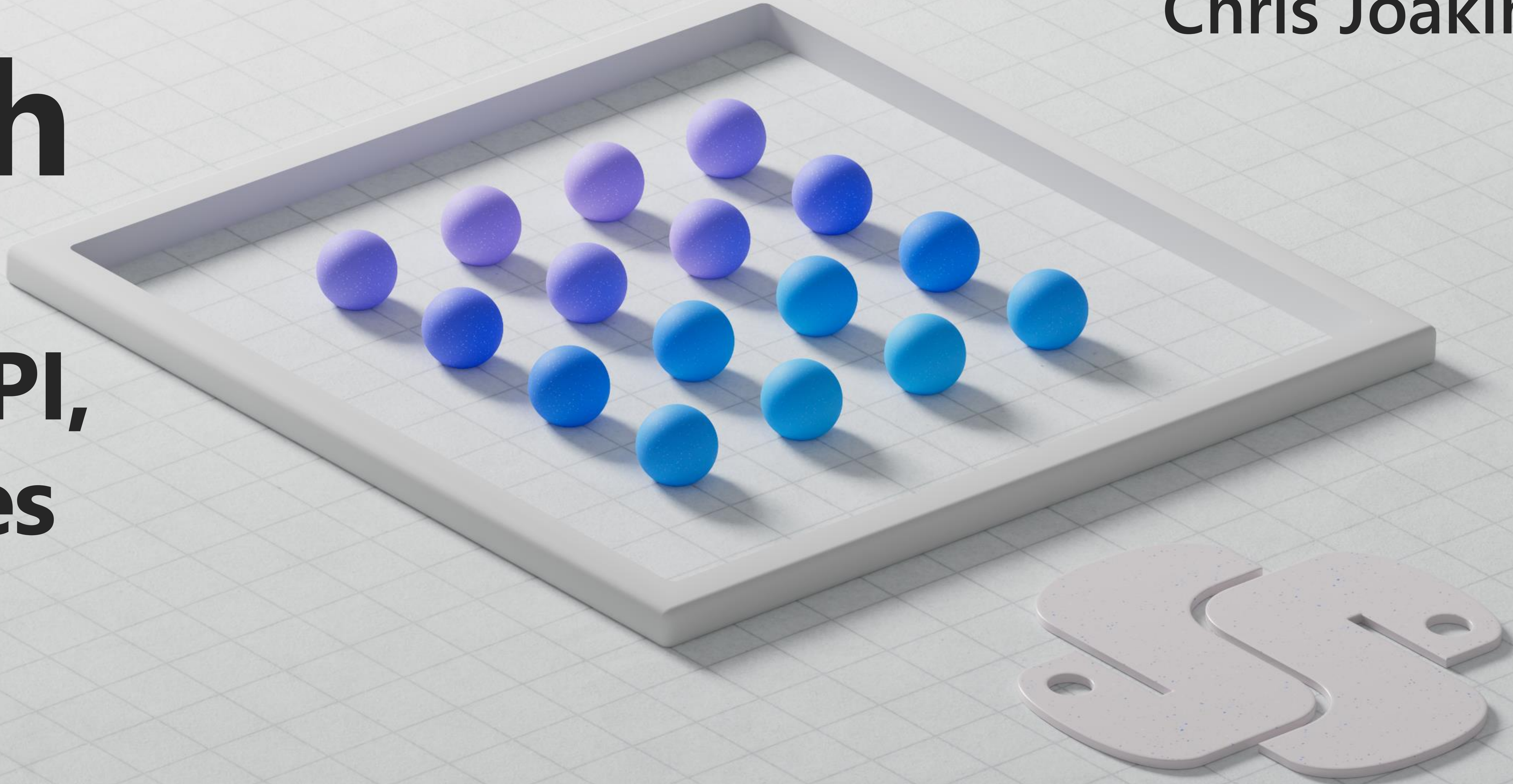
Microsoft

Chris Joakim

Azure Developer

Vector Search with OpenAI, Cosmos DB NoSQL API, and Cognitive Services



Python Day

Welcome

Presentation Outline

This Presentation

Vector Search in Microsoft Azure with:

- Azure OpenAI
- Azure Cosmos DB NoSQL API
- Azure Cognitive Search
- The Sean Lahman Baseball Database (CSV files)
- Implemented with Python 3

Description

Outline:

- Concepts - Vectors, Vectorization, Vector Search
- The Business Use-Case
- The Implementation

Concepts

Concepts

What is a Vector?

