Medium Search Write Sign up Sign in

Get unlimited access to the best of Medium for less than $5/week. Become a member

Supercharging Big Data: The Complete Guide to Integrating LLMs and Agents with PySpark

Sachin Gangan Follow 6 min read · Sep 8, 2025

Integrating LLMs and Agents with PySpark

In the rapidly evolving landscape of data science and artificial intelligence, two powerful technologies — large language models (LLMs) and Apache Spark's Python API, PySpark — are converging to create unprecedented opportunities for processing and analyzing big data. This comprehensive guide explores how organizations can integrate these cutting-edge technologies to new capabilities in big data processing, analysis, and intelligent automation.

Introduction

The marriage of Large Language Models with distributed data processing frameworks represents one of the most exciting developments in enterprise data analytics. While traditional big data processing excels at structured data with the sophisticated reasoning and natural language understanding of LLMs, organizations can now process massive datasets while applying advanced AI capabilities to extract deeper insights, automate complex workflows, and enable natural language interfaces for data exploration.

This integration addresses a fundamental challenge in the AI ecosystem: while LLMs demonstrate remarkable reasoning and language capabilities, they often lack efficient mechanisms for processing large-scale data. Conversely, while PySpark excels at distributed data processing, it traditionally required specialized technical knowledge to operate effectively. Bringing these technologies together creates synergies that overcome the limitations of each while amplifying their respective strengths.

As we explore this technological convergence, we'll examine implementation approaches, industry trends, practical applications, and future directions. Whether you're a data engineer looking to enhance your big data pipelines with AI capabilities, a data scientist seeking to scale your language models, or a technical leader evaluating enterprise AI strategies, this guide provides the insights needed to navigate this exciting frontier.

Background and Evolution

The Rise of PySpark in Big Data Processing

PySpark emerged as a Python API for Apache Spark, which itself was developed at UC Berkeley's AMPLab in 2009 and became a major Apache project by 2014. The framework revolutionized distributed computing with its lightning-fast performance, offering 100 times faster processing for significantly faster data operations and a more flexible programming model.

PySpark brought Spark's powerful distributed computing capabilities to the Python ecosystem, enabling data scientists and engineers familiar with Python and its libraries to scale their Python-based workflows across clusters. Key features of PySpark include robust libraries for machine learning (MLlib), streaming data processing, graph processing, and SQL operations.

The Evolution of LLMs and AI Agents

Large Language Models represent the culmination of decades of research in natural language processing. From early rule-based systems to statistical models, through neural networks and transformer architectures, LLMs have the capabilities of language models have given organizations The release of models like GPT, BERT, and their successors marked a pivotal moment when language models began demonstrating remarkable understanding of human language and context.

AI agents — software systems that can perceive their environment, reason about it, and take actions to achieve specific goals — have evolved alongside LLMs. Modern AI agents often leverage LLMs as their reasoning engines, combining them with planning capabilities, tool use, and memory systems to create more capable autonomous systems.

Early Approaches to Integration

The integration of distributed computing frameworks with language models initially faced significant challenges. Early attempts often involved:

1. **Batch processing pipelines**: Using PySpark to preprocess data before feeding it into standalone language models
2. **Simple inference layers**: Adding basic NLP capabilities to PySpark applications through libraries like SparkNLP
3. **Custom extensions**: Building proprietary bridges between Spark clusters and external AI services

As noted by Bert Hamer in a Medium article, these early integrations aimed to "leverage the LLM Large language model to enhance the data processing and analysis using PySpark." However, these approaches often suffered from performance limitations, complex architectures, and challenges in maintaining consistency between different processing environments.

Recent advancements have enabled more seamless integration, with frameworks that allow LLMs and agents to work directly with PySpark's distributed data structures, enabling truly scalable AI-powered data processing.

Technical Implementation Approaches

Integrating LLMs and intelligent agents with PySpark can be approached through several architectural patterns, each with its own advantages and use cases.

Direct Integration Using PySpark-AI

One of the most exciting developments in this space is PySpark-AI, described as the "English SDK for Apache Spark." This open-source library allows data engineers and scientists to use natural language instructions to create and manipulate PySpark objects.

According to the GitHub repository, PySpark-AI "takes English instructions and compiles them into PySpark objects like DataFrames." This approach fundamentally changes how users interact with PySpark, enabling data processing accessible to those without deep technical knowledge of PySpark's APIs.

