Member-only story

Turbocharging Denodo AI SDK: How Semantic Caching Makes Text-to-SQL 9X Faster

How Natural Language Processing and AI Frameworks Slash Query Times for Enterprise Data

MKWriteshere · Follow

Published in Data Science Collective · 12 min read · Mar 11, 2025

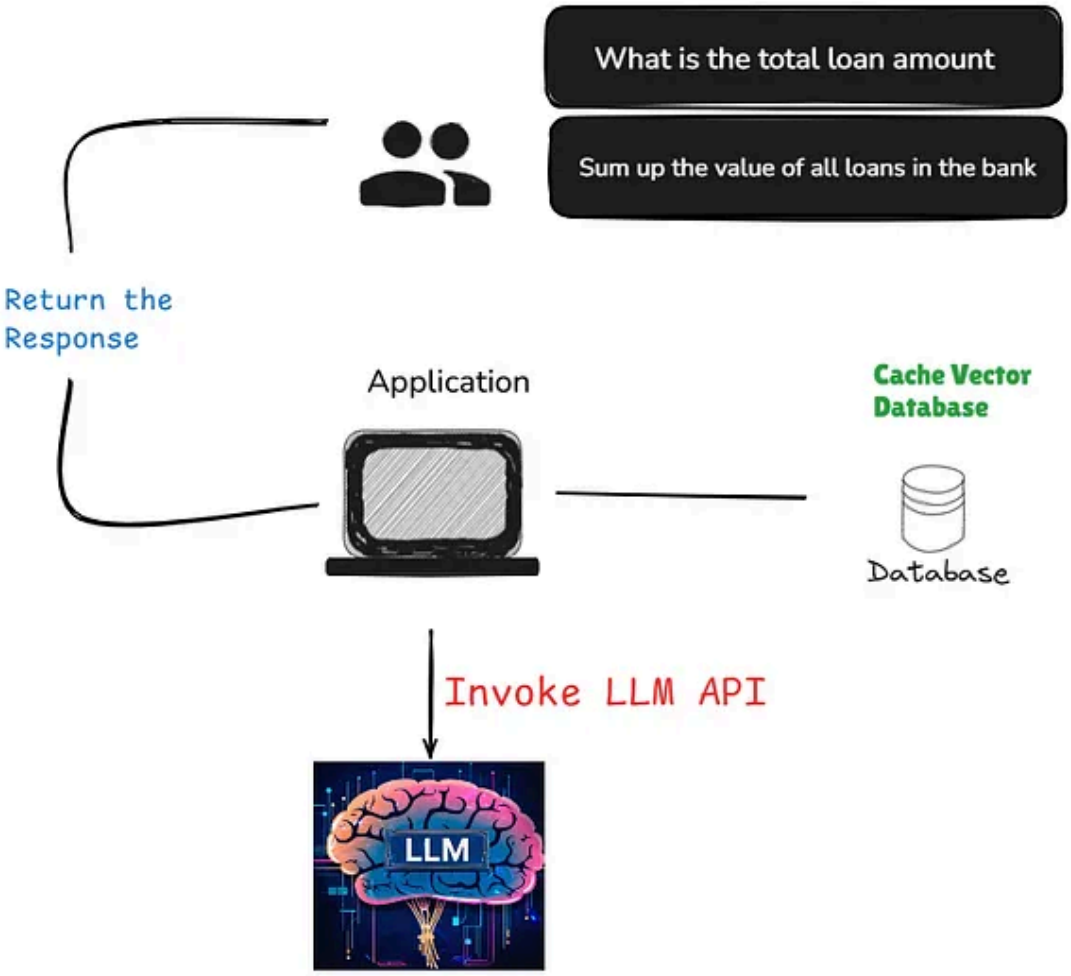


Image by Author

If you are not a member of the Medium, please **CLICK HERE** to read

Text-to-SQL transformed how enterprises interact with their data. However, for similar questions, regenerating SQL from scratch isn't always the most efficient approach.

When LLMs start translating natural language into database queries, it marked a significant advancement in data democratization. Non-technical users could finally ask complex questions of their data without learning SQL.

Among the solutions in this space, Denodo AI SDK stands out by connecting and combining enterprise data silos through a logical data layer, providing secure, governed access to structured data through natural language.

With remarkable accuracy, Denodo's text-to-SQL functionality translates questions into Denodo's Virtual Query Language (VQL). While this process produces excellent results, it presents an optimization opportunity,

> Why regenerate SQL from scratch when users ask similar questions with minor variations?

By implementing a semantic caching layer on top of Denodo AI SDK, we've seen dramatic performance improvements, making responses 9x faster while maintaining all of Denodo's powerful governance and security features. This enhancement makes Denodo's already robust platform even more responsive for users asking variations of common questions.

In this article, I'll show how semantic caching can turbocharge Denodo AI SDK, complete with code examples and performance metrics you can implement in your environment.

What is Semantic Caching?

Semantic caching stores the meaning behind queries, retrieving information based on intent rather than exact matches. This provides more relevant results than traditional caching while being faster than direct LLM responses.

Like a perceptive librarian who understands the context of requests rather than just matching titles, semantic caching intelligently retrieves data that

best aligns with the user's actual needs.

Key Components

1. **Embedding model:** Creates vector representations of data to measure query similarity.
2. **Vector database:** Stores embeddings for fast retrieval based on semantic similarity rather than exact matches.
3. **Cache:** Central storage for responses and their semantic meaning.
4. **Vector search:** Quickly evaluates similarity between incoming queries and cached data to determine the best response.

Benefits Over Traditional Caching : Traditional caching speeds up access to frequently requested information but disregards query meaning. Semantic caching adds an intelligent layer that understands query intent, storing and retrieving only the most relevant data. By using AI embedding models to capture meaning, semantic caching delivers faster, more relevant results while reducing processing overhead and improving efficiency.

The Challenge: Denodo AI SDK and the Cost of Natural Language

Denodo's data virtualization platform excels at solving trusted data challenges by connecting and combining data silos through a logical data layer.

Denodo's AI feature — Denodo AI SDK offers functionality which helps AI Developers to build an llm application which translates natural language questions into Denodo's Virtual Query Language (VQL) with remarkable accuracy that retrieves the Enterprise data, also it is remarkably easy to

integrate structured data into text-to-SQL LLM applications with robust governance and security.

However, this powerful capability comes with a cost challenge. Every time a user asks a question, the standard Denodo AI SDK workflow involves:

1. Vector search to find relevant tables and columns
2. Analysis of database schema and relationships
3. Understanding of filters and aggregations
4. Generation of valid VQL (Denodo SQL)
5. Execution of the query and formatting results

For organizations handling hundreds of queries daily through Denodo AI SDK, each consuming thousands of tokens in a large language model, this quickly becomes unsustainable from both a cost and performance perspective.