- **A Vector is a one-dimensional array of scalar values**.
- Think of them as a **numpy array of floats**

Description

What is Vectorization?

- **Vectorization is the process of converting text data into vectors**
- These vectors are called embeddings in OpenAI
- The OpenAI SDK contains the functionality to produce a vector, or an **embedding**, from text data
- OpenAI's text **embeddings measure the relatedness of text strings**
- Embeddings are information-dense and are computationally efficient (but look verbose when printed)
- They use Cosine similarity to measure semantic similarity

- https://learn.microsoft.com/en-us/azure/ai-services/openai/
- https://platform.openai.com/docs/libraries
- https://platform.openai.com/docs/guides/embeddings/what-are-embeddings

[-0
[-0.028514496982097626, 0.02490937151014805, -0.006417802534997463, ... -0.01409399788826704, -0.026895591989159584, -0.007012988440692425]

Concepts

What is Vector Search?

- Searching a database, or search-engine, using vectors
- A vector is passed in as the **query criteria**
- The DB/engine matches rows/documents **based on the given vector column/attribute in the DB**

Description

What does a Vector Look Like? (an array of 1536 floats in the range -1 to +1)

```
[
    -0.028514496982097626,
    0.02490937151014805,
    -0.006417802534997463,
    ...
    -0.01409399788826704,
    -0.026895591989159584,
    -0.007012988440692425
  ]
```

It looks very large and verbose when printed, but it's actually a **very efficient data structure** and it enables greater computational efficiency vs text data

```
[ -0
  [ -0.028514496982097626, 0.02490937151014805, -0.006417802534997463, ... -0.01409399788826704, -0.026895591989159584, -0.007012988440692425 ]
```

Concepts

What can I do with Vector Search?

- Find **images** that are similar to a given image based on their visual content and style
- Find **documents** that are similar to a given document based on their topic and sentiment
- Find products that are similar to a given product based on their **features** and ratings
- In short, many use-cases. Including finding similar Baseball Players
- But, IMO, it doesn't replace standard search. It augments it

Wait, why are we using Azure OpenAI here again?

- To generate the vectors (i.e. – embeddings) from your text data input
- **OpenAI contains models like GPT-4 and DALL-E**
- But it also contains models like **text-embedding-ada-002**

- https://platform.openai.com/docs/models

The Business Use-Case

The Business Use-Case

This project is focused on Baseball data and matching Baseball Players

- But the solution is applicable to many vector search use-cases

Description

What's the Business Problem we're trying to solve?

While other search techniques can answer **simple searches** like:

- Who hits home runs at a similar rate as Hank Aaron?
- Who steals bases at a similar rate as Rickey Henderson?
- Who has a similar pitching ERA (earned run average) as Ron Guidry?

This project instead seeks to answer this more complex question, using vector search:

- **Who has a similar OVERALL PERFORMANCE PROFILE as player x?**

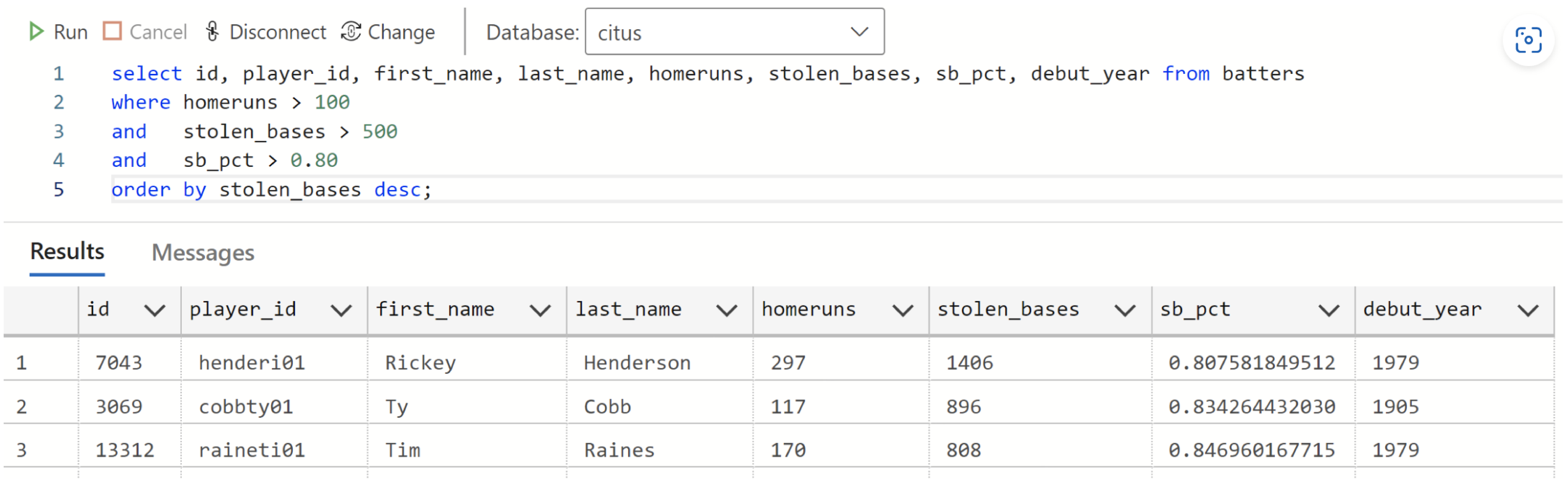
This type of search is more nuanced and subtle, but **can yield more relevant search results**.

The Business Use-Case – Find Similar Baseball Players

- Meet **Rickey Henderson**, Hall of Fame Player, **Statistical Unicorn** - rare combination of speed and power
- All-time MLB leader in stolen bases, by far
- Also very high, statistically, in home runs, triples, walks, IBB, HBP, and runs

The Business Use-Case – Search (ex: Who could possibly be like Rickey Henderson?)

- **You can try to use a simplistic query** (SQL is shown here) to identify similar players
- But this WHERE clause only contains **only three attributes** ... it's not a "**full-spectrum**" query of the many dimensions of baseball player statistics
- And ***YOU*** have to construct the query
- **With vector search you simply ask: Who is like Rickey Henderson? And you get pertinent results.**

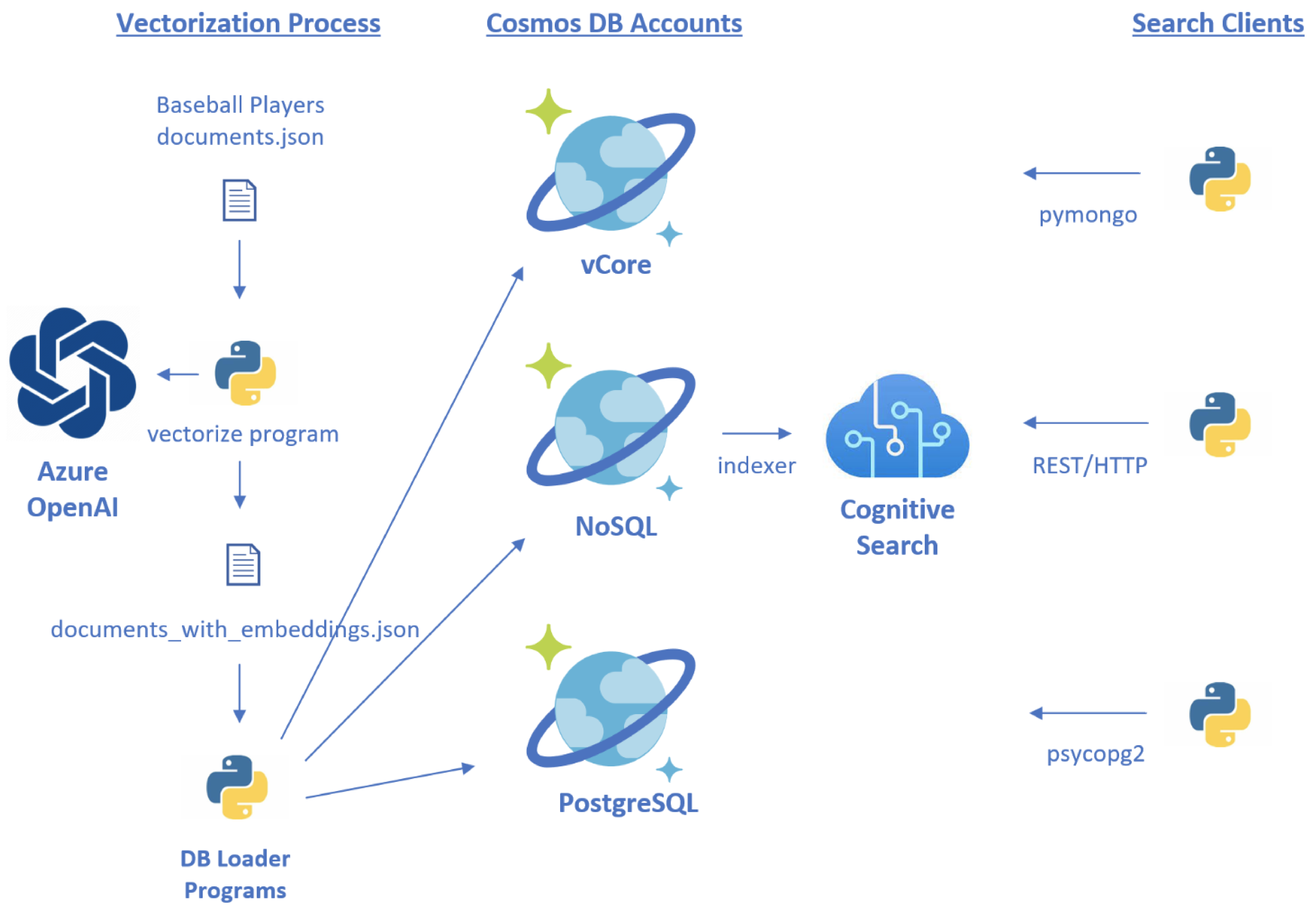


```sql
select id, player_id, first_name, last_name, homeruns, stolen_bases, sb_pct, debut_year from batters
where homeruns > 100
and   stolen_bases > 500
and   sb_pct > 0.80
order by stolen_bases desc;
```