A typical implementation using PySpark-AI looks like this:

```
# Install required packages
pip install pyspark langchain spacy pyspark-ai

# Download English language model for spaCy
python -m spacy download en_core_web_sm

# Export your OpenAI API key (required for default configuration)
export OPENAI_API_KEY="your-openai-api-key"

# Start PySpark shell
pyspark --packages com.crossverse.data.ai:1.0.0

# Create a SparkSession with AI capabilities
from pyspark.sql import SparkSession
from pyspark_ai.spark_ai import SparkAI

spark = SparkSession.builder \
    .appName("PySpark-AI Example") \
    .getOrCreate()

# Initialize SparkAI with default configuration
spark_ai = SparkAI(spark=spark, verbose=True)

# Activate the AI functions
spark_ai.activate()

# Create a DataFrame from sample data
df = spark.range(100).toDF("id")

# Use natural language to generate transformations
df_ai = df.ai.transform("add a column called 'is_even' that checks if the value in the 'id' column is even")

# Show the results
df_ai.show()

# Ask questions about the data
print(df_ai.ai.ask("how many even numbers are there?"))
```

This approach leverages LangChain and OpenAI frameworks behind the scenes to translate natural language instructions into valid PySpark operations, enabling complex data transformations accessible through simple English commands.

LLM-Powered ETL and Data Processing

Another approach involves using LLMs to generate or optimize PySpark code for Extract, Transform, Load (ETL) processes. This pattern is particularly valuable when organizations need to quickly develop or refine data pipelines.

As explored in an analysis by Turbolearn.ai, modern LLMs like GPT-4 can generate efficient PySpark code for data transformation tasks when provided with clear requirements. This approach typically involves:

1. Describing the desired data transformation in natural language
2. Using an LLM to generate optimized PySpark code
3. Validating and executing the generated code in a PySpark environment

This pattern can significantly accelerate development cycles by automating the creation of complex data transformation logic.

Distributed Vector Search with PySpark and Vector Databases

For applications requiring similarity search across large datasets — a common requirement in recommendation systems, anomaly detection, and natural language processing — the integration of PySpark with vector databases provides an efficient solution.

According to a Tech-talend tutorial, integrating Pinecone (a vector database) with PySpark enables efficient similarity searches across massive datasets. This architectural pattern typically involves:

1. Using PySpark for data preprocessing and feature engineering
2. Generating embeddings for the data using LLMs
3. Storing these embeddings in a vector database
4. Performing distributed similarity searches across the database

This approach allows organizations to implement semantic search capabilities that scale with their data volume.

Agent-Based Workflow Orchestration

More sophisticated integrations involve using intelligent agents to orchestrate complex workflows that span both PySpark and LLM and external services. In this pattern, an LLM powered agent acts as the coordinator, making decisions about data processing steps based on intermediate results.

This approach is particularly valuable for complex analytics pipelines that require adaptive processing. The agent can monitor data quality metrics, analyze data quality metrics from a PySpark job, decide which cleaning operations to apply, and then determine the most appropriate modeling approach based on the characteristics of the processed data.

Current Industry Trends

The integration of LLMs and agents with PySpark is gaining traction across industries, with several notable trends emerging in implementation approaches and use cases.

English-First Data Engineering

One of the most significant trends is the move toward "English-first" data engineering, where natural language becomes the primary interface for working with big data systems. This trend is exemplified by tools like PySpark-AI, which enable data professionals to express their intent in plain English rather than writing complex code.

This approach is democratizing access to big data capabilities, allowing domain experts without deep technical knowledge to perform sophisticated analyses. As Sparkbyexamples.com notes, "pyspark.ai leverages large and spatial framework to use Google Large Language Models (LLM) to simplify the usage of PySpark."

Automated Data Quality and Transformation

Organizations are increasingly using LLMs to automate the development and optimization of data transformation logic. This trend is driven by the ability of advanced language models to understand data structures and generate efficient PySpark code for complex transformations.

According to Turbolearn.ai's comparison of LLM-generated PySpark code, SI models can generate code that follows best practices, uses appropriate functions, offers matching or exceeding the quality of code written by human experts. This trend is accelerating development cycles and reducing the technical debt associated with manually maintaining complex data pipelines.

Cloud Provider Integration

Major cloud providers are investing heavily in integrating LLMs with their Spark services. For example, Amazon Web Services has been implementing Retrieval Augmented Generation (RAG) at scale using AWS Glue for Apache Spark.

As described in an AWS blog post, "To make this solution scalable and performant, we implement the data preprocessing and embedding capabilities of Spark in our RAG implementation. Vector embeddings are making it easier for organizations to leverage sophisticated AI capabilities on top of their existing data infrastructure.