How could we preserve Denodo's excellent data governance while making the text-to-SQL conversion more efficient?

Spotting the Opportunity — Why Redo What's Already Done?

The key insight is that most database questions follow patterns with minor variations. When analysts ask "**How many customers in California?**" followed by "**Count clients in New York,**" they're asking the same question with one parameter changed. We can enhance performance for these similar queries by implementing a semantic caching pattern:

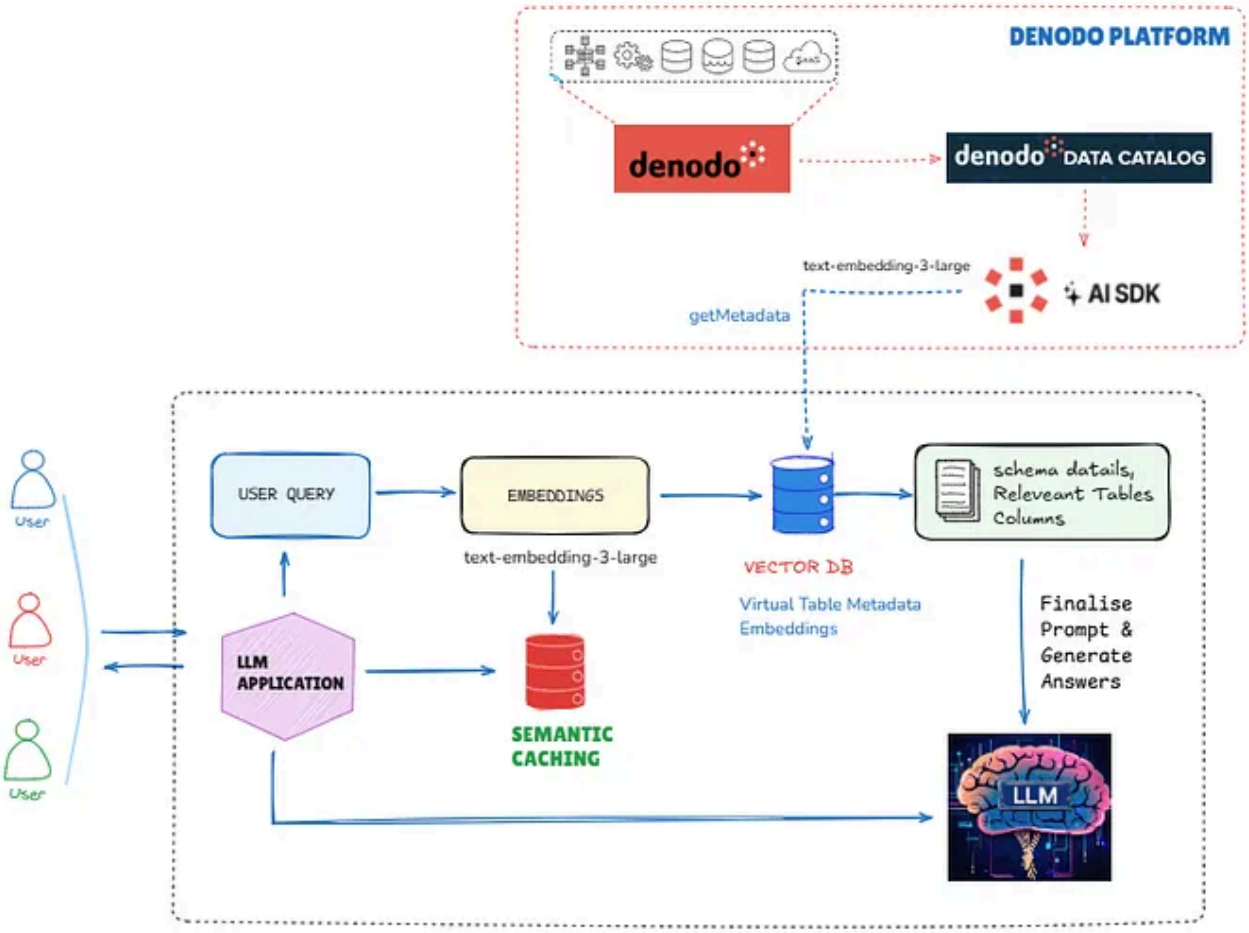


Image by Author

1. **Vector-based Cache Database** — Store natural language questions, their SQL queries, and query explanations in a vector database that enables semantic search
2. **Match similar questions** — When a new question arrives, use vector embeddings to find semantically similar previously asked questions
3. **Validate semantic relationship** — Use a small language model or cheaper models to verify if the questions truly have the same intent but with different parameters
4. **Modify the existing SQL query** — Instead of generating a completely new SQL, adapt the cached query by changing only the necessary parts

5. **Execute and validate** — Run the modified query against the denodo's data catalog to retrieve real-time results with secured layer.

This approach preserves all the benefits of Denodo's trusted data platform while making it significantly more responsive and cost-effective for common query patterns.

Step 1: Designing the Solution — A Smart Two-Path Workflow

The trick is to blend Denodo's full power for new questions with a fast lane for similar ones. My solution uses a semantic cache with two paths:

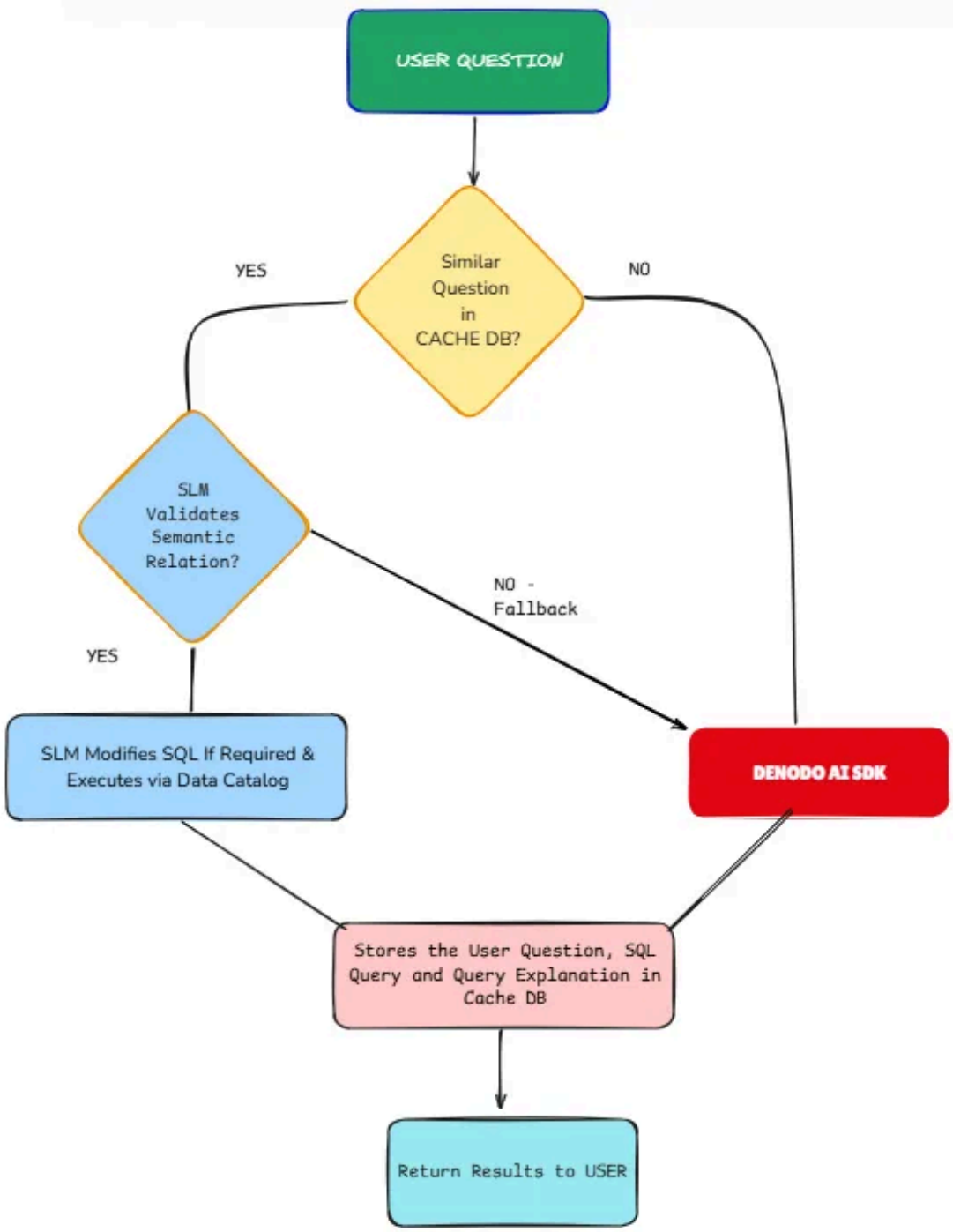


Image by Author

✅ **Cache Check:** When someone asks “Count the number of clients in NewYork,” we check our cache/vector database (FAISS)

✅ **Validation Layer:**

- Vector similarity identifies candidate matches like “How many customers do we have in the state of CA?”
- A small language model validates semantic relatedness
- This prevents false positives while maintaining high accuracy

✅ **Execution Paths:**

- **Cache Hit:** For repeated/similar questions, modify the existing SQL (changing “CA” to “NY”) if required and execute the SQL.
- **Cache Miss:** For actual questions, use Denodo AI SDK’s standard pipeline (fallback mechanism)

This complementary approach means we get the best of both worlds, Denodo’s powerful data virtualization for novel questions and lightning-fast responses for similar ones.

Step 2:Building the Core — Semantic Cache in Python

Let’s get hands-on. The system hinges on a SemanticCache class that pairs vector embeddings with a FAISS index for lightning-fast similarity checks. Here’s how it starts:

```
class SemanticCache:
    def __init__(self, embedding_model, similarity_threshold=0.90):
        self.embedding_model = embedding_model
        self.similarity_threshold = similarity_threshold
        self.cache = {"questions": [], "embeddings": None, "sql_queries": [], "r
        self.faiss_index = faiss.IndexFlatL2(3072)  # 3072 dims for text-embeddi
```

- **Embedding Model:** I used OpenAI's text-embedding-3-large to convert questions into vectors.
- **FAISS Index:** Stores embeddings for fast lookups.
- **Threshold:** 0.90 ensures only truly similar questions match.

When a question comes in, **find_similar_question** checks the cache:

```
def find_similar_question(self, question):
    if self.faiss_index.ntotal == 0:
        return None, None, None, 0.0
    query_vector = np.array([self.embedding_model.embed_query(question)], dtype=
    distances, indices = self.faiss_index.search(query_vector, 1)
    similarity = 1 - (distances[0][0] / 20)  # Normalize distance to similarity
    if similarity >= self.similarity_threshold:
        index = indices[0][0]
        return self.cache["questions"][index], self.cache["sql_queries"][index],
    return None, None, None, 0.0
```

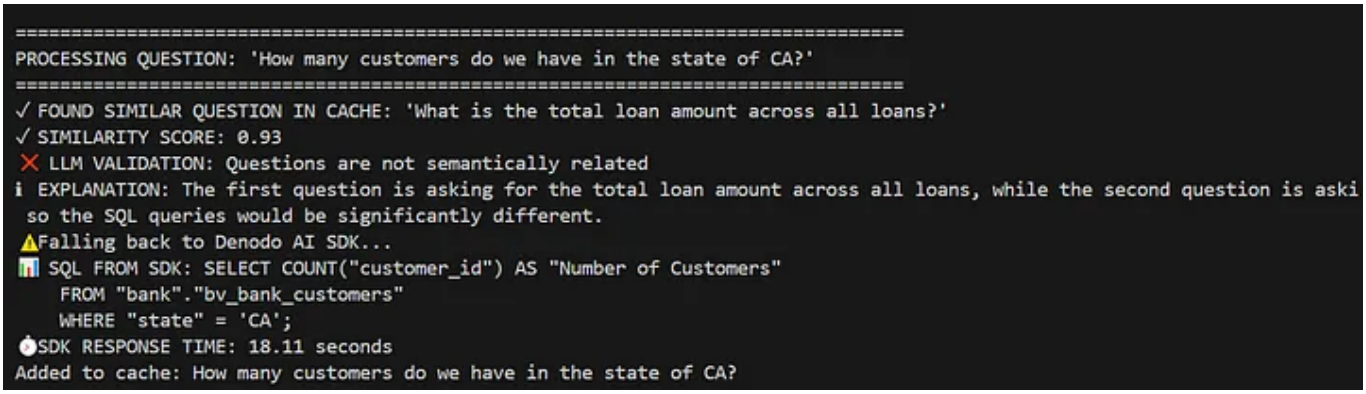
Step 3: Validating Similarity — The LLM Gatekeeper

Vector similarity alone isn't foolproof. "List top 5 loans" and "List top 5 customers" might look close in vector space but need different SQL.

In the below example, the system initially found a high similarity score (0.93) between the question *'How many customers do we have in the state of CA?'* and the cached question *'What is the total loan amount across all loans?'*.