	id	player_id	first_name	last_name	homeruns	stolen_bases	sb_pct	debut_year
1	7043	henderi01	Rickey	Henderson	297	1406	0.807581849512	1979
2	3069	cobbty01	Ty	Cobb	117	896	0.834264432030	1905
3	13312	raineti01	Tim	Raines	170	808	0.846960167715	1979

The Implementation

Implementation – In this Presentation we'll focus on the NoSQL & Cognitive Search Path



Implementation – Baseball Player Document, Part 1/4, Mostly from **People.csv** fields

```
{
    "playerID": "henderi01",
    "birthYear": 1958,
    "birthCountry": "USA",
    "deathYear": "",
    "nameFirst": "Rickey",
    "nameLast": "Henderson",
    "weight": 180,
    "height": 70,
    "bats": "R",
    "throws": "L",
    "debut": "1979-06-24",
    "finalGame": "2003-09-19",
    "teams": {
        "total_games": 3081,
        "teams": {
            "OAK": 1704,
            "NYA": 596,   ...
         },
        "primary_team": "OAK"
    },
    "primary_position": "LF",
...
```

Description

Implementation – Baseball Player Document, Part 2/4, merged in **Batting.csv** fields

```
...
    "batting": {
        "G": 3081,
         "AB": 10961,
         "R": 2295,
         "H": 3055,
         "2B": 510,
         "3B": 66,
         "HR": 297,
         "RBI": 1115,
         "SB": 1406,
         "CS": 335,
         "BB": 2190,
         "SO": 1694,
         "IBB": 61,
         "HBP": 98,
         "SF": 67,
      ...
```

Description

Implementation – Baseball Player Document, Part 3/4, calculated fields

```
...
"batting": {
    "calculated": {
        "runs_per_ab": 0.20937870632241584,
        "batting_avg": 0.2787154456710154,
        "2b_avg": 0.046528601404981294,
        "3b_avg": 0.006021348417115227,
        "hr_avg": 0.027096067877018522,
        "rbi_avg": 0.10172429522853754,
        "bb_avg": 0.19979928838609615,
        "so_avg": 0.15454794270595748,
        "ibb_avg": 0.005565185658242861,
        "hbp_avg": 0.008940790073898367,
        "sb_pct": 0.8075818495117748
    }
},
...
```