Industry-Specific Applications

Different industries are finding unique ways to leverage the integration of LLMs with PySpark:

- **Financial Services**: Implementing fraud detection systems that combine transaction pattern analysis (via PySpark) with natural language processing of communication data (via LLMs)
- **Healthcare**: Analyzing patient records at scale while extracting insights from unstructured clinical notes
- **Retail**: Enhancing recommendation engines with natural language understanding of product descriptions and customer reviews
- **Manufacturing**: Optimizing production processes by analyzing sensor data alongside maintenance logs and technical documentation

According to a Medium article on industry-specific LLM applications, "LLM agents can be tailored to specific industries, such as healthcare, finance, or spending on complex tasks, improving decision-making, and automating processes." When combined with PySpark's scalable data processing, these capabilities become available for enterprise-scale datasets.

Social Discussions

The integration of LLMs with PySpark has sparked numerous discussions across social media platforms, providing valuable insights into practitioners' experiences, challenges, and emerging best practices.

Twitter Insights

On Twitter, data professionals have been sharing experiences and insights about integrating LLMs with PySpark.

Allison Wang (@allisonwang0) highlighted the expanding integration capabilities: "Now, data engineers can efficiently integrate a broader array of intelligent capabilities into their data processing workflows, including PyTorch, MLlib, XGBoost." This observation points to the growing ecosystem of connectors that enable Spark to interact with diverse data sources, a critical capability for LLM training and inference.

Cheng Lian (@lichengcs) shared an impressive performance milestone: "Casually performed a 5B x 13M cross join using PySpark on DataBricks serverless." Such examples demonstrate the scale at which modern PySpark applications can operate, making them suitable for processing large-scale LLM inference and data processing.

Ismail Prabusai (@ismailprabusai) noted similarities between LLM interfaces and existing data processing paradigms: "I feel they render easy to read solutions, and are similar to the piping approaches in pyspark, pandas, tidyr and the tidy." This observation highlights how natural the integration of LLMs into data workflows can be, as they adopt familiar patterns that data engineers already use.

Reddit Discussions

On Reddit, the community has been exploring more detailed aspects of implementation and sharing open-source projects.

A notable thread in r/dataengineering titled "[Open Source] PySpark-AI: Turn text into a Universal Interface" discussed an open-source project that connects PySpark to LLMs. The project's creator explained that the project represents the growing trend of enhancing development environments with LLM capabilities to simplify complex PySpark implementations.

In r/apachespark, a discussion about "Organizing PySpark Transformations with Python" explored frameworks for structuring PySpark transformations, highlighting the community's focus on creating maintainable, scalable architectures for implementing complex data pipelines that might incorporate LLM components.

A thread in r/ai/ai-agents about "Model training on Datalecks" included the insight that "Pyspark with distributed backend like ray/spark/ dask/beam/k8s is a good combination." This discussion highlights the importance of distributed perspective emphasizing the practical advantages of PySpark based solutions for deploying machine learning models, including LLMs, at scale.

These social media discussions reveal a community actively exploring the integration possibilities, with a focus on practical implementation details, performance optimization, and architectural patterns that work in production environments.

Performance Optimization and Scaling

When integrating LLMs with PySpark, performance optimization becomes critical due to the resource-intensive nature of language model inference and the potentially massive scale of data processing tasks.

Optimizing PySpark for LLM Workloads

Several optimization techniques can significantly improve the performance of PySpark applications that incorporate LLMs:

1. Efficient Join Strategies

According to a Linkedin article on optimizing PySpark joins, choosing the right join strategy can dramatically impact performance:

- **Broadcast Joins**: Ideal when joining a large dataset with a small one that can fit in memory
- **Sort-Merge Joins**: The default for large datasets, but can be optimized with proper partitioning
- **Bucket Joins**: Pre-organizing data to optimize specific join patterns

For LLM applications that need to combine large feature datasets with model outputs, these optimization strategies can significantly reduce processing time.

2. Data Serialization and Storage Formats

As noted in Linkedin's article on optimizing PySpark joins, "PySpark performance can be optimized by using techniques like data serialization and caching, and the Spark DataFrame API."