However, despite the high similarity score, the two questions are semantically different the first asks for a count of customers in a specific state, while the second asks for a total loan amount. This mismatch occurred because the cache contained very limited data, causing the similarity

algorithm to overestimate relevance. To mitigate this, i used an SLM for semantic validation, which correctly identified the questions as unrelated. As a result, the system fell back to the Denodo AI SDK to generate the correct SQL query, ensuring accurate results



Code Snippet :

```
def are_questions_semantically_related(question1, question2):
    """
    Use a small, cost-effective language model to determine if two questions
    are semantically related enough to potentially reuse and modify the SQL.
    This function uses GPT-3.5 Turbo, but could easily be replaced with
    open-source models like Llama 3 or Mistral for even greater cost savings.
    """
    llm = ChatOpenAI(api_key=OPENAI_API_KEY, model="gpt-3.5-turbo")

    prompt = ChatPromptTemplate.from_template("""
    I need to determine if two questions about a banking database are semantical
    that the SQL query for one could be modified to answer the other.

    Question 1: {question1}
    Question 2: {question2}

    First, analyze what each question is asking for:
    - What entity/table is each question about?
    - What operation is being performed?
    - What filters or conditions are applied?

    Then determine if they are related enough that one SQL could be modified to
```

```
        Output your decision as a JSON object with these fields:
        {{
          "are_related": true/false,
          "explanation": "Brief explanation of your reasoning",
          "primary_entity": "The main entity/table being queried",
          "operation_type": "The type of operation",
          "parameter_differences": "Description of any parameter differences"
        }}
        """)

        response = llm.invoke(prompt.format(question1=question1, question2=question2

        # Extract JSON response and return decision
        # ...
```

Step 4: Modifying SQL — Precision Tweaks

And here's the function that handles the SQL modification: Once validated, we tweak the cached SQL for the new question:

Let's look at a real example from my testing:

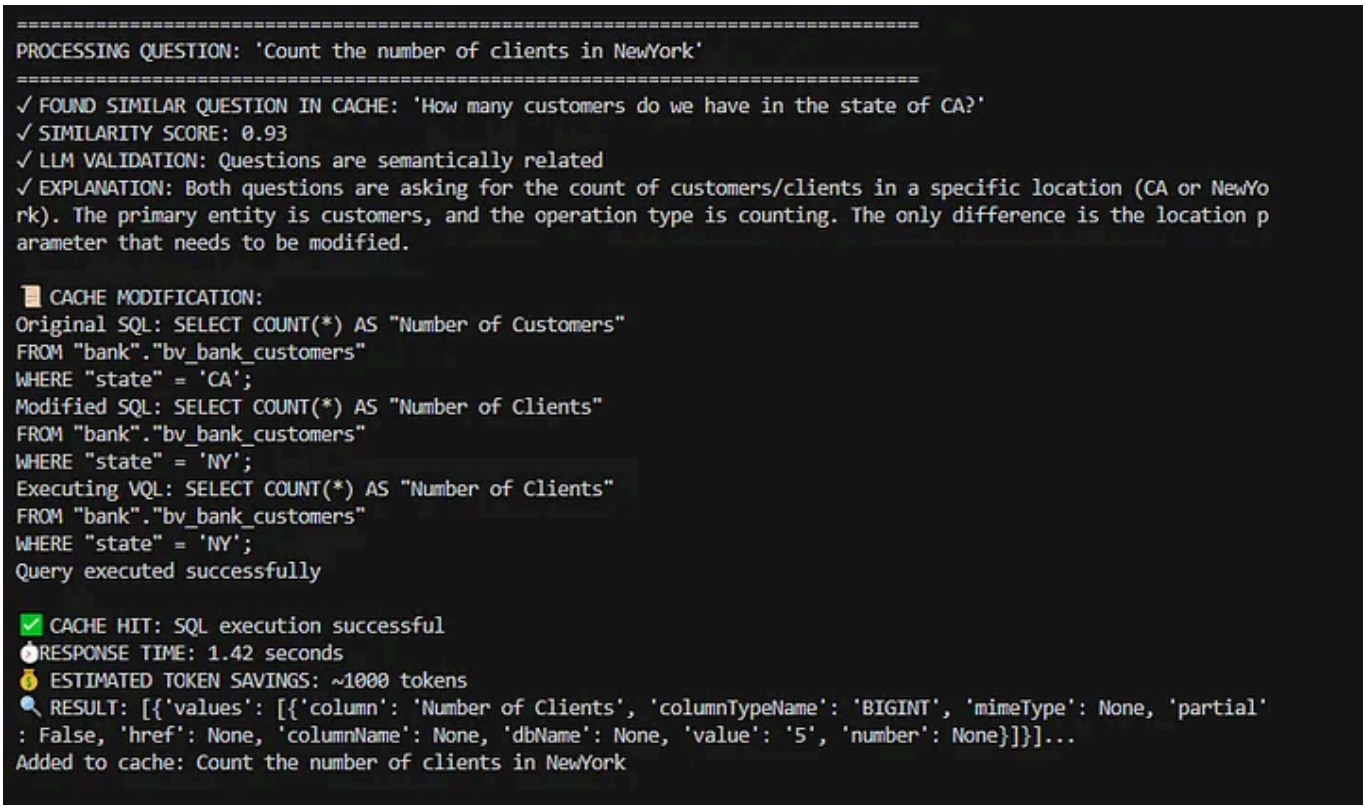


Image by Author

In this example, when processing “Count the number of clients in NewYork”:

1. The system found a similar question “How many customers do we have in the state of CA?” with 0.93 similarity
2. The LLM confirmed these are semantically related — both counting customers in specific states
3. It modified only the state code in the WHERE clause from ‘CA’ to ‘NY’
4. The query executed successfully in just 1.42 seconds
5. The result showed 5 clients in NewYork

This entire process took just 1.42 seconds, compared to the 10+ seconds it would have taken with the standard Denodo AI SDK approach. The modification was minimal and precise, changing just the state code while preserving the query structure.