Description

These are starting to look like "**Machine Learning Features**". **But is this the correct approach?**

Implementation – Baseball Player Document, Part 4/4, embeddings fields

```
    ....
    "category": "fielder",
    "debut_year": 1979,
    "final_year": 2003,

    "embeddings_str": "fielder primary_position_lf total_games_3081 bats_r throws_l hits_3055 hr_297 batting_avg_279
runs_per_ab_209 2b_avg_47 3b_avg_6 hr_avg_27 rbi_avg_102 bb_avg_200 so_avg_155 ibb_avg_6 hbp_avg_9 sb_1406 sb_pct_81",

    "embeddings": [
      -0.028514496982097626,
      0.02490937151014805,
      ...
      -0.026895591989159584,
      -0.007012988440692425
    ]
  }
```

Description

The “embeddings” value is produced by a call to OpenAI. **But what is this “embeddings_str” attribute? …**

Implementation – embeddings_str and "binned text" values

- Since we pass **string/text** values to the **OpenAI SDK** to produce embeddings
- Rather than use **"normalized numeric features"** as in **Machine Learning**
- The approach taken here is to create **a series of "binned text" values**, and create an **embeddings_str** value from these to pass to OpenAI. This is intended to produce the most precise and pertinent text input to the OpenAI model

- For example, a batting average of **0.2787154456710154** becomes **"batting_avg_279"**
- And a complete **embeddings_str** value looks like this one logical string:

"fielder primary_position_lf total_games_3081 bats_r throws_l
hits_3055 hr_297 batting_avg_279 runs_per_ab_209 2b_avg_47 3b_avg_6
hr_avg_27 rbi_avg_102 bb_avg_200 so_avg_155 ibb_avg_6 hbp_avg_9
sb_1406 sb_pct_81"

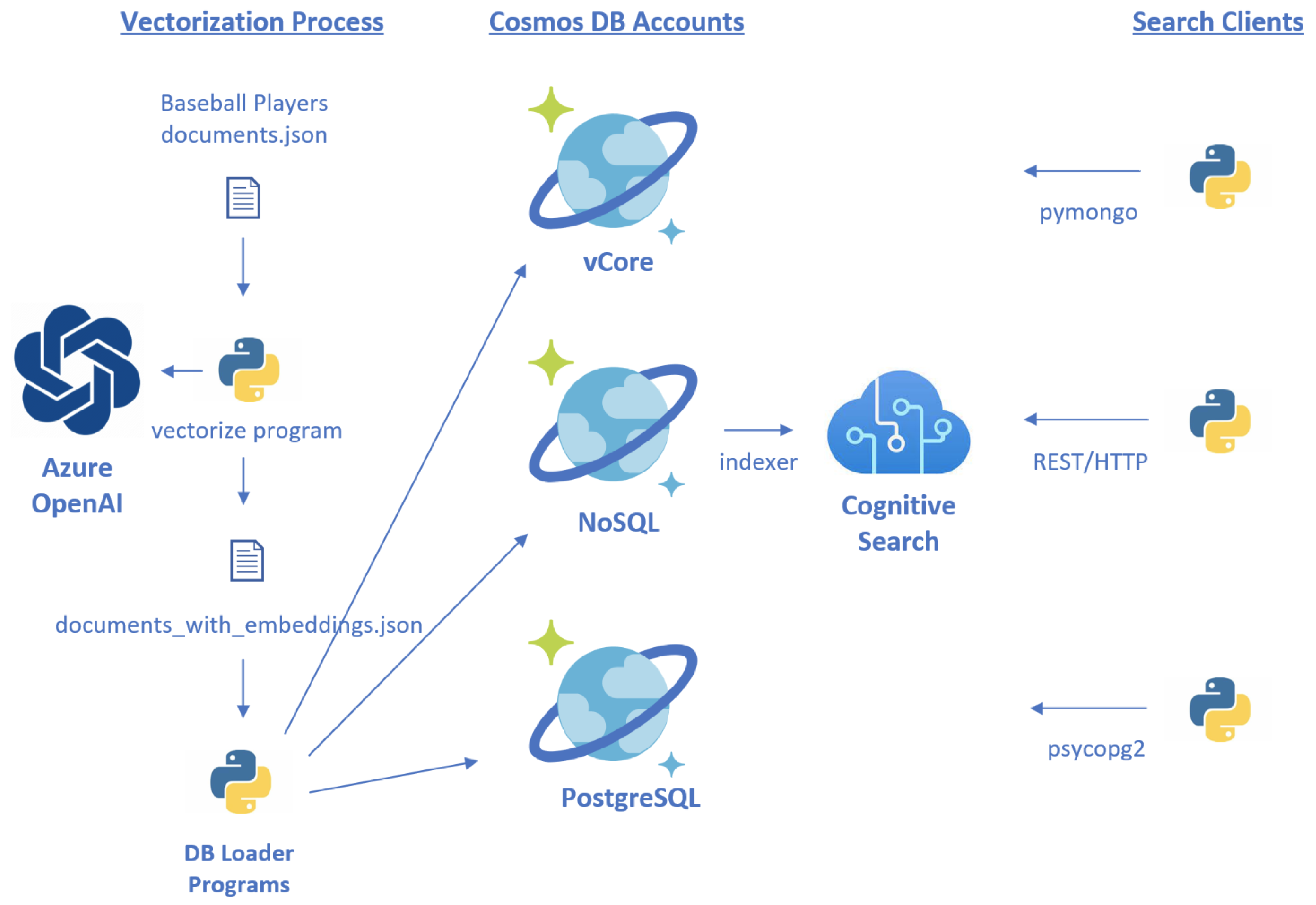
- Is this the new art of "***Text Engineering***", much like "Feature Engineering" in ML and "Prompt Engineering" with Chat-GPT?