Specific recommendations include:

- Using Parquet or ORC formats for efficient storage and retrieval
- Leveraging Kryo serialization for faster data transfer between nodes
- Strategic caching of frequently accessed DataFrames

3. Resource Allocation and Cluster Configuration

Proper resource allocation is essential for LLM workloads, which are typically more compute-intensive than traditional data processing tasks:

- Allocating more memory to executor instances to accommodate model parameters
- Configuring the right level of parallelism based on cluster resources
- Using dynamic allocation to adapt resource usage based on workload

4. Partitioning Strategies

As Analytics Vidhya articles on PySpark best practices emphasizes the importance of proper partitioning:

- Repartitioning data based on the columns used for filtering or joining
- Avoiding data skew by choosing appropriate partition keys
- Coalescing partitions when the number becomes too high

Distributed LLM Inference

Running LLM inference across a distributed Spark cluster presents unique challenges and opportunities:

1. Batching and Pipelining

Efficiency can be significantly improved by:

- Batching requests to maximize throughput
- Implementing pipelined processing to overlap computation with data transfer
- Using adaptive batch sizes based on input complexity

2. Model Parallelism vs. Data Parallelism

Different strategies for distributing LLM workloads across a cluster include:

- **Data Parallelism**: Processing different data chunks with the same model on different nodes
- **Model Parallelism**: Distributing a single model across multiple nodes (typically more complex to implement)
- **Pipeline Parallelism**: Dividing model layers across different nodes

For most PySpark integrations, data parallelism provides the simplest implementation with good scaling properties.

3. Model Quantization and Optimization

To improve inference speed and reduce resource requirements:

- Using quantized versions of LLMs (e.g., 4-bit or 8-bit quantization)
- Implementing model pruning to reduce parameter count
- Leveraging specialized hardware accelerators like GPUs through appropriate PySpark configurations

Practical Implementation Guide

Let's walk through a step by step tutorial for integrating LLMs with PySpark, focusing on practical implementation patterns.

Setting Up the Environment

First, we need to configure our environment with the necessary dependencies:

```
# Install required packages
pip install pyspark langchain spacy pyspark-ai

# Download English language model for spaCy
python -m spacy download en_core_web_sm

# Export your OpenAI API key (required for default configuration)
export OPENAI_API_KEY="your-openai-api-key"

# Start PySpark shell with the AI connector package
pyspark --packages com.crossverse.data.ai:1.0.0

# Import required libraries
from pyspark.sql import SparkSession
from pyspark_ai.spark_ai import SparkAI

# Create a Spark session
spark = SparkSession.builder \
    .appName("PySpark-AI Example") \
    .getOrCreate()

# Initialize SparkAI with default configuration
spark_ai = SparkAI(spark=spark, verbose=True)

# Activate the AI functions
spark_ai.activate()

# Create a sample DataFrame
df = spark.range(100).toDF("id")

# Use natural language to create a new column
df_ai = df.ai.transform("add a column called 'is_even' that checks if the value in the 'id' column is even")

# Show the results
df_ai.show()

# Ask questions about the data
print(df_ai.ai.ask("how many even numbers are there?"))
```

Approach 1: Using PySpark-AI for Natural Language Data Analysis

PySpark-AI enables natural language interfaces to PySpark operations:

```
# Import required libraries
from pyspark.sql import SparkSession
from pyspark_ai.spark_ai import SparkAI

# Create a Spark session
spark = SparkSession.builder \
    .appName("PySpark-AI Example") \
    .getOrCreate()

# Initialize SparkAI with default configuration
spark_ai = SparkAI(spark=spark, verbose=True)

# Activate the AI functions
spark_ai.activate()

# Create a sample DataFrame with some data
df = spark.range(100).toDF("id")

# Use natural language to create a new column
df_ai = df.ai.transform("add a column called 'is_even' that checks if the value in the 'id' column is even")

# Show the results
df_ai.show()

# Ask questions about the data
print(df_ai.ai.ask("how many even numbers are there?"))
```

```
# Initialize Spark2
spark_df = SparkSession\

# Load sample data
df = spark.read.csv("sales_data.csv", header=True, inferSchema=True)
df.create0rReplaceTempView("sales")

# Use natural language for analysis
response = llm.invoke("""
    Analyze the sales data to identify the top 5 products by revenue,
    break it down by region, and show month-over-month growth rates
""")

# Display results
result.show()
```