Here’s the code snippet:

```
def modify_sql_query(original_query, new_question, original_question):
    """
    Use a small language model (GPT-3.5 Turbo) to modify the original VQL query.
    This function could be implemented with open-source models like Llama 3 or M
    for further cost reduction or on-premises deployment.
    """
    llm = ChatOpenAI(api_key=OPENAI_API_KEY, model="gpt-3.5-turbo")

    prompt = ChatPromptTemplate.from_template("""
    Original question: {original_question}
    Original SQL query: {original_query}

    New question: {new_question}
```

```
        Please analyze the differences between the questions and modify the SQL quer
        Common changes include:
        - Different parameters (e.g., top 5 vs top 10)
        - Different time periods or date ranges
        - Different filtering conditions (e.g., state codes like CA vs NY)

        Look at the original SQL to understand the schema and table structure. For s
        it should always use the state code, even when the question contains names.

        Only output the modified SQL query, with no additional text.
        """)

        chain = prompt | llm | StrOutputParser()
        modified_query = chain.invoke({
            "original_question": original_question,
            "original_query": original_query,
            "new_question": new_question
        })

        # Clean up any markdown formatting and return
        # ...
```

Step 5: Integrating with Denodo — The Full Picture

The main() function ties it all together:

1. **Initialize**: Set up the vector embeddings model, semantic cache database, and Denodo AI SDK client to enable the full workflow.
2. **Process Questions**: Loop through diverse test cases that demonstrate various patterns like parameter variations (“Top 5 customers with highest loans” → “Top 10 customers…”), wording differences (“How many customers in CA?” → “Count clients in California”), and complex queries with joins and sorting.
3. **Apply the Pattern**: For each question, search for similar cached questions, validate semantic relationships, modify SQL when appropriate, and measure performance improvements.

5. **Add to Cache DB** : Save the User Question, SQL Query and Query Explanation to the cache database.
6. **Measure Benefits**: Track and visualize performance metrics showing response time improvements, token savings, and cache hit rates.

Here's a snippet:

```
for question in test_questions:
    cached_question, cached_sql, cached_result, similarity = cache.find_similar_
    if cached_question and are_questions_semantically_related(cached_question, q
        modified_sql = modify_sql_query(cached_sql, question, cached_question)
        status_code, result = execute_vql(modified_sql, auth)
        if 200 <= status_code < 300:
            cache.add_to_cache(question, modified_sql, f"Query that {question}")
        else:
            sdk_result = denodo_client.answer_data_question(question)
            cache.add_to_cache(question, sdk_result["sql_query"], sdk_result["ex
    else:
        sdk_result = denodo_client.answer_data_question(question)
        cache.add_to_cache(question, sdk_result["sql_query"], sdk_result["execut
```

Seeing the Results — 9x Faster in Action

When I tested this approach on a sample banking database with varied natural language queries, the difference in performance was striking:

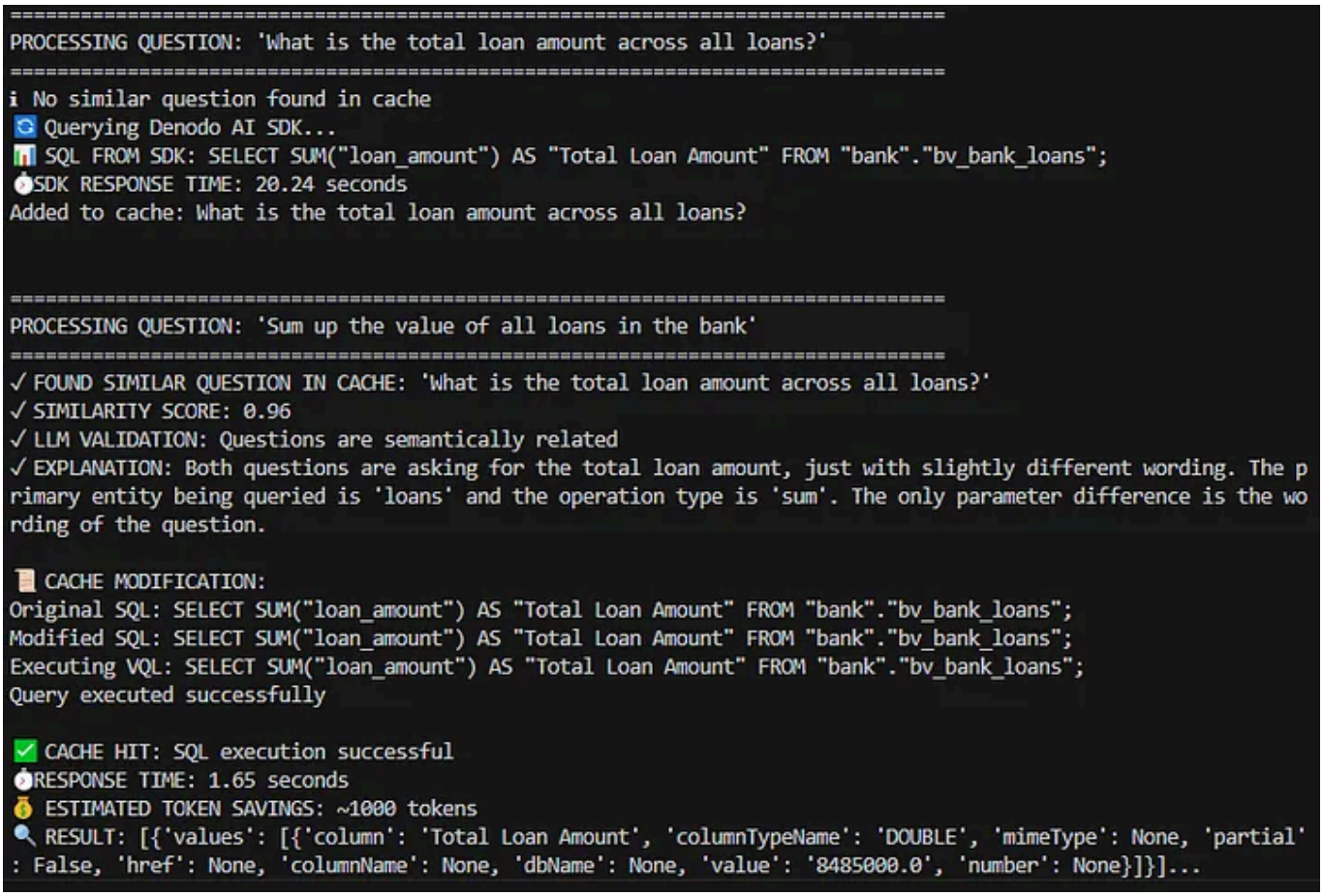


Image by Author

In this comparison, you can see:

1. The first question “What is the total loan amount across all loans?” had no cache match, so it went through the full Denodo AI SDK pipeline, taking 20.24 seconds to complete.
2. For the follow-up question “Sum up the value of all loans in the bank”, the system:

- Found a match with 0.96 similarity
- Confirmed semantic relatedness through LLM validation
- Modified just the column label while keeping the same query structure

- Executed in just 1.65 seconds — **9x faster** than the Denodo AI SDK direct approach

These real-world examples demonstrate the dramatic performance improvements:

- **40x reduction in cost per question:** Using small, efficient models (GPT-3.5 Turbo) or even open-source models for SQL modification instead of full-featured LLMs
- **5–9x reduction in response time:** Modified queries processed in 1–2 seconds vs 10–15 seconds for full generation
- **Enhanced accuracy:** Modifying proven SQL queries had fewer errors than generating from scratch
- **Cross-domain support:** The same technique works for questions across different business domains within the same database

