Gowtham SB
www.linkedin.com/in/sbgowtham/

📘 Syllabus – 25 Gen AI Projects for Data Engineers

🧠 Section 1: Foundations of Gen AI for Data Engineers

- Top 25 Gen AI Terms Every Data Engineer Should Know
- Architecture Patterns: RAG, Chains, Agents
- Tools Overview: LangChain, FAISS, Pinecone, OpenAI, HuggingFace, Streamlit, FastAPI

⚙️ Section 2: Core Projects (1–10)

1. **LLM-based Data Quality Checker**
2. **AI SQL Generator**
3. **Data Pipeline Auto-Documenter**
4. **Chatbot for Data Lake Queries**
5. **Schema Drift Detector using LLM**
6. **Data Lineage Mapper**
7. **Auto PII Identifier**
8. **Data Catalog Summarizer**
9. **Smart ETL Code Reviewer**
10. **DAG Creation Assistant**

⚡ Section 3: Real-time & Monitoring Projects (11–17)

11. **Real-time Data Anomaly Detector**
12. **Semantic Search for Databases**
13. **API Generator from Data**
14. **Log Analyzer with LLM**
15. **Data Pipeline Cost Estimator**
16. **SLA Breach Predictor**
17. **LLM-enhanced Feature Store**

🔐 Section 4: Migration, Security & Optimization (18–23)

18. **Data Migration Assistant**
19. **OLAP Query Optimizer using LLM**
20. **SQL Explainer Bot**
21. **Synthetic Data Generator**
22. **Data Lake Explorer Assistant**
23. **Security Risk Detector for Data Pipelines**

📊 Section 5: Business + Reporting Use Cases (24–25)

24. **Business KPI Analyzer**
25. **Data Science Notebook Explainer**

Gowtham SB
www.linkedin.com/in/sbgowtham/

📎 Appendices

- Glossary: Top Gen AI Terms
- Mock Resume for Gen AI + Data Engineering
- Prompt Engineering Tips for Each Use Case
- Cloud & Cost Recommendations
- Diagram Gallery (for all 25 projects)

Gowtham SB
www.linkedin.com/in/sbgowtham/

Top 25 Gen AI Terms — Clean & Clear Descriptions

---

1. Token
A token is a piece of text (word or subword) that a language model reads and generates. It's how models break down and understand language.

2. Prompt
A prompt is the input or instruction given to an AI model to guide its output.

3. LLM (Large Language Model)
An AI model trained on massive text data to understand and generate human-like language.

4. Agent
A self-directed AI process that can use tools, memory, and reasoning to perform tasks.

5. Embedding
A numerical representation of text used for comparison, similarity search, or semantic understanding.

6. Vector Store
A database for storing embeddings that allows fast similarity search (e.g., FAISS, Pinecone).

7. Fine-tuning
Training a pre-trained model on specific data to specialize it for a task.

8. RAG (Retrieval-Augmented Generation)
Combines search with LLMs to generate answers from external sources/documents.

9. Chain
A sequence of steps (like prompt → LLM → tool → LLM) used to perform complex Gen AI tasks.

10. LangChain
A Python framework for building LLM-powered applications with memory, chains, and agents.

11. LangGraph
An extension of LangChain used to build agent workflows with branching logic (graph-based chains).

12. Prompt Engineering
The process of designing effective prompts to guide LLMs to produce better results.

13. System Prompt
A special instruction that defines the behavior or personality of an LLM assistant.

Gowtham SB
www.linkedin.com/in/sbgowtham/

14. Temperature
Controls the randomness of AI output. Lower values = focused; higher values = more creative.

15. Top-k / Top-p Sampling
Techniques to control how LLMs pick the next word/token from possible options.

16. API Call
A request made to a model or service (like OpenAI API) to get a generated response.

17. Guardrails
Rules or filters to prevent unsafe, biased, or incorrect AI outputs.

18. Context Window
The maximum number of tokens a model can consider in a single interaction.

19. Prompt Template
Reusable prompts with placeholders that can be filled dynamically.

20. Memory
Stores past interactions so the model can use previous context in multi-turn conversations.

21. Tool Use / Tool Calling
When an LLM uses external tools (like code interpreters or databases) to complete a task.

22. Function Calling
A feature that lets the LLM decide when and how to call a backend function with structured arguments.

23. Autonomous Agent
An AI that runs with minimal human input, using planning, feedback, and memory.

24. Hallucination
When an AI generates factually incorrect or made-up information confidently.

25. Prompt Injection
A security issue where attackers trick the model into ignoring previous instructions.

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 1: LLM-based Data Quality Checker

---

1. Problem Statement

Build a **Gen AI agent** that can **analyze a CSV file** and **generate a natural language report** about the file's **data quality issues** — like missing values, duplicate rows, outliers, wrong data types, etc.

---

2. Why It Matters

In real-world projects, **bad data = bad analytics**.
Data engineers usually write manual scripts to check quality.
With this project, **an LLM will automate** the boring initial analysis.

