Structured RAG is better than RAG

Guido van Rossum - Microsoft

gvanrossum@microsoft.com
guido@python.org

Intro

We claim that Structured RAG is an improvement over (classic) RAG, and that it can handle longer conversations, is faster, more precise, and can handle more complex queries.

Most of this is not my work but that of my colleagues Steven Lucco and Umesh Madan (who wrote it in TypeScript).

My responsibility is the Python port and the storage architecture.

Umesh and Rob Gruen made the demos possible.

Much of the code was written by various LLMs (usually Claude Sonnet): Translation from TypeScript to Python, refactoring, new features, etc.

Review:

Classic RAG

(Retrieval-Augmented Generation)

Agents need memory

- To recall relevant past interactions
- To retrieve relevant documents

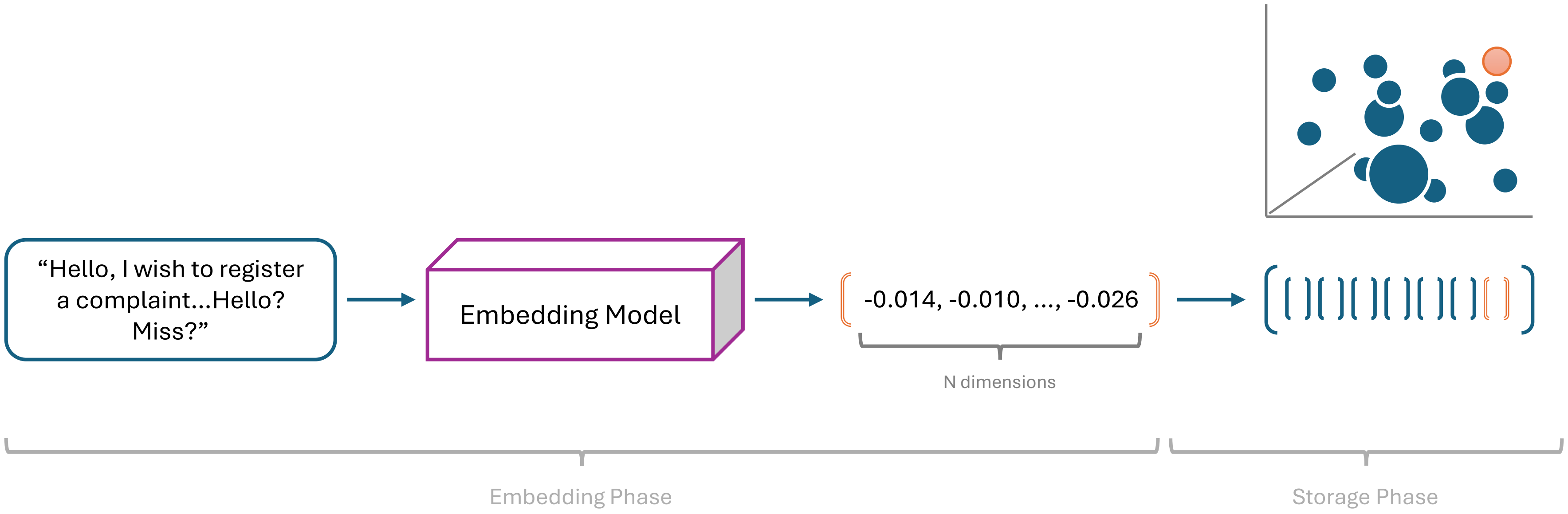
Classic RAG uses *embeddings*

Embeddings require a (fast) roundtrip to a service maintained by an LLM provider

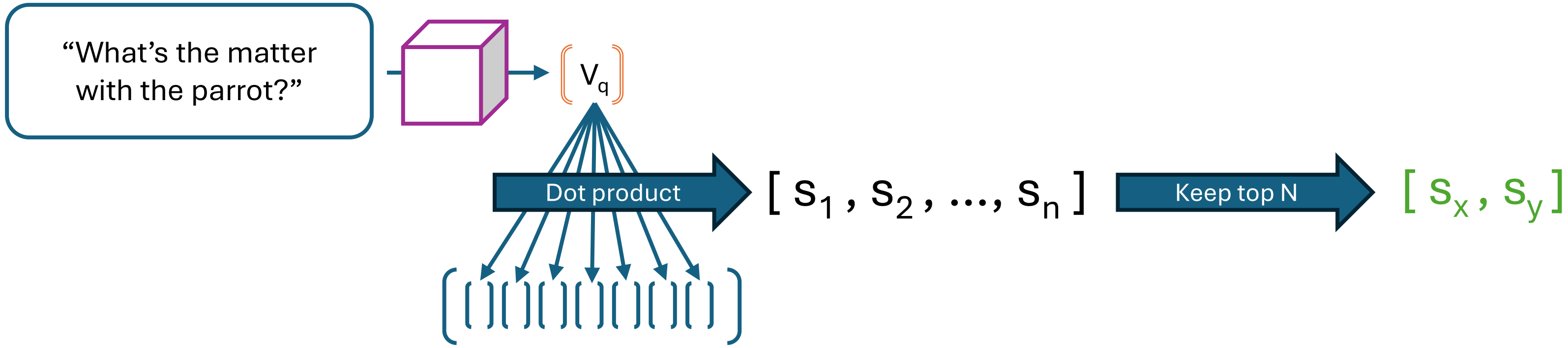
For best results:

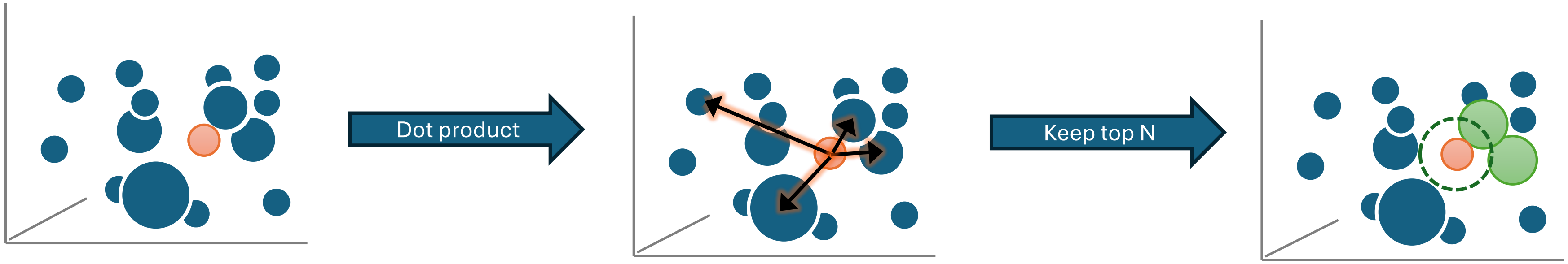
- Chop long documents up in chunks
- Use batching

Embedding Storage



Embedding Retrieval





Classic RAG:

Advantages and Downsides

Pros

- Well understood (mostly).
- Easy to deploy.

Cons

- Expensive and complicated.
- Take up lots of space.
- Large input strings give “mushy” results, near many things, not all that near anything in your query.

Structured RAG explained:

1. Ingestion

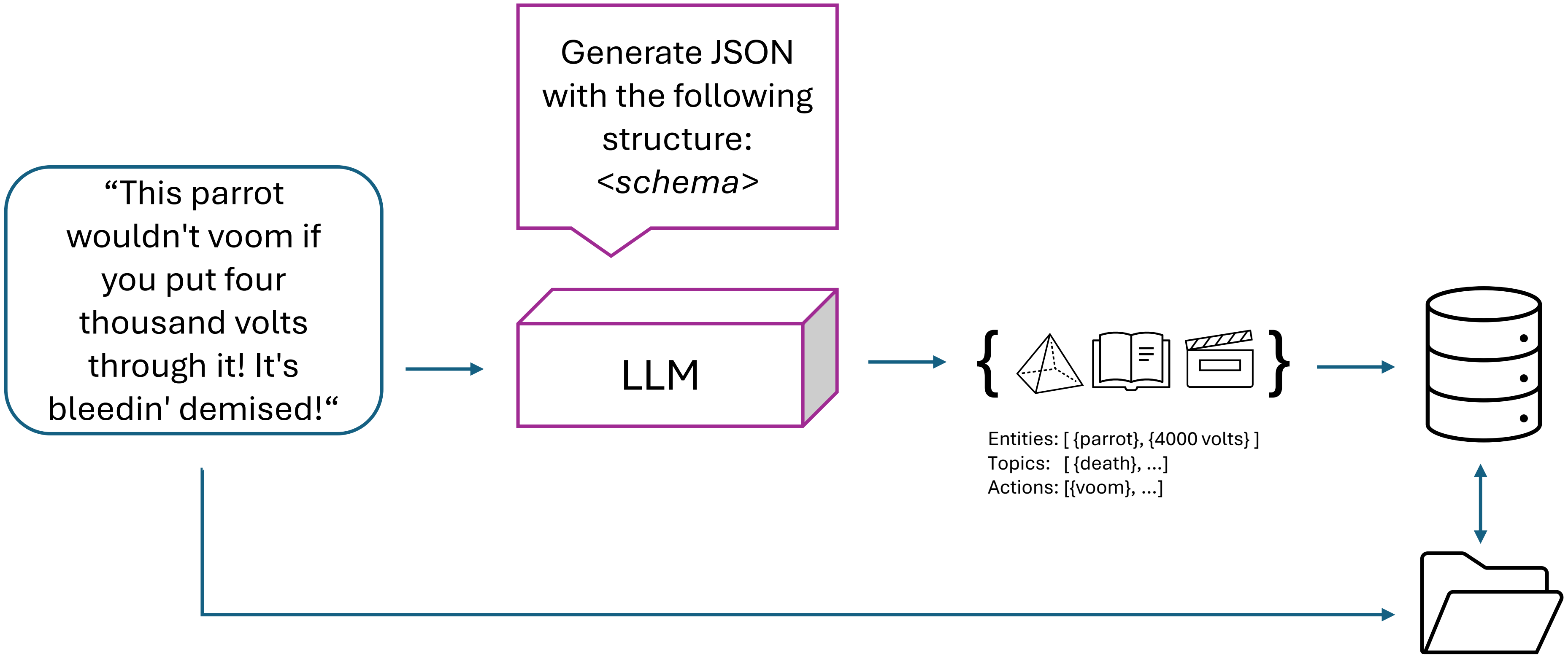
Run each input string through an LLM that extracts “*knowledge*” from it.

Knowledge: entities (people, places, etc.), actions, topics, relationships, etc.

Store knowledge nuggets in a classic database and create indexes over them.

The database is easily queried – it’s classic computer science.

Structured RAG – Ingestion Pipeline



Structured RAG explained:

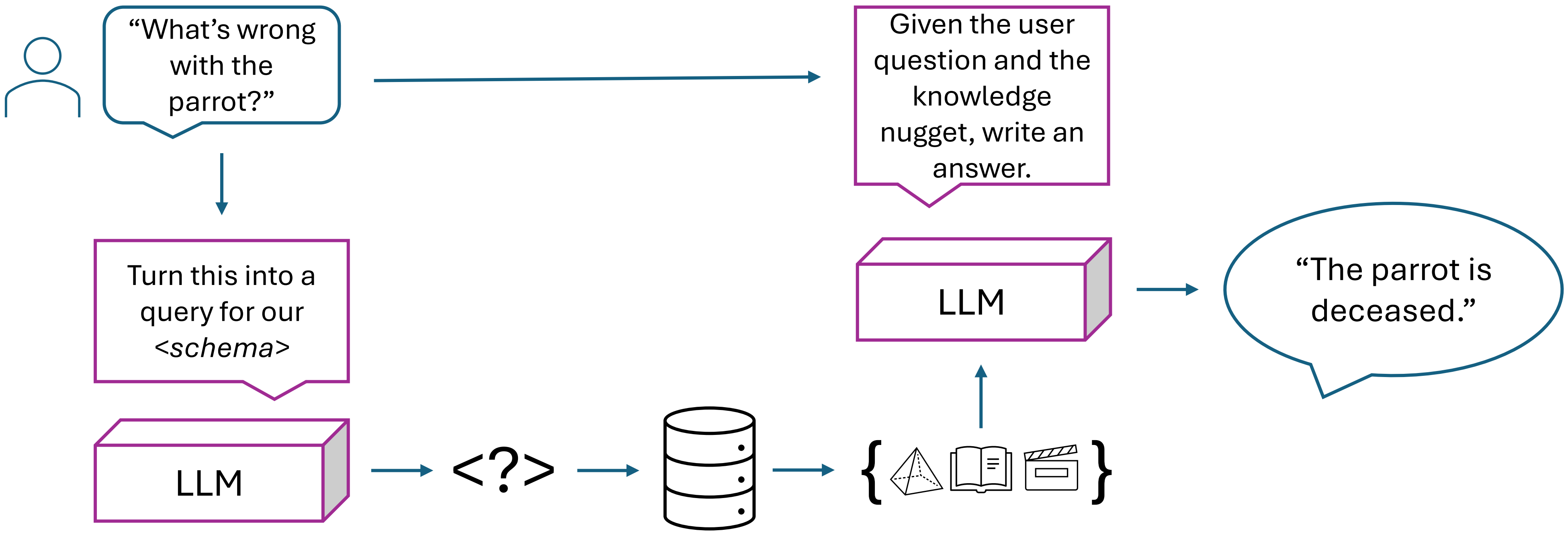
2. Querying

Turn user question into an (abstracted) database query.

Run the query, producing {Entities, Topics, Actions}.

Produce answer from best query results.

Structured RAG – Query Pipeline



Structured RAG evaluated

- Retrieval recall and precision are better due to extraction and storage of knowledge nuggets, which can be queried more precisely, generating higher information density, leading to better answers.
- Scalable to much larger conversation histories: indexing approach needs less compute and data.
- Can do inference over knowledge nuggets. E.g. *artist(Palin)* → *person(Palin),* which helps with e.g. “what people did we talk about yesterday?”
- Some knowledge can be extracted without consulting an LLM (e.g. email headers).
- Bulk ingestion is slower, due to the use of an LLM instead of just embeddings.

Demo: Transcripts

Demo: Messages

Installation
(Python 3.12,13,14)

```
$ pip install typeagent                    # version 0.3.0
$ export OPENAI_API_KEY=xxxxxxxx
# Or put env vars in .env
```

Ingest messages

```
from  typeagent import create_conversation

from typeagent.transcripts.transcript import (
        TranscriptMessage, TranscriptMessageMeta)

conv = await create_conversation(
                "mymemory.db", TranscriptMessage)

msgs = [TranscriptMessage(
             text_chunks=[chunk],
             metadata=
               TranscriptMessageMeta(speaker=speaker))
         for chunk, speaker in .....]  # You have to code the '.....'

await conv. add_messages_with_indexing(msgs)
```

Query database
(tentative API)

```
from  typeagent import create_conversation

conv = await create_conversation("mymemory.db",
                                 TranscriptMessage)

question = input(“typeagent> ")
answer = await conv.query(question)
print(answer)
```

Resources

Repo:
https://github.com/microsoft/typeagent-py

Docs:
https://github.com/microsoft/typeagent-py/tree/main/docs/README.md

PyPI:
https://pypi.org/project/typeagent

Papers:
To be written

Thank you!