Aggregation and TOP — N Testing with Cache:

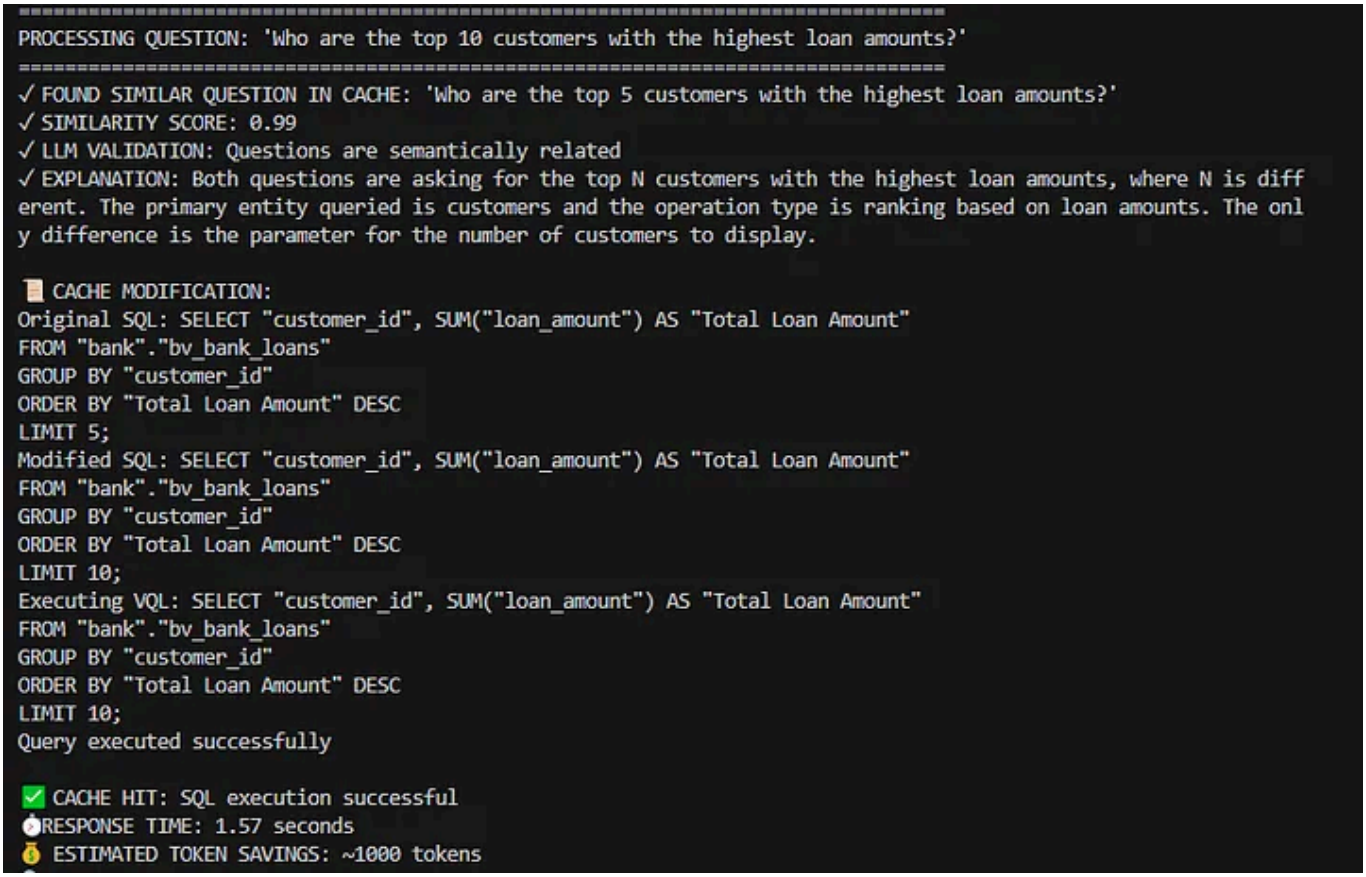


Image by Author

Joins Example with Cache

```
PROCESSING QUESTION: 'Rank borrowers by loan size, with alphabetical ordering by surname when amounts match'
================================================================================
✓ FOUND SIMILAR QUESTION IN CACHE: 'Sort customers by total loan amount (highest first) and then alphabetically
 by last name for ties'
✓ SIMILARITY SCORE: 0.97
✓ LLM VALIDATION: Questions are semantically related
✓ EXPLANATION: Both questions are about customers and involve ranking them based on loan amounts with secondary
 sorting by last name. The operations of sorting and ranking are closely related and the SQL queries can be mod
ified with minor changes.

📝 CACHE MODIFICATION:
Original SQL: SELECT
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name",
    SUM("l"."loan_amount") AS total_loan_amount
FROM
    "bank"."bv_bank_customers" "c"
JOIN
    "bank"."bv_bank_loans" "l"
ON
    "c"."customer_id" = "l"."customer_id"
GROUP BY
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name"
ORDER BY
    total_loan_amount DESC,
    "c"."last_name";
Modified SQL: SELECT
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name",
    SUM("l"."loan_amount") AS loan_size
FROM
    "bank"."bv_bank_customers" "c"
JOIN
    "bank"."bv_bank_loans" "l"
ON
    "c"."customer_id" = "l"."customer_id"
GROUP BY
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name"
ORDER BY
    loan_size DESC,
    "c"."last_name";
```