---

3. Architecture Diagram

User Uploads CSV
↓
Python (FastAPI Backend) Reads File
↓
Summarizes data stats (like missing %, outliers) using Pandas
↓
LLM (local or OpenAI) Generates a human-readable Data Quality Report
↓
Returns the Report to User

4. Tech Stack

Item	Tool
Backend	Python + FastAPI

Gowtham SB
www.linkedin.com/in/sbgowtham/

Data Processing	Pandas
LLM	OpenAI (gpt-3.5-turbo) or local model (if needed)
Frontend (Optional)	Simple HTML upload or Postman

---

5. Step-by-Step Implementation

Step 1: Install dependencies

pip install fastapi uvicorn pandas openai python-multipart

Step 2: Create FastAPI app (`main.py`)

```
# main.py
from fastapi import FastAPI, UploadFile, File
import pandas as pd
import openai

app = FastAPI()

# Set your OpenAI key
openai.api_key = "YOUR_OPENAI_API_KEY"

@app.post("/analyze_csv/")
async def analyze_csv(file: UploadFile = File(...)):
    # Step 1: Read the uploaded file
    df = pd.read_csv(file.file)

    # Step 2: Generate simple quality statistics
    report = ""
    report += f"Total Rows: {len(df)}\n"
    report += f"Total Columns: {len(df.columns)}\n\n"

    # Missing value percentage
    missing_report = df.isnull().mean() * 100
    report += "Missing Values (% per column):\n"
    report += missing_report.to_string()
    report += "\n\n"

    # Duplicate rows
    duplicate_count = df.duplicated().sum()
    report += f"Duplicate Rows: {duplicate_count}\n\n"
```

```
    # Data types
    report += "Data Types:\n"
    report += df.dtypes.to_string()
    report += "\n\n"

    # Step 3: Send to LLM to generate a nicer summary
    prompt = f"Given this data quality raw report:\n{report}\nWrite a human-readable summary of the data quality issues and suggest 2-3 improvements."

    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
          {"role": "system", "content": "You are a helpful data quality analyst."},
          {"role": "user", "content": prompt}
        ]
    )

    final_summary = response['choices'][0]['message']['content']

    return {"data_quality_summary": final_summary}
```

Step 3: Run the server

```
uvicorn main:app --reload
```

Step 4: Test it

- Open Postman
- POST to `http://127.0.0.1:8000/analyze_csv/`
- Upload a CSV file
- Get back a **beautiful AI-written data quality report**!

---

6. Code Snippet Highlights

- Reading CSV safely
- Calculating missing percentages
- Asking LLM for **natural language transformation**

---

7. Challenges and Tips

- If CSV is too large (>10MB), you should **sample** first 5000 rows.
- Always validate CSV file format (only allow `.csv` extension).
- Set OpenAI usage **token limits** if costs matter.

---

8. Extensions (Next Level Ideas)

- Show a **Data Quality Score** (out of 100)
- Build a simple **frontend** for upload + report display
- Add **Excel support** (`.xlsx`) too
- Integrate local models like **Llama 3**, **Mistral** instead of OpenAI

Final Output Example

User uploads: `sales_data.csv`

Response generated:

> "The uploaded dataset contains 10,245 rows and 18 columns.

- Missing values were found in 4 columns, with 'Email' having the highest missing rate at 12.5%.
- There are 23 duplicate rows detected.
- The data types appear consistent, though 'PhoneNumber' is stored as float, which should ideally be string.

Recommendations:

1. Fill missing values in Email or drop those records.
2. Remove duplicate rows to avoid double-counting.
3. Correct the data type of PhoneNumber."

Summary

This project is super **practical** because:

- Every data engineer does **data quality checks**.
- LLM saves hours by **generating readable reports**.
- Extensible to many types of datasets.

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 2: AI SQL Generator

---

1. Problem Statement

Build a **Gen AI agent** that can convert **natural language questions** into **SQL queries** that can work with databases like **Redshift**, **BigQuery**, or **Snowflake**.

---

2. Why It Matters

Many analysts and business users find SQL difficult. This tool empowers them to query data using plain English, reducing dependency on data engineers.

---

3. Architecture Diagram

User inputs question in English
↓
FastAPI backend receives the input
↓
LLM (OpenAI or local model) converts input to SQL
↓
SQL returned to user (or optionally run on database)

---

4. Tech Stack

Item	Tool
Backend	Python + FastAPI
LLM	OpenAI (gpt-3.5-turbo) or local model
Optional DB	BigQuery, Redshift, Snowflake
Optional Execution	SQLAlchemy

---

5. Step-by-Step Implementation

Step 1: Install dependencies

Gowtham SB
www.linkedin.com/in/sbgowtham/
pip install fastapi uvicorn openai pydantic

Step 2: Create FastAPI app (`main.py`)

```
from fastapi import FastAPI
from pydantic import BaseModel
import openai

app = FastAPI()
openai.api_key = "YOUR_OPENAI_API_KEY"

class QueryRequest(BaseModel):
    question: str

@app.post("/generate_sql/")
async def generate_sql(req: QueryRequest):
    prompt = f"Convert this to a SQL query: {req.question}"
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "You are an expert SQL assistant."},
            {"role": "user", "content": prompt}
        ]
    )
    return {"sql_query": response['choices'][0]['message']['content']}
```

Step 3: Run the server

```
uvicorn main:app --reload
```

Step 4: Test the API

Use Postman or cURL to send a POST request:

```
{
  "question": "Show me total sales for each region in 2023"
}
```

Response:

```
SELECT region, SUM(sales)
FROM sales_data
WHERE YEAR(date) = 2023
```

GROUP BY region;

---

6. Code Snippet Highlights

- Dynamic LLM prompting
- Stateless API call with simple pydantic validation

---

7. Challenges and Tips

- Add schema info to prompt if accuracy drops
- Validate generated SQL before execution
- Limit permissions if running queries on real DB

---

8. Extensions (Next Level Ideas)

- Allow users to click and run the generated SQL on real database
- Add table schema preview to improve LLM context
- Support for JOIN, CTEs, and nested queries
- Token-level logging for cost control

Final Output Example

Input: “Show me average delivery time for each city in 2024.”

Response:

SELECT city, AVG(delivery_time)
FROM orders

Gowtham SB
www.linkedin.com/in/sbgowtham/
WHERE YEAR(order_date) = 2024
GROUP BY city;

Project 3: Data Pipeline Auto-Documenter

---

1. Problem Statement

Build a Gen AI agent that automatically reads your Airflow DAGs (or any pipeline code) and generates easy-to-understand documentation explaining each task, dependency, and scheduling logic.