Approach 2: Implementing LLM-Based Feature Engineering

LLMs can be used to enrich structured data with insights from text fields:

```
from pyspark.sql.functions import udf, udf
from pyspark.sql.types import StringType, ArrayType
import requests
import json


# Define function to call LLM API
def analyze_sentiment(text):
    if text is None or len(text.strip()) == 0:
        return "neutral"

    response = requests.post(
        "https://api.openai.com/v1/chat/completions",
        headers={
            "Authorization": f"Bearer
{os.environ['OPENAI_API_KEY']}",
            "Content-Type": "application/json"
        },
        data=json.dumps(
            {"model": "gpt-3.5-turbo",
            "messages":
            [{"role": "system", "content": "You are a sentiment
analyzer. Respond with exactly one word: positive, negative, or
neutral."},
             {"role": "user", "content": f"Analyze the sentiment
of this text: {text}"}
            ],
            "temperature": 0
            }
        )

    result = response.json()
    sentiment = result.get("choices", [{}])[0].get("message",
{}).get("content", "").strip().lower()
    return sentiment


# Register UDF
sentiment_udf = udf(analyze_sentiment, StringType())

# Apply to DataFrame
feedback_df = spark.read.csv("customer_feedback.csv",
header=True, inferSchema=True)
enriched_df = customer_feedback_df.withColumn("sentiment",
sentiment_udf(feedback_df["text"]))

# Analysis results
sentiment_counts = enriched_df.groupBy("sentiment").count()
sentiment_counts.show()
```

Approach 3: Implementing Distributed Vector Search

Combining PySpark with vector databases for semantic search:

```
from langchain.embeddings import OpenAIEmbeddings
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, FloatType
import numpy as np


# Initialize embeddings model
embeddings = OpenAIEmbeddings()

# Create function to generate embeddings
def generate_embedding(text):
    if text is None or len(text.strip()) == 0:
        return [0.0] * 1536  # Default dimension for OpenAI
embeddings

    embedding = embeddings.model.embed_query(text)
    return embedding

# Register UDF
embedding_udf = udf(generate_embedding, ArrayType(FloatType()))

# Apply to product descriptions
products_df = spark.read.csv("products.csv", header=True,
inferSchema=True)
products_with_embeddings = products_df.withColumn("embedding",
embedding_udf(col("description")))

# Save embeddings for later use in a vector database
products_with_embeddings.write.parquet("product_embeddings.parquet")

# Example search function (simplified)
def find_similar_products(query_text, top_n=5):
    # Generate query embedding
    query_embedding = embeddings.model.embed_query(query_text)

    # Load product embeddings
    stored_embeddings =
spark.read.parquet("product_embeddings.parquet")

    # Define similarity function
    def cosine_similarity(v1, v2):
        return float(np.dot(v1, v2) / (np.linalg.norm(v1) *
np.linalg.norm(v2)))

    similarity_udf = udf(lambda x: cosine_similarity(x,
query_embedding), FloatType())

    # Calculate similarities and find top matches
    results = stored_embeddings.withColumn("similarity",
similarity_udf(col("embedding")))
                        .orderBy(col("similarity").desc())
                        .limit(top_n)

    return results.select("product_id", "name", "description",
"similarity")
```

Approach 4: Agent-Based Data Analysis

Creating an agent that can interact with PySpark data:

```
from langchain.agents import create_sql_agent
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain.sql_database import SQLDatabase
from langchain.llms import OpenAI


# Create connection string for Spark SQL
jdbc_url = "jdbc:spark://localhost:10000/default"

# Initialize database connection
db = SQLDatabase.from_uri(jdbc_url)

# Create LLM instance
llm = OpenAI(temperature=0)

# Create SQL agent toolkit
toolkit = SQLDatabaseToolkit(db=db, llm=llm)

# Create agent
agent = create_sql_agent(
    llm=llm,
    toolkit=toolkit,
    verbose=True
)

# Use agent to analyze data
agent.run("Analyze our customer data and tell me which customer
segments have the highest lifetime value")
```

Best Practices and Common Pitfalls

When implementing these integration patterns, keep in mind:

1. **Memory Management**: Monitor executor memory carefully when running LLM inference
2. **API Rate Limiting**: Implement backoff strategies when calling external LLM APIs
3. **Cost Control**: Batch requests and monitor API usage to prevent unexpected costs
4. **Caching**: Cache results of expensive LLM operations to avoid redundant API calls
5. **Error Handling**: Implement robust error handling for network failures and API issues
6. **Data Privacy**: Consider privacy implications when sending data to external LLM services

Challenges and Limitations

Despite the exciting possibilities, integrating LLMs with Pyspark presents several significant challenges that organizations must address.