Image by Author

```
Executing VQL: SELECT
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name",
    SUM("l"."loan_amount") AS loan_size
FROM
    "bank"."bv_bank_customers" "c"
JOIN
    "bank"."bv_bank_loans" "l"
ON
    "c"."customer_id" = "l"."customer_id"
GROUP BY
    "c"."customer_id",
    "c"."first_name",
    "c"."last_name"
ORDER BY
    loan_size DESC,
    "c"."last_name";
Query executed successfully

✅ CACHE HIT: SQL execution successful
⏱️RESPONSE TIME: 2.43 seconds
💰 ESTIMATED TOKEN SAVINGS: ~1000 tokens
🔍 RESULT: [{'values': [{'column': 'customer_id', 'columnTypeName': 'INTEGER', 'mimeType': None, 'partial': Fal
se, 'href': None, 'columnName': None, 'dbName': None, 'value': '22', 'number': None}, {'column': 'first_name',
 'columnTypeName': 'NVARCHAR', 'mimeType': None, 'partial': False, 'href': None, 'columnName': None, 'dbName': N
one, 'value': 'Kayla', 'number': None}, {'column': 'last_name', 'columnTypeName': 'NVARCHAR', 'mimeType': None,
 'partial': False, 'href': None, 'columnName': None, 'dbName': None, 'value': 'King', 'number': None}, {'column
': 'loan_size', 'columnTypeName': 'DOUBLE', 'mimeType': None, 'partial': False, 'href': None, 'columnName': Non
e, 'dbName': None, 'value': '1200000.0', 'number': None}]}, {'values': [{'column': 'customer_id', 'columnTypeNa
me': 'INTEGER', 'mimeType': None, 'partial': False, 'href': None, 'columnName': None, 'dbName': None, 'value':
'9', 'number': None}, {'column': 'first_name', 'columnTypeName': 'NVARCHAR', 'mimeType': None, 'partial': False
, 'href': None, 'columnName': None, 'dbName': None, 'value': 'Robert', 'number': None}, {'column': 'last_name',
 'columnTypeName': 'NVARCHAR', 'mimeType': None, 'partial': False, 'href': None, 'columnName': None, 'dbName':
None, 'value': 'Taylor', 'number': None}, {'column': 'loan_size', 'columnTypeName': 'DOUBLE', 'mimeType': None,
 'partial': False, 'href': None, 'columnName': None, 'dbName': None, 'value': '675000.0', 'number': None}]}, {'
values': [{'column': 'customer_id', 'columnTypeName': 'INTEGER', 'mimeType': None, 'partial': False, 'href': No
ne, 'columnName': None, 'dbName': None, 'value': '3', 'number': None}, {'column': 'first_name', 'columnTypeName
': 'NVARCHAR', 'mimeType': None, 'partial': False, 'href': None, 'columnName': None, 'dbName': None, 'value': '
Michael', 'number': None}, {'column': 'last_name', 'columnTypeName': 'NVARCHAR', 'mimeType': None, 'partial': F
alse, 'href': None, 'columnName': None, 'dbName': None, 'value': 'Jackson', 'number': None}, {'column': 'loan_s
ize', 'columnTypeName': 'DOUBLE', 'mimeType': None, 'partial': False, 'href': None, 'columnName': None, 'dbName
': None, 'value': '570000.0', 'number': None}]}]...
Added to cache: Rank borrowers by loan size, with alphabetical ordering by surname when amounts match
```

Image by Author

Semantic Cache Performance Report

This dashboard presents results from testing our semantic cache with 18 questions across 5 categories, with each category containing variations of the same question phrased differently.

Image By Author

Key Takeaways:

- **Token Efficiency:** 97.1% token reduction with 71,353 total tokens saved
- **Speed Improvement:** 8.9x faster response times (2.45s vs 21.47s)

- **Cache Effectiveness:** 83.3% hit rate (15/18 questions served from cache)
- **Cost Savings:** Significant reduction in API costs at scale
- **Semantic Recognition:** Successfully identified question variations despite different phrasing

The results confirm that our semantic cache effectively recognizes similar questions regardless of phrasing variations, delivering substantial performance and cost benefits.

Implementation Tips From Experience

After implementing this semantic caching with Denodo AI SDK, here are key lessons learned:

1. **Start with Seed Questions** Create 30–40 common questions across your data domains to build an initial cache. This gives the system a foundation before handling user queries.
2. **Tune Your Similarity Threshold** We found 0.85 vector similarity to be the sweet spot. Higher misses opportunities; lower creates false positives.
3. **Balance Model Selection**

- Novel questions: Full Denodo AI SDK capabilities
- Validation and modification: Smaller, faster models
- Embedding: Choose based on dimensionality needs

4. Two-Step Validation Process The two-step process (vector similarity → semantic validation) ensures high accuracy while maintaining performance.

Looking Ahead: Future Enhancements

We can explore these additional optimizations to further improve the semantic cache system:

1. **Adaptive Similarity Thresholds** — Dynamically adjust matching thresholds based on historical performance and domain context
2. **Proactive Cache Warming** — Preload common question variations to improve initial hit rates
3. **Federated Query Optimization** — Intelligently decompose complex queries to maximize cache utilization
4. **Vector Database Integration** — Replace in-memory FAISS with scalable vector stores for enterprise-grade performance

Summary : Taking Denodo AI SDK to the Next Level

Semantic caching takes Denodo AI SDK's already powerful capabilities and makes them even more responsive for users asking similar questions. This Semantic caching preserves all the benefits of Denodo's data virtualization platform, secure access to trusted data across silos , while dramatically improving performance for common query patterns.

By complementing Denodo's comprehensive text-to-SQL pipeline with targeted optimizations for similar queries, we've created a system that delivers:

- **9x faster responses** for common question variations
- **Enhanced user experience** through more responsive interactions
- **Optimized resource utilization** by avoiding redundant processing
- **Full compatibility** with existing Denodo implementations

For enterprises leveraging Denodo AI SDK, semantic caching represents a straightforward enhancement that makes an already excellent platform even better by delivering lightning-fast responses for similar queries while maintaining Denodo's industry-leading data governance and security.

The complete semantic cache evaluation code is available on GitHub, with documentation for seamless integration with your Denodo AI SDK environment.

Go through my other articles if you want to know more about DENODO AI SDK integration

How I Integrated Enterprise Data into AI Application in Minutes with Denodo

Stop Wrestling with Data Silos — Let Denodo Do the Heavy Lifting!

generativeai.pub

How denodo solves Text-to-SQL Challenges

Why Text-to-SQL Falls Short in Production: Challenges and Smarter Alternatives

Why Text-to-SQL Stumbles in Production and How to Overcome It

generativeai.pub

Resources

- What is Semantic Caching? — Redis blog article explaining semantic caching concepts and benefits
- Denodo Generative AI Solutions — Overview of Denodo's capabilities and offerings in generative AI
- Denodo AI SDK User Manual — Official documentation for the Denodo AI SDK
- FAISS Similarity Search — Information about FAISS and how it enables efficient similarity search for embedding vectors

Writing | Artificial Intelligence | Productivity | Technology | Marketing

Responses (3)

Alex Mylnikov

What are your thoughts?

Nazeer Khan
2 days ago

Nice

48 1 reply Reply

Calendula
2 days ago

I acknowledge that I just passed by. I am not from this field. I don't know what you're saying, but I am glad that there are people passionate about their work.

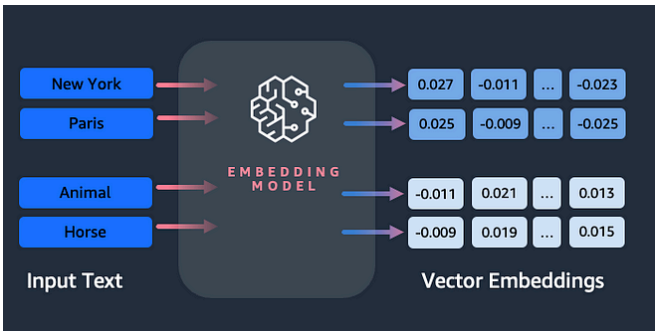
60 2 replies Reply

Nadeem Chuhan
19 hours ago

I believe in writing because it helps me see my thoughts clearly and improves my critical thinking.