---

2. Why It Matters

Maintaining pipeline documentation is boring and often skipped. This project automates that, making pipelines easier to understand and debug for new team members.

---

3. Architecture Diagram

User uploads or points to Airflow DAGs (Python scripts)
↓
Python backend reads the DAG files
↓
Summarizes task names, schedules, dependencies
↓
LLM converts it into human-readable documentation
↓
Returns Markdown or PDF output to user

---

4. Tech Stack

Item	Tool
Code Parsing	Python AST / regex
Backend	FastAPI

LLM	OpenAI (gpt-3.5-turbo) or local model
Optional Output	Markdown or PDF

---

5. Step-by-Step Implementation

Step 1: Install dependencies

```
pip install fastapi openai pydantic uvicorn
```

Step 2: FastAPI App (main.py)

```
from fastapi import FastAPI, UploadFile, File
import openai

app = FastAPI()
openai.api_key = "YOUR_API_KEY"

@app.post("/upload_dag/")
async def document_dag(file: UploadFile = File(...)):
    content = await file.read()
    dag_code = content.decode("utf-8")

    prompt = f"""
You are a data engineering documentation assistant.
Given this Airflow DAG code, write documentation that explains:
- What the DAG does
- Key tasks and dependencies
- The schedule and trigger rules

Here is the DAG code:
{dag_code}
"""

    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}]
    )

    doc = response['choices'][0]['message']['content']
    return {"documentation": doc}
```

Step 3: Run the server

uvicorn main:app --reload

Step 4: Upload a sample DAG

Use Postman to send a `.py` DAG file or upload via frontend.

---

6. Code Snippet Highlights

- Uses file upload and reads DAG content
- Sends entire script to LLM for interpretation

---

7. Challenges and Tips

- Long DAGs might exceed token limits — truncate or summarize parts
- Provide task docstrings in DAGs to improve output quality
- Secure API: Don't expose sensitive files

---

8. Extensions (Next Level Ideas)

- Add support for Prefect, dbt, or Kedro
- Export output as PDF/Markdown
- Create version history for documentation
- Allow voice-based pipeline explanation via TTS

Final Output Example

Upload: `daily_sales_pipeline.py`

Response:

> This DAG is called `daily_sales_pipeline` and it runs daily at midnight. It first extracts sales data from the S3 bucket, loads it to staging, transforms it using Spark, and then loads the cleaned data to Redshift. The final task sends a Slack notification. Each task depends on the successful completion of the previous.

Project 4: Chatbot for Data Lake Queries

---

1. Problem Statement

Create a Gen AI chatbot that lets users ask questions like "How many files are in the S3 bucket for January?" and receive responses by executing underlying cloud commands.

---

2. Why It Matters

Cloud storage like S3 can be huge and complex. This tool makes it easier for non-engineers to query storage content without knowing AWS CLI or Athena.

---

3. Architecture Diagram

User enters natural language question
↓
FastAPI backend receives request
↓
LLM interprets and translates question to S3/CLI command
↓
Python runs AWS SDK (boto3) command and gets result
↓
LLM converts response to natural summary
↓
Returns answer to user

---

4. Tech Stack

Gowtham SB
www.linkedin.com/in/sbgowtham/

Item	Tool
Backend	Python + FastAPI
LLM	OpenAI or Local Model (Mistral)
Cloud	AWS S3 (via boto3)
Optional	Streamlit UI for chatbot interface

---

5. Step-by-Step Implementation

Step 1: Install dependencies

```
pip install fastapi boto3 openai uvicorn
```

Step 2: Code (`main.py`)

```
from fastapi import FastAPI
from pydantic import BaseModel
import boto3, openai

app = FastAPI()
openai.api_key = "YOUR_API_KEY"
s3 = boto3.client('s3')

class Question(BaseModel):
    query: str

@app.post("/ask/")
async def ask_query(data: Question):
    prompt = f"Translate this to an AWS S3 Python (boto3) code snippet: {data.query}"
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    boto_code = response['choices'][0]['message']['content']

    # Caution: avoid exec in prod — just simulate output here
    result = "Simulated AWS output"

    summary_prompt = f"Here's the AWS response: {result}. Give a short answer."
    summary = openai.ChatCompletion.create(
```

```
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": summary_prompt}]
    )
    return {"answer": summary['choices'][0]['message']['content']}
```

---

6. Code Snippet Highlights

- Uses LLM to generate boto3 code
- Uses second LLM call to simplify raw AWS output

---

7. Challenges and Tips

- Avoid using `exec()` with generated code unless sandboxed
- Use predefined templates instead of raw generation if possible
- Add IAM policy checks for security

---

8. Extensions (Next Level Ideas)

- Add Athena SQL support
- Voice-based question input
- Use LangChain agent with tool calling
- Integrate Slack or MS Teams chatbot

Final Output Example

Input: “How many files in `s3://data-2023/jan`?”

Response: “There are 154 files in the folder `data-2023/jan`.”

Project 5: Schema Drift Detector using LLM

---

1. Problem Statement

Build a Gen AI tool that compares two versions of table schemas and identifies structural changes like column additions, deletions, renaming, or data type changes.

---

2. Why It Matters

Schema drift is a silent killer in data pipelines. Unnoticed changes can break ETL jobs, affect reporting, or cause data loss. This tool gives early warnings.