Implementation – Using the OpenAI SDK to get embedding values

```
from openai.embeddings_utils import get_embedding    # after ‘pip install openai’
from openai.openai_object import OpenAIObject

# Configure the openai client library
openai.api_base     = opts['url']    # <-- value from an environment variable, Azure Key Vault, etc
openai.api_key      = opts['key']   # <-- value from an environment variable, Azure Key Vault, etc
openai.api_type     = 'azure'
openai.api_version = '2023-05-15’   # '2022-06-01-preview' '2023-05-15'

# Ask the OpenAI SDK to calculate and return the embedding value
# The OpenAI embedding model used here is 'text-embedding-ada-002'
e = openai.Embedding.create(input=[embeddings_str], engine='text-embedding-ada-002')

return e['data'][0]['embedding']  # returns a list of 1536 floats
```

Now, let’s revisit the System Design…

Implementation – System Design, Revisited



We've talked about Data Wrangling, and Vectorization. Now let's **configure Cognitive Search**, and **load Cosmos DB**…

Implementation – Configure Azure Cognitive Search with the REST API

- In the repo, **see the cognitive_search directory**
- Azure Cognitive Search publishes a beautiful, simple, easy-to-use **REST API**
- Easy to invoke with the Python '**requests**' library for HTTP
- Create Three Primary Objects in Cognitive Search:
  - **Datasource** - defines where the source data is (i.e. - the Cosmos DB account, database, and container)
  - **Index** - defines what document attributes to index and make searchable
  - **Indexer** - associates a Datasource to an Index, with a schedule
  - All of these are defined with simple JSON schemas
  - https://learn.microsoft.com/en-us/rest/api/searchservice/

Wait, why is it called "Azure Cognitive Search" and not just Azure Search?

- It offers AI Enrichment using Azure Cognitive Services
- Use a document processing "**pipeline**" of built-in and custom **skills** (i.e. – functionality)
- For example, "**crack**" a PDF document, extract the images, identify the images, extract the image text, translate it all into English, identify the sentiment and top words, etc, etc.   Make it all searchable.

Implementation – Configure Azure Cognitive Search - Index Schema 1/2

```
{
  "name": "baseballplayers",
  "fields": [
    {
      "name": "id",
      "key": "true",
      "type": "Edm.String",
      "searchable": "true",
      "filterable": "true",
      "sortable": "true",
      "facetable": "true"
    },
    {
      "name": "playerID",
      "key": "false",
      "type": "Edm.String",
      "searchable": "true",
      "filterable": "true",
      "sortable": "true",
      "facetable": "true"
    },
   ....
```

Nothing unusual here; this is a standard index definition so far...

