```
import torch
import numpy
for chunk in chunks:
    inputs = code_tokenizer(chunk,
        padding=True, truncation=True,
        max_length=512, return_tensors="pt")
```

Generate embeddings (iv)

- Generate code embeddings:

```
with torch.no_grad():
    outputs = code_model(**inputs)
    embedding_array =
       outputs.squeeze(0).detach().cpu().numpy()
    chunk_embeddings.append(emb_array)
```

Generate embeddings (v)

- Average and normalize the embeddings from all chunks:

```
stacked = np.stack(chunk_embeddings)
final_embedding = np.mean(stacked, axis=0)
norm = np.linalg.norm(final_embedding)
if norm > 0:
    final_embedding = final_embedding / norm
return [float(x) for x in final_embedding]
```

Generate embeddings (vi)

- Generate semantic embeddings, average and normalize

```
for chunk in chunks:
    semantic_emb = semantic_model.encode(chunk,
        convert_to_tensor=False)
    semantic_chunk_embeddings.append(semantic_emb)
    semantic_stacked = np.stack(semantic_chunk_embeddings)
    semantic_array = np.mean(semantic_stacked, axis=0)
    semantic_norm = np.linalg.norm(semantic_array)
    if semantic_norm > 0:
        semantic_array = semantic_array / semantic_norm
    return [float(x) for x in semantic_array]
```

Store embeddings in database

```
INSERT INTO source_embeddings (repo_name,
  file_path, content, semantic_embedding,
  code_embedding, git_commit)
VALUES (%s, %s, %s, %s::vector, %s::vector, %s)
ON CONFLICT (repo_name, file_path)
DO UPDATE SET content = EXCLUDED.content,
  semantic_embedding = EXCLUDED.semantic_embedding,
  code_embedding = EXCLUDED.code_embedding,
  git_commit = EXCLUDED.git_commit;
```

Index embeddings

```
CREATE INDEX ON source_embeddings
USING ivfflat (semantic_embedding
  vector_cosine_ops) WITH (lists = 100);
CREATE INDEX ON source_embeddings
USING ivfflat (code_embedding
  vector_cosine_ops) WITH (lists = 100);
```

Query the database (i)

- Here I cheat a little: I have the LLM optimize the natural language prompt
  - It outputs a semantic query and a code query
- Generate query embeddings using both semantic and code queries

```
query_semantic, query_code =
    await get_embedding(query, code_query)
```

- pgvector expects `[x,y,z]` format

```
semantic_vector = f"[{','.join(str(round(x, 8))
    for x in query_semantic)}]"
code_vector = f"[{','.join(str(round(x, 8))
    for x in query_code)}]"
```

Query the database (ii)

```
WITH ranked AS (
SELECT (repo_name, file_path, content,
  semantic_embedding::text, code_embedding::text,
  (1 - (semantic_embedding <=> %s::vector)) AS semantic_sim,
  (1 - (code_embedding <=> %s::vector)) AS code_sim,
  ROW_NUMBER() OVER
  (ORDER BY (1 - (semantic_embedding <=> %s::vector)) DESC)
  AS semantic_rank,
  ROW_NUMBER() OVER
  (ORDER BY (1 - (code_embedding <=> %s::vector)) DESC)
  AS code_rank
FROM source_embeddings)
```

Query the database (iii)

- I then **SELECT FROM** `ranked`
  - The top 20 semantic matches
  - The top 10 code matches
- If they score in both those top brackets they get a boost

```
ORDER BY dual_match DESC, similarity DESC
```

Send the augmented query to the LLM

```
system_prompt = ("""You are an intelligent code assistant, with a heavy focus
on PostgreSQL. Below you will find relevant code patterns from the user's
codebase. Use these patterns to provide accurate, contextual responses about
their specific database implementation.

The code patterns are categorized by match type:
  - HIGHLY RELEVANT: These patterns matched both semantically and by code structure,
  making them particularly important examples.
  - Semantic match: These patterns matched based on natural language understanding
  of the query.
  - Code structure match: These patterns matched based on code structure similarity.

HIGHLY RELEVANT patterns first, then Semantic and Code matches.
While you can describe the patterns you see, do not directly quote the code.
Your response format must be in Markdown. Format any code blocks with ```
prefix."""
f"{code_context}")
```

Sample queries

- For an insurance company:
  - How do we price life insurance contracts?
  - What is the method of calculation for additional contract discounts?
- For the PostgreSQL Europe conference system:
  - What is the workflow for adding an attendee registration to the system?

Tips

Application-side (i)

- I used `FastAPI` to build a web interface and Slack Python libraries to create a bot
  - You can make this into a simple API or cmdline tool
- By using `sentence-transformers` and other free resources you can save costs
  - vs. using an expensive commercial API to index the data
- If your system consists of multiple repositories, this tool can give you answers of system-level scope

Application-side (ii)

- You can send your augmented prompt to an external LLM API
  - Cost ($$$)
  - Stability (this server is overloaded...)
  - Reliability (will the model be the same tomorrow?)
- You can run Ollama locally and run your LLM yourself
  - Best for information security, cost
  - I would use `codellama:7b-instruct` for my chatbot
  - Capabilities may not be on par with commercial offerings

Database-side

- I used `IVFFlat`
  - Faster index creation
- HNSW
  - Can give more accurate results
  - Performance: faster queries
  - Index build time is slower