---

3. Architecture Diagram

---

4. Tech Stack

Item	Tool
Backend	Python + FastAPI
File Format	JSON or CSV schema dump
LLM	GPT-3.5, Claude, or Llama 3

---

5. Step-by-Step Implementation

Step 1: Install dependencies
pip install fastapi pydantic openai uvicorn

Step 2: FastAPI code (`main.py`)
from fastapi import FastAPI, UploadFile, File
import json, openai

```
app = FastAPI()
openai.api_key = "YOUR_KEY"

@app.post("/schema-drift")
async def detect_drift(old_file: UploadFile = File(...), new_file: UploadFile = File(...)):
    old_schema = json.loads((await old_file.read()).decode("utf-8"))
    new_schema = json.loads((await new_file.read()).decode("utf-8"))

    prompt = f"Compare the following two table schemas and explain changes:
Old Schema: {old_schema}
New Schema: {new_schema}"

    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    return {"drift_report": response['choices'][0]['message']['content']}
```

---

6. Challenges and Tips

- Limit schema to <100 columns to avoid token overload
- Use JSON schema exports for consistent structure
- Use LLM to summarize complex diffs understandably

---

7. Extensions (Next Level Ideas)

- Schedule to run on daily schema snapshots
- Integrate with Git or Airflow CI/CD
- Alert via Slack/Email

Final Output Example

2 columns were added (`created_at`, `updated_by`), and `user_id` was renamed to `account_id`. Data type of `amount` changed from INT to FLOAT.

Project 6: Data Lineage Mapper

---

1. Problem Statement

Build a Gen AI-powered tool that maps column-level lineage from ETL SQL scripts and pipeline code.

---

2. Why It Matters

Lineage is critical for debugging, compliance, and impact analysis. Manual lineage tracing is slow. LLMs can extract it automatically.

---

3. Architecture Diagram

---

4. Tech Stack

Item	Tool
Code	SQL, PySpark, dbt, Airflow
Backend	Python + FastAPI
LLM	OpenAI, Claude, or Mistral

---

5. Implementation Sample

```
prompt = f"""
Given this SQL:
{sql_code}

Show column-level lineage (source -> target column mapping)
```

Gowtham SB
www.linkedin.com/in/sbgowtham/
"""

Send this prompt to LLM, parse result and display mapping.

---

6. Challenges

- Complex joins may confuse LLM
- CTEs or nested SQL may need flattening first

---

7. Extensions

- Visualize as DAG or flowchart
- Integrate into data catalog
- Support multi-hop lineage

Output Sample

Source Column	Target Column
`sales.price`	`final_price`
`orders.id`	`order_id`

Project 7: Auto PII Identifier

---

1. Problem Statement

Create a tool that scans datasets and flags columns likely to contain PII (Personally Identifiable Information) such as emails, names, phone numbers, etc.

Gowtham SB
www.linkedin.com/in/sbgowtham/

---

2. Why It Matters

Data privacy laws (GDPR, HIPAA) require organizations to know where PII resides. Manual scanning is slow; LLMs can identify patterns automatically.

---

3. Architecture Diagram

---

4. Tech Stack

Item	Tool
Backend	FastAPI
Parsing	Pandas
LLM	GPT-3.5 / Llama / Claude
File Types	CSV, Parquet

---

5. Code Sketch

```
columns = df.columns.tolist()
sample_values = df.head(5).to_dict()
prompt = f"""
Identify which of these columns likely contain PII:
{columns}
Sample Data: {sample_values}
"""
```

Send to LLM and return flagged results.

---

6. Challenges

- Model might over-flag columns with generic names
- Always confirm LLM decisions with rules-based scan

---

7. Extensions

- Add regex-based fallback logic
- Highlight risk levels (High/Medium/Low)
- Build visual dashboard for PII auditing

Output Example

Column	PII Likely?	Type
email	Yes	Email ID
dob	Yes	Date of Birth
salary	No	Not PII

Project 8: Data Catalog Summarizer

---

1. Problem Statement

Create a Gen AI agent that reads metadata from thousands of data tables and generates human-readable summaries — including table purpose, data types, and relationships — for building automated data catalogs.

---

2. Why It Matters

Modern data platforms contain hundreds or thousands of tables. Manual documentation is slow and error-prone. Automating catalog summaries improves discoverability and data literacy.

---

3. Architecture Diagram

Metadata of Tables (Schema, Sample Rows)

Gowtham SB
www.linkedin.com/in/sbgowtham/
↓
Python backend prepares and filters input
↓
LLM generates human-friendly summary per table
↓
Stores in markdown, PDF, or data catalog (e.g., Amundsen)

---

4. Tech Stack

Component	Tool
Backend	Python + FastAPI
Input	JSON/CSV metadata exports
LLM	GPT-3.5, Claude, LLaMA 3
Optional Output	Markdown, Confluence, PDF, Notion

---

5. Step-by-Step Implementation

Step 1: Prepare metadata export

- For each table, gather:
    - Column names & types
    - Sample rows (first 5)
    - Table name & description (if any)

Step 2: Prompt example

```
prompt = f"""
Given this table metadata:
Table: {table_name}
Columns: {columns}
Sample Rows: {sample_rows}

Write a plain-English summary of what this table represents.
"""
```

Step 3: Generate via LLM

```
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[{"role": "user", "content": prompt}]
)
summary = response['choices'][0]['message']['content']
```

---

6. Tips

- If schema is long, focus only on column names and skip rows
- Use chunking if table metadata is large
- Save summaries per table for search integration

---

7. Extensions

- Embed summaries into DataHub, Amundsen, or custom catalog
- Add visual lineage based on inferred relationships
- Use streamlit UI for search and review

Final Output Example

> **Table: customer_orders** This table stores individual purchase transactions made by customers. It includes customer ID, order details, total amount, and order status. It is updated daily and used for downstream billing and analytics.

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 9: Smart ETL Code Reviewer

---

1. Problem Statement

Build a Gen AI agent that reviews ETL code (PySpark, SQL, Airflow, etc.) and provides suggestions for optimization, readability, modularity, and performance improvements.

---

2. Why It Matters

ETL pipelines often get bloated, hard to debug, and inefficient. Code reviews catch issues, but human reviews are slow and inconsistent. An LLM can help provide near-instant feedback.