Reply

More from MKWriteshere and Data Science Collective

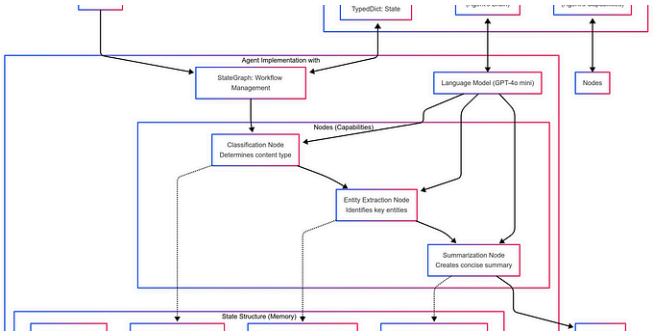


In Generative AI by MKWriteshere

Finding the Best Open Source Embedding Model for Text-to-SQ...

Evaluating Open Source Embedding Models for Enhancing Text-to-SQL Performance

Mar 3 50



In Data Science Collective by Paolo Perrone

The Complete Guide to Building Your First AI Agent (It's Easier Th...

Three months into building my first commercial AI agent, everything collapsed...

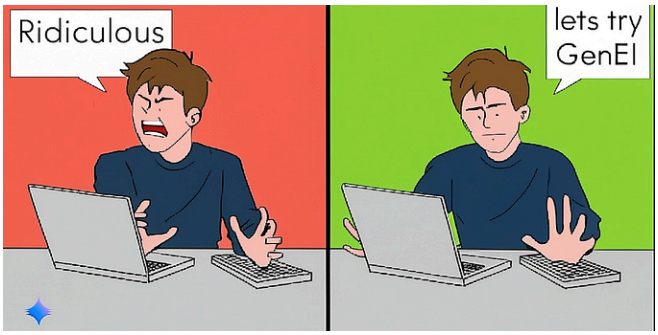
Mar 11 1.8K 43

In Data Science Collective by Ari Joury, PhD

Stop Copy-Pasting. Turn PDFs into Data in Seconds

Automate PDF extraction and get structured data instantly with Python's best tools

Feb 21 1.2K 27



In Data Science Collective by MKWriteshere

GenAI for GenEI: How "Emotionless" AI Can Strengthen...

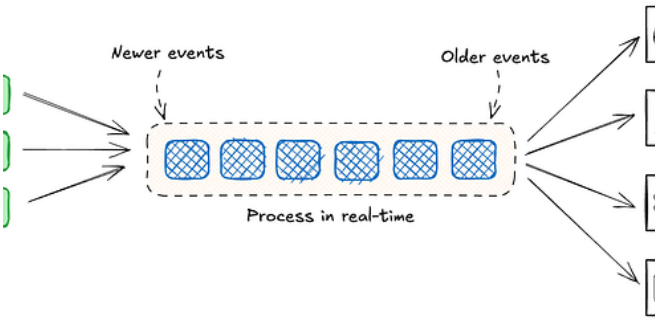
Generative Emotional Intelligence using GenAI

2d ago 30 1

See all from MKWriteshere

See all from Data Science Collective

Recommended from Medium



Sean Falconer

The Future of AI Agents is Event-Driven

AI agents promise autonomy and adaptability. Event-driven architecture provides the...

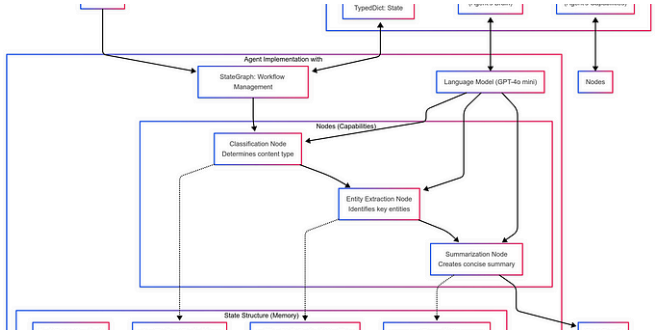
Mar 12 861 21

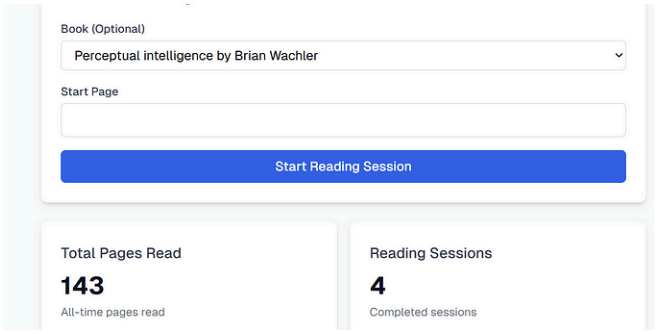
Eduard Ruzga

Claude with MCPs Replaced Cursor & Windsurf—How Did That...

You can see in the screenshot that I was using Windsurf in December. But by January and...

Feb 26 722 16





In Data Science Collective by Paolo Perrone

The Complete Guide to Building Your First AI Agent (It's Easier Th...

Three months into building my first commercial AI agent, everything collapsed...

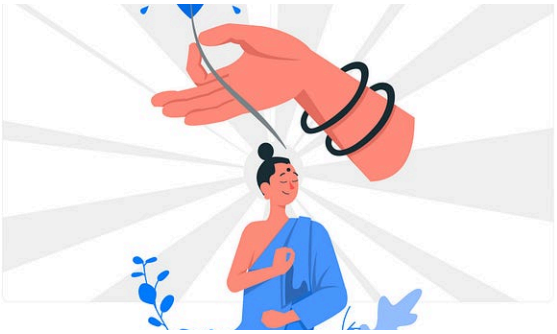
Chris Dunlop

Is Cursor better than VS Code with Copilot? Absolutely and it's not...

Cursor 3.7 + Claude is the first time for me in coding that AI has got to a level that makes it...

Mar 11 1.8K 43

Mar 8 435 11



In The Preamble by K.W. Hampton, PhD, MPA

What Google and Meta's Leaked Internal Memos Reveal About...

They're scared as sh*t!

Mar 3 4K 144

In Code Like A Girl by Nidhi Jain 👩‍💻

9 Lessons from a Principal Engineer That Made Me a Better...

The small but important missing pieces that no one talks about

Mar 10 1.3K 27

See more recommendations