Implementation – Configure Azure Cognitive Search - Index Schema 2/2

```
   …
    {
      "name": "embeddings",
      "type": "Collection(Edm.Single)",
      "searchable": true,
      "retrievable": true,
      "dimensions": 1536,
      "vectorSearchConfiguration": "vectorConfig"
    }
  ],
  "vectorSearch": {
    "algorithmConfigurations": [
        {
            "name": "vectorConfig",
            "kind": "hnsw"
        }
    ]
  }
}
```

This is the new functionality to define a Vector Index.

Implementation – Loading Cosmos DB NoSQL API with the vectorized data

- In the repo, **see the cosmos_pg directory**
- Single Cosmos DB container, partition key is playerID, 4000 RU autoscale
- **pip install azure-cosmos**   The SDK for the NoSQL API

```
from azure.cosmos import cosmos_client, diagnostics, exceptions    <-- Cosmos DB SDK classes
# Connect to Cosmos DB, select database, select container, iterate and load documents
opts = dict()
opts['url'] = Env.var('AZURE_COSMOSDB_NOSQL_URI')
opts['key'] = Env.var('AZURE_COSMOSDB_NOSQL_RW_KEY1')
c = Cosmos(opts)  # My wrapper class for cosmos_client and functionality
c.set_db('dev')
c.set_container('baseballplayers’)
for idx, pid in enumerate(player_ids):   <--- the documents_with_embeddings.json file is read and iterated
    doc = documents[pid]
    doc['id'] = str(uuid.uuid4())           <--- create a unique document id
    result = c.upsert_doc(doc)              <--- insert/update the doc, which includes the embeddings attribute
```

After we load Cosmos DB, **the documents will be indexed by the Indexer in Azure Cognitive Search**

Implementation – Execute a Search for Players like Rickey Henderson

- First, we do a direct lookup of henderi01 to get his embedding value.
    - Alternative workflow for some apps: Calculate the embedding from User or other input
- Then we use that embedding to execute a vector search with this request, with up to 10 results (k)
    - HTTP POST to the search endpoint

```
{
  "count": "true",
  "select": "id,playerID,nameFirst,nameLast,primary_position",
  "orderby": "playerID",
  "vectors": [
    {
      "value": [
        -0.028514497,
        0.024909372,
        ...
        -0.026895592,
        -0.0070129884
      ],
      "fields": "embeddings",
      "k": 10
    }
  ]
}
```

Implementation – The Search Results

Both Power Hitters (Barry Bonds) and great Base Stealers (Lou Brock) are in the pertinent result set that matches for Rickey Henderson. Like Rickey, they are Left Fielders.

```
{
  "@odata.context": "https://gbbcjsearch.search.windows.net/indexes('baseballplayers')/$metadata#docs(*)",
  "@odata.count": 10,
  "value": [
    {
      "@search.score": 1.0,
      "id": "e4cc38fd-18c8-4418-8841-98a1403f5ef1",
      "playerID": "bondsba01",
      "nameFirst": "Barry",
      "nameLast": "Bonds",
      "primary_position": "LF"
    },
    {
      "@search.score": 1.0,
      "id": "c9867b7b-8c34-4e95-a672-16f1bdb393cf",
      "playerID": "brocklo01",
      "nameFirst": "Lou",
      "nameLast": "Brock",
      "primary_position": "LF"
    },
```

- …

Summary

- **Vectors are computationally efficient text-matching data structures**
- Vector searching offers pertinent **"full spectrum" searches** with no parameters
- Use other search syntaxes for simple searches (i.e. – stolen_bases > 1000 and throws = 'L')
- Emerging art of ***"Text Engineering"*** for Vector Search vs "Feature Engineering" in Machine Learning
- You can implement Vector Search with these three Azure Cosmos DB Solutions:
  - **Cosmos DB vCore Mongo API** (native vector search)
  - **Cosmos DB PostgreSQL API** (native vector search with pgvector extension)
  - **Cosmos DB NoSQL API with Azure Cognitive Search** (this presentation)

- All three solutions are implemented with Python in this GitHub repo:
  - https://github.com/cjoakim/azure-cosmos-db-vector-search-openai-python

- I encourage you to **learn Python!**
  - Suitable for so many use-cases – Data Science, Spark, AI, data wrangling, etc.
  - Low learning curve, and widely used

Questions?

Thank you!