---

3. Architecture Diagram

User pastes ETL code (PySpark / SQL / Python)
↓
FastAPI backend reads the code
↓
LLM analyzes structure, logic, anti-patterns
↓
Returns a natural language review with suggestions

---

4. Tech Stack

Component	Tool
Interface	FastAPI or Streamlit
LLM	GPT-4 / Claude / Llama 3
Input Code	SQL, PySpark, dbt, Python

---

5. Step-by-Step Implementation

Step 1: Prompt Template

```
prompt = f"""
Act like a senior data engineer.
Review the following ETL code and provide feedback on:
1. Performance improvements
2. Readability
3. Modular design
4. Best practices

Code:
{etl_code}
"""
```

Step 2: Pass to LLM and return feedback

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- If code is large, split by function/task
- Tag known frameworks (PySpark, dbt) in prompt for better context
- Keep logs to track repeated issues

---

7. Extensions

- Add linting suggestions via flake8 or pylint
- Export review as markdown
- Use LangChain agents to chain review + refactor actions

Gowtham SB
www.linkedin.com/in/sbgowtham/

Final Output Example

Review:

- Avoid using `.collect()` in PySpark; use `.show()` or write to file instead
- Consider modularizing repeated transformation logic
- Use `.alias()` for readability of complex expressions

Project 10: DAG Creation Assistant

---

1. Problem Statement

Create a Gen AI assistant that takes a plain English description of a data pipeline and automatically generates a valid Airflow DAG (Python code) with tasks, dependencies, and scheduling.

---

2. Why It Matters

Writing DAGs can be repetitive and error-prone for new users. Automating it with an LLM improves productivity, especially for teams defining many small workflows.

---

3. Architecture Diagram

User types pipeline workflow in plain English
↓
FastAPI backend receives and parses input
↓
LLM generates Airflow-compatible Python DAG code
↓
Returns final DAG file to download or copy

---

4. Tech Stack

Gowtham SB
www.linkedin.com/in/sbgowtham/

Component	Tool
Backend	Python + FastAPI
LLM	GPT-3.5 / GPT-4 / Claude
Output Format	Airflow DAG (Python file)

---

5. Step-by-Step Implementation

Step 1: Define the prompt

```
prompt = f"""
Generate an Airflow DAG based on the following description:
"{user_input}"

Include default_args, scheduling, task definitions, and task dependencies.
"""
```

Step 2: Call LLM and return output

```
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[{"role": "user", "content": prompt}]
)
code = response['choices'][0]['message']['content']
```

---

6. Tips

- Add sample templates to fine-tune output style
- Handle syntax errors gracefully by validating DAG code
- Keep description format structured (steps, schedule, etc.)

---

7. Extensions

- Add UI to describe workflows visually

- Export as `.py` or zip bundle for deployment
- Allow inline comments from user

Final Output Example

Input:

> "Run an ETL job every morning at 5 AM. First extract sales from S3, then transform using Spark, then load to Redshift."

Output: (Python snippet)

```
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 1, 1),
    'retries': 1
}

with DAG('etl_sales_pipeline', default_args=default_args, schedule_interval='0 5 * * *') as dag:
    extract = BashOperator(...)
    transform = SparkSubmitOperator(...)
    load = RedshiftOperator(...)

    extract >> transform >> load
```

Project 11: Real-time Data Anomaly Detector

---

1. Problem Statement

Build a Gen AI system that receives real-time metrics or logs and detects anomalies (spikes, drops, missing values) while also explaining the possible root cause in plain English.

---

2. Why It Matters

Gowtham SB
www.linkedin.com/in/sbgowtham/
Data issues are hard to debug when they happen silently. This project helps teams react quickly by detecting unexpected changes and letting LLMs explain what might be going wrong.

---

3. Architecture Diagram

Streaming data (logs / metrics / Kafka)
↓
Python backend detects anomalies with rules or stats
↓
LLM receives raw metrics + metadata
↓
Generates a human-readable explanation of the issue

---

4. Tech Stack

Component	Tool
Stream Source	Kafka / Log files / API
Anomaly Engine	Z-score, IsolationForest, or custom rules
LLM	GPT-4 / Claude / Mistral
Output	Slack / Dashboard / Console

---

5. Implementation Flow

Step 1: Define rules or use Z-score

```
import numpy as np
z_scores = np.abs((values - values.mean()) / values.std())
anomalies = values[z_scores > 2]
```

Step 2: Send anomaly info to LLM

```
prompt = f"""
We detected a spike in `conversion_rate` from 3.1% to 7.8%.
The user count dropped during the same period.
Explain possible reasons.
"""
```

Step 3: Return natural-language summary

> "The anomaly could be due to bot traffic inflating conversion artificially. The drop in user count supports this. Also check for recent campaign tagging issues."

---

6. Tips

- Add thresholds for ignoring micro-variations
- Add time context in LLM prompts (e.g., compare to past 1h/24h)
- Include known incidents or config changes in metadata

---

7. Extensions

- Integrate with Grafana or Kibana
- Add self-healing workflows (e.g., trigger rollback)
- Store LLM outputs for RCA documentation

Final Output Example

> "High drop in event count due to pipeline ingestion lag. Logs show retries and Kafka consumer lag for topic `event_logs`."

Project 12: Semantic Search for Databases

---

1. Problem Statement

Create a Gen AI-powered semantic search system that lets users query across tables, schemas, and documentation using natural language instead of exact column or table names.

Gowtham SB
www.linkedin.com/in/sbgowtham/

---

2. Why It Matters

Traditional keyword-based search fails when users don't know exact column names or spelling. Semantic search improves data discovery and trust in the data platform.