Caveats

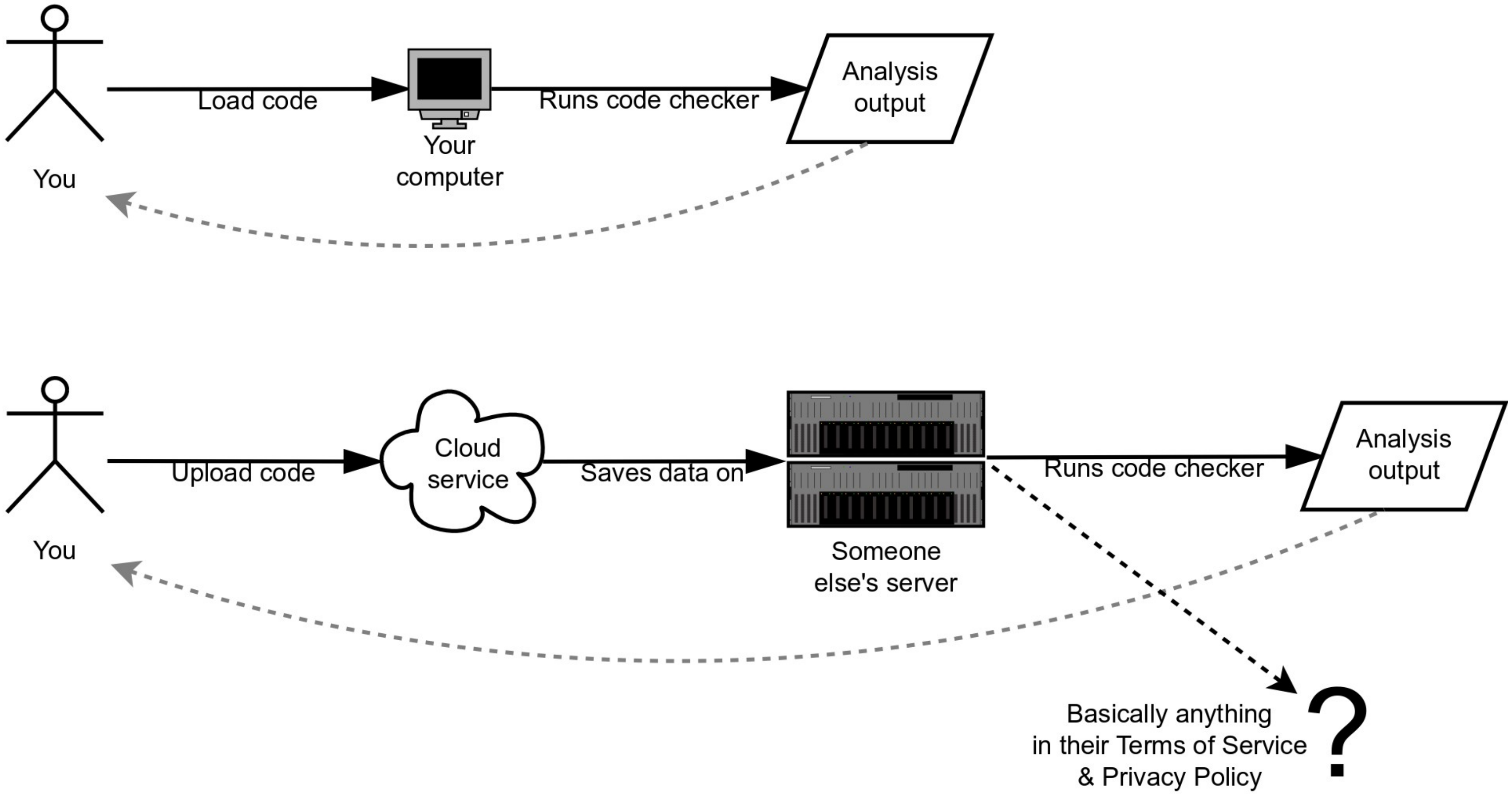
Caveats

(i)

- LLMs can generate misinformation even when pulling from factually correct sources
  - If they misinterpret the context
- "Open source" models
  - Where's the training data?

Caveats

(ii)

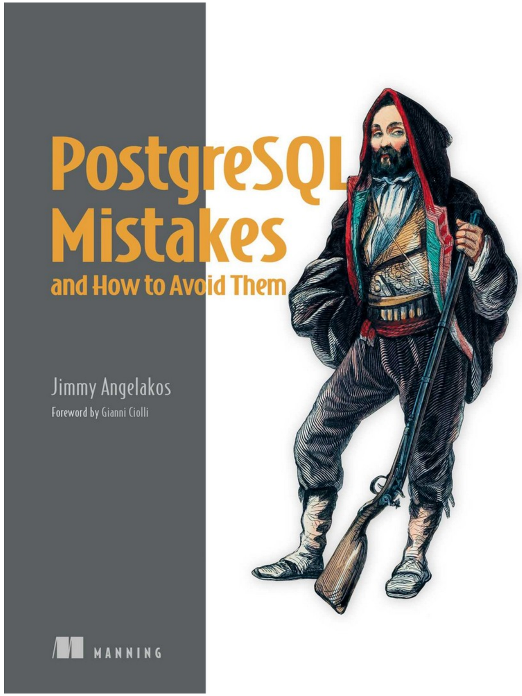


Load code
Runs code checker
Analysis output
Your computer
You
Upload code
Cloud service
Saves data on
Runs code checker
Analysis output
Someone else's server
You
Basically anything in their Terms of Service & Privacy Policy
?

Find me on socials

- YouTube: https://youtube.com/JimmyAngelakos
- Mastodon: https://fosstodon.org/@vyruss
- BlueSky: https://bsky.app/profile/vyruss.org
- LinkedIn: https://linkedin.com/in/vyruss

Thank you ❤️



45% off everything!
Code: **jafossy45**

Questions?