Resource Requirements and Infrastructure Constraints

The combination of big data processing and language model inference places substantial demands on computing infrastructure:

- **Memory Consumption**: LLMs typically require significant memory for model parameters
- **Compute Requirements**: Inference at scale can become computationally expensive
- **Network Bandwidth**: Distributed processing increases network traffic, especially with large models

According to a Medium article on Spark ML challenges, "One of the main challenges is hardware and resource requirements. Implementing ML models, especially more complex ones, on large datasets requires substantial computational resources."

Organizations must carefully plan their infrastructure to balance cost and performance when implementing these integrations.

Integration Complexity

Building robust LLM-PySpark integrations requires expertise across multiple domains:

- **API Compatibility**: Ensuring smooth interaction between different libraries and frameworks
- **Versioning Issues**: Managing dependencies across the integration stack
- **Testing Complexity**: Validating behavior across distributed environments
- **Deployment Challenges**: Orchestrating the deployment of integrated systems

These challenges increase the technical debt associated with these integrations and require dedicated DevOps resources to address effectively.

LLM-Specific Limitations

The nature of large language models introduces additional challenges:

1. Hallucinations and Accuracy

LLMs are prone to generating plausible-sounding but incorrect information:

- When used for code generation, LLMs might produce PySpark code that looks correct but contains subtle bugs
- Generated analysis might include spurious patterns or conclusions
- Without appropriate guardrails, LLMs might confidently present inaccurate results

According to ProjectPro's article on LLM limitations, "These limitations span computational constraints, issues with accuracy and knowledge updating, lack of long-term memory, and struggles with complex reasoning."

2. Performance and Latency

Real-time LLM inference introduces latency that can impact interactive data analysis:

- Model inference may become a bottleneck in data processing pipelines
- Batch processing approaches might be necessary for large-scale applications
- Response time variability can affect user experience in interactive scenarios

As noted in Predibler's blog on LLM challenges, "As LLMs are increasingly being asked to solve complex problems with large amounts of data, one obvious challenge is latency & performance."

3. Security and Privacy Concerns

The integration of LLMs with sensitive data raises important security considerations:

- Data leakage risks when sending information to external LLM services
- Potential for prompt injection attacks
- Complexity in ensuring compliance with data protection regulations

According to HiddenLayer's analysis of LLM security issues, "Technology of any kind is always a double-edged sword. It can hugely improve our life, but it can also inadvertently cause harm." This is particularly relevant when combining powerful LLMs with enterprise data.

* * *

Ready to start your Pyspark ML journey? Begin by exploring the official documentation and hands-on tutorials, then gradually tackle increasingly complex use cases as you build your expertise in distributed machine learning.

Pyspark · Llm · Big Data · Generative-Ai Tools

Written by Sarthak Dargan
23 followers · 7 Following

Follow

No responses yet

What are your thoughts?

More from Sarthak Dargan

Sarthak Dargan
Paper Synopsis | Reinforcement Learning for Long-Horizon...
Why I Wrote This Blog
Apr 20

Sarthak Dargan
Unleashing the Power of Machine Learning with PySpark: A...
In today's data-driven world, the ability to process and analyze massive datasets for...
Apr 24

Paradise Dafaga in Sarthak Dargan
Paper Synopsis: CloudCast—A configuration-free invoice analysis...
Source: https://arxiv.org/pdf/1708.07452.pdf
May 27, 2025

Sarthak Dargan
Recommendation Engine for Dummies!
Imagine you had a lost breakup with your girlfriend, what music do you want to hear?
Jul 15, 2025

See all from Sarthak Dargan

Recommended from Medium

AI in Datamics
Governing AI Agents with Unity Catalog
Author: Jinde He (Senior Solutions Architect @ Databricks)
Jun 10

THE DECK LEARNING
Tadaf! MLflow 3.0 is here loaded with Gen AI Capabilities Day 1
The AI Stack Reimagined
Jun 4

Lizaveta Dzikaya in Towards Data Lake
Building a Complex, Production-Ready RAG System with...
Langchain/Langgraph, agents, tool-graphs, plan-execution, orchestration, and more.
Sep 11

Jackson Cox
Data Lake Architectures 2025
Ingestion, Storage, Processing, Governance, Consumption
Nov 11

Python in Plain English in Mahesh Alles
Building Scalable Task Queues with Celery in Production
How I Architected a Distributed Python Worker System That Handles Thousands of...
Jun 27

AI Advances in Dzimitsa Biness
Agentic AI Lifecycle for Enterprise Processes
The art of transforming Manual processes into an Orchestrated fleet of AI Agents
Jun 28

See more recommendations