---

3. Architecture Diagram

User types natural language query
↓
LLM generates semantic vector embedding
↓
Metadata and table samples are also embedded
↓
Cosine similarity finds most relevant tables or fields
↓
Returns matched results with explanation

---

4. Tech Stack

Component	Tool
Embedding Model	OpenAI embeddings / HuggingFace transformers
Vector Store	FAISS / Pinecone / Weaviate
Metadata Source	BigQuery, Redshift, Hive Metastore
Backend	Python + FastAPI + LangChain

---

5. Step-by-Step Implementation

Step 1: Embed metadata

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")

columns = ["customer_id", "total_sales", "signup_date"]
embeddings = model.encode(columns)
```

Step 2: Store in FAISS

```
import faiss
index = faiss.IndexFlatL2(embeddings[0].shape[0])
index.add(embeddings)
```

Step 3: Handle user query

```
query = "which table has user purchase data?"
query_vec = model.encode([query])
distances, indices = index.search(query_vec, k=3)
```

---

6. Tips

- Include column descriptions and sample values in context
- Normalize naming conventions (underscores, camelCase)
- Add source table ranking heuristics (e.g., recent use)

---

7. Extensions

- Integrate into dbt docs or open metadata UI
- Return SQL previews from matched tables
- Add feedback loop to fine-tune search results

Final Output Example

Input: "Where do we store customer emails and signup time?"

Response:

- Table: `customer_profile`

- Columns: `email`, `created_at`

- Description: Stores registered users and their signup metadata

---

Project 13: API Generator from Data

---

1. Problem Statement

Build a Gen AI system that takes a dataset and generates a working REST API to serve it, allowing users to query it using HTTP calls.

---

2. Why It Matters

APIs make datasets accessible across teams and apps. Creating them manually is tedious. An LLM-powered assistant can automate most of it, including query filters, endpoints, and documentation.

---

3. Architecture Diagram

User uploads dataset (CSV or JSON)
↓
Python backend reads data schema
↓
LLM generates FastAPI endpoint code
↓
User receives downloadable API project

---

4. Tech Stack

Component	Tool
Backend	Python + FastAPI
LLM	GPT-3.5 / GPT-4 / Claude

Gowtham SB
www.linkedin.com/in/sbgowtham/

Dataset Formats	CSV / JSON / Parquet
Output	Auto-generated Python API code

---

5. Step-by-Step Implementation

Step 1: Upload and inspect data

```
import pandas as pd
file = pd.read_csv("sales.csv")
columns = list(file.columns)
```

Step 2: Prompt example

```
prompt = f"""
Generate FastAPI code for a REST API that serves a dataset with columns:
{columns}

Include endpoints for fetching all records, filtering by any column, and paginated results.
"""
```

Step 3: Run with LLM

```
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[{"role": "user", "content": prompt}]
)
api_code = response['choices'][0]['message']['content']
```

---

6. Tips

- Add validations for column types
- Support both GET and POST APIs
- Export response in JSON or CSV

---

7. Extensions

- Allow schema-to-SQL generation
- Add OpenAPI / Swagger documentation
- Generate Dockerfile for containerized deployment

Final Output Example

```
GET /data?region=APAC&min_sales=5000&page=2
```

Returns 10 records filtered from the dataset with pagination.

Project 14: Log Analyzer with LLM

---

1. Problem Statement

Create a Gen AI tool that analyzes server or application logs and summarizes errors, warnings, and performance bottlenecks with plain English explanations.

---

2. Why It Matters

Log files are often massive and hard to interpret. Automating their analysis with LLMs helps in faster debugging, monitoring, and system reliability.

---

3. Architecture Diagram

Upload or stream log file
↓
Parser extracts key logs (errors, warnings, timestamps)
↓
LLM summarizes logs in structured report
↓

Gowtham SB
www.linkedin.com/in/sbgowtham/
Returns readable explanation and insights

---

4. Tech Stack

Component	Tool
Parser	Regex / Log parser libraries
LLM	GPT-3.5 / Claude / Mistral
Interface	Python + FastAPI or Streamlit
Output Format	HTML, JSON, or Markdown

---

5. Step-by-Step Implementation

Step 1: Extract important parts of logs

```
with open("server.log", "r") as f:
    log_lines = f.readlines()
errors = [line for line in log_lines if "ERROR" in line or "WARN" in line]
```

Step 2: Prompt the LLM

```
prompt = f"""
Analyze these logs and explain any errors or issues in simple English:
{errors[:100]}
"""
```

Step 3: Get insights

```
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- Limit token size by sending top errors or last 200 lines

- Include system metadata (OS, service version) for better context

---

7. Extensions

- Auto-group errors by type or component
- Add time-series chart using Matplotlib
- Connect to log streaming services like CloudWatch or ELK

Final Output Example

> "Between 10:00–10:15 AM, the app experienced high latency and 504 gateway timeout errors due to Redis being unreachable. Suggest checking connection pooling settings."

Project 15: Data Pipeline Cost Estimator

---

1. Problem Statement

Build a Gen AI agent that estimates the cost of running a data pipeline based on its configuration, data volume, and cloud platform (AWS, GCP, or Azure).

---

2. Why It Matters

Cloud costs can escalate quickly without proper visibility. This tool allows data engineers and managers to estimate costs before deploying large-scale workflows.

---

3. Architecture Diagram

User describes pipeline (steps, services, data size)

↓

Backend parses pipeline components and usage estimates
↓
LLM evaluates pricing model for chosen cloud provider
↓
Returns a cost breakdown by component

---

4. Tech Stack

Component	Tool
Backend	Python + FastAPI
LLM	GPT-3.5 / Claude / Mistral
Cost API (Optional)	AWS Pricing API, GCP Cloud Billing
Output	JSON or Markdown Cost Summary

---

5. Step-by-Step Implementation

Step 1: Collect Inputs

- Number of jobs / DAG tasks
- Data size per task (e.g., 1 TB S3 read)
- Services used (e.g., Glue, BigQuery, Dataflow)

Step 2: Prompt Template

```
prompt = f"""
Estimate the cloud cost of this pipeline:
- 3 jobs on AWS Glue (10GB each)
- 1 Redshift load (500GB)
- 2 Lambda functions (runs 10,000 times/month)
Return a cost breakdown.
"""
```

Step 3: Get Estimate

```
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
```

```
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- Keep cloud pricing reference sheets updated
- Allow users to input custom price overrides
- Normalize to monthly or daily cost

---

7. Extensions

- Add charts to visualize cost trends
- Suggest cheaper service alternatives (Athena vs Redshift)
- Allow scenario comparison (Option A vs Option B)

Final Output Example

Pipeline Cost Summary:

- AWS Glue: $0.96
- Redshift: $3.50
- Lambda: $1.25
- **Total Monthly Cost: $5.71**

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 16: SLA Breach Predictor

---

1. Problem Statement

Build a Gen AI-based tool that predicts whether a data pipeline is likely to breach its SLA based on historical run data, delays, dependencies, and seasonal variations.

---

2. Why It Matters

SLAs are critical for business reports and downstream dependencies. Predicting SLA breaches helps in proactive monitoring, alerting, and resource optimization.

---

3. Architecture Diagram

Historical pipeline run data (duration, start, end, failure)
↓
Preprocessing & feature engineering
↓
Model (LLM + ML) predicts likelihood of breach
↓
Explanation returned via LLM + alert if risk is high

---

4. Tech Stack

Component	Tool
Data	Airflow Logs, ETL metadata, time series
Model	XGBoost / Prophet + LLM (GPT/Claude)
Backend	Python + FastAPI
Alerting	Slack / Email / Webhook

Gowtham SB
www.linkedin.com/in/sbgowtham/

5. Step-by-Step Implementation

Step 1: Collect and preprocess data

```
features = df[['dag_id', 'start_time', 'end_time', 'delay_minutes', 'success']]
features['duration'] = (features['end_time'] - features['start_time']).dt.total_seconds()
```

Step 2: Predict breach

```
# Simple rule or ML classifier
if delay_minutes > threshold:
    risk = 'HIGH'
else:
    risk = 'LOW'
```

Step 3: Send to LLM for summary

```
prompt = f"Pipeline `{dag_id}` is predicted to miss SLA by 30 mins. Suggest root causes and mitigation."
```

---

6. Tips

- Include public holiday calendar and cron schedule as inputs
- Use LLM to rewrite tech alerts into business-friendly summaries

---

7. Extensions

- Train time series models (ARIMA, Prophet) on delays
- Auto-throttle resource usage if breach risk is high
- Connect to pipeline orchestrators like Airflow or Dagster

Final Output Example

> "DAG `monthly_sales_refresh` has a 78% chance of breaching SLA due to prior slowdowns during month-end loads and increased warehouse usage. Recommend running earlier or increasing cluster size."

Project 17: LLM-enhanced Feature Store

---

1. Problem Statement

Build a Gen AI integration with your feature store that allows users to query, understand, and search features in natural language.

---

2. Why It Matters

Feature stores often contain hundreds of engineered features. Understanding their origin, logic, or usage takes time. LLM can summarize them for faster onboarding and reuse.

---

3. Architecture Diagram

User enters question about features ("what features contain churn?" or "explain feature X")
↓
LLM retrieves relevant metadata and feature definitions
↓
Generates plain English explanations and usage hints
↓
Returns response to user via UI or API

---

4. Tech Stack

Component	Tool
Feature Store	Feast, Vertex AI Feature Store
LLM	GPT-4 / Claude / Mistral
Backend	Python + FastAPI

Vector Store	FAISS / Pinecone (optional for semantic search)

---

5. Implementation Steps

Step 1: Export feature metadata

- Feature name
- Description
- Transformation logic
- Last used model or frequency

Step 2: Prompt structure

```
prompt = f"""
Here is a feature store entry:
Name: churn_prediction_ltv_30
Description: lifetime value of customer in last 30 days
Logic: sum(purchase_amount) / 30

Explain what this feature means, and when to use it.
"""
```

Step 3: Get explanation

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- Use LangChain to allow follow-up questions (e.g., "Where else is this used?")
- Embed all features for semantic similarity search

---

7. Extensions

- Connect to live search bar UI for feature discovery
- Include example model pipelines using the feature
- Add impact scores or usage graphs

Final Output Example

> **Feature:** churn_prediction_ltv_30 **Explanation:** This feature represents the average customer spend per day over the past 30 days. It is typically used in churn models to capture recent monetary behavior. Can be combined with engagement or session features.

Project 18: Data Migration Assistant

---

1. Problem Statement

Create a Gen AI agent that helps plan, monitor, and validate data migration tasks between data sources — such as from on-prem to cloud, or Redshift to BigQuery.

---

2. Why It Matters

Data migrations are complex and risky. LLMs can assist in planning steps, identifying dependencies, and even validating that post-migration data matches source systems.

---

3. Architecture Diagram

User defines source and target systems (schemas, sample data)

↓

LLM assists in planning steps and generating SQL/dataflow scripts

Gowtham SB
www.linkedin.com/in/sbgowtham/

↓

Migration script is validated with row-level comparisons

↓

Summary report provided via UI or email

---

4. Tech Stack

Component	Tool
Backend	Python + FastAPI
LLM	GPT-4 / Claude / Mistral
DB connectors	SQLAlchemy, BigQuery, Redshift, Snowflake drivers
Validation	Pandas / Spark for row mismatch check

---

5. Step-by-Step Implementation

Step 1: Define Migration Plan Prompt

```
prompt = f"""
I want to migrate a table from Redshift to BigQuery.
Schema: {schema_details}
Please give me a 5-step plan with possible pitfalls and SQL compatibility issues to watch for.
"""
```

Step 2: Use LLM to explain and generate copy logic

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

Step 3: Validate Row-Level Accuracy

```
# pandas compare row count or hash checksum
src_df = pd.read_sql("SELECT * FROM table", redshift_conn)
tgt_df = pd.read_sql("SELECT * FROM table", bq_conn)
mismatch = src_df.compare(tgt_df)
```

Gowtham SB
www.linkedin.com/in/sbgowtham/

6. Tips

- Validate both schema (column types) and data values
- Watch for datetime/timezone mismatches across platforms
- Support parallel loading for large datasets

---

7. Extensions

- Suggest automated DDL conversion
- Track logs of migrated vs skipped rows
- Email success/failure report

Final Output Example

Migration Plan:

- Step 1: Export Redshift data to S3
- Step 2: Format data as Parquet
- Step 3: Load into BigQuery using `bq load`
- Step 4: Validate row count and primary keys
- Step 5: Switch over reporting pipeline

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 19: OLAP Query Optimizer using LLM

---

1. Problem Statement

Build a Gen AI assistant that takes complex OLAP queries (from tools like Looker, Tableau, Redshift, or BigQuery) and suggests performance optimizations — including partition filters, join rewrites, and aggregation strategies.

---

2. Why It Matters

OLAP queries often process billions of rows. Minor changes in query structure or filter conditions can lead to major performance gains. LLMs can speed up this review process.

---

3. Architecture Diagram

User pastes OLAP SQL query
↓
LLM parses query for structure, joins, filters, and patterns
↓
Suggests optimized query with explanation
↓
Returns improved SQL and summary of impact

---

4. Tech Stack

Component	Tool
Backend	Python + FastAPI
LLM	GPT-4 / Claude / Mistral
Optional DB	BigQuery / Redshift / Snowflake
Frontend (optional)	Streamlit UI

Gowtham SB
www.linkedin.com/in/sbgowtham/

5. Step-by-Step Implementation

Step 1: Sample OLAP SQL

```
SELECT region, COUNT(*) FROM orders
JOIN customers ON orders.customer_id = customers.id
WHERE signup_date > '2023-01-01'
GROUP BY region;
```

Step 2: Prompt Template

```
prompt = f"""
Review the following OLAP SQL and suggest:
1. Filter or partition optimizations
2. Better JOIN or WHERE logic
3. Aggregation improvements
SQL:
{sql_query}
"""
```

Step 3: Generate Optimized SQL

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
optimized_sql = response['choices'][0]['message']['content']
```

---

6. Tips

- Include table metadata in context (e.g., partition column info)
- Log improvements in cost or estimated run time when possible

---

7. Extensions

- Add benchmark testing after suggestions
- Rank optimizations by expected impact (High/Medium/Low)

- Integrate with dbt or BI layer

Final Output Example

> "Moved filter on `signup_date` before JOIN to reduce join size. Suggested using `DISTINCT` before COUNT. Applied table partition filter for `orders` based on `order_date`."

Project 20: SQL Explainer Bot

---

1. Problem Statement

Create a Gen AI tool that explains complex SQL queries in plain English, breaking down the purpose of each clause, joins, filters, and aggregations for easy understanding.

---

2. Why It Matters

Many data consumers, analysts, and junior engineers struggle with reading complex SQL. A natural language explainer helps in faster onboarding and documentation.

---

3. Architecture Diagram

User inputs complex SQL query
↓
LLM parses structure and logic of the query
↓
Returns human-readable explanation with structure

---

4. Tech Stack

Component	Tool

Gowtham SB
www.linkedin.com/in/sbgowtham/

Interface	Streamlit or FastAPI
LLM	GPT-4 / Claude / Mistral
Optional Backend	LangChain + Prompt templates

---

5. Step-by-Step Implementation

Step 1: Sample SQL Input

```
SELECT c.name, COUNT(o.id) AS order_count
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE o.order_date >= '2023-01-01'
GROUP BY c.name
HAVING COUNT(o.id) > 5;
```

Step 2: Prompt for LLM

```
prompt = f"""
Explain this SQL query in simple English. Also include:
1. What tables are used?
2. What is being calculated?
3. Are there any filters or conditions?

SQL:
{sql_query}
"""
```

Step 3: Generate Explanation

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- Chunk long queries into CTEs and explain block by block

- Add color-coded display in frontend (e.g., highlight JOIN, GROUP BY)

---

7. Extensions

- Auto-generate documentation from SQL
- Voice-based input for SQL review on the go
- Translate SQL to flowcharts

Final Output Example

> **Explanation:** This query finds customer names and counts how many orders they placed since Jan 1, 2023. It only includes those who placed more than 5 orders. It joins `customers` and `orders` tables using customer ID.

Project 20: SQL Explainer Bot

---

1. Problem Statement

Create a Gen AI tool that explains complex SQL queries in plain English, breaking down the purpose of each clause, joins, filters, and aggregations for easy understanding.

---

2. Why It Matters

Many data consumers, analysts, and junior engineers struggle with reading complex SQL. A natural language explainer helps in faster onboarding and documentation.

Gowtham SB
www.linkedin.com/in/sbgowtham/

3. Architecture Diagram

User inputs complex SQL query

↓

LLM parses structure and logic of the query

↓

Returns human-readable explanation with structure

---

4. Tech Stack

Component	Tool
Interface	Streamlit or FastAPI
LLM	GPT-4 / Claude / Mistral
Optional Backend	LangChain + Prompt templates

---

5. Step-by-Step Implementation

Step 1: Sample SQL Input

```
SELECT c.name, COUNT(o.id) AS order_count
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE o.order_date >= '2023-01-01'
```

GROUP BY c.name

HAVING COUNT(o.id) > 5;

Step 2: Prompt for LLM

prompt = f"""

Explain this SQL query in simple English. Also include:

1. What tables are used?

2. What is being calculated?

3. Are there any filters or conditions?

SQL:

{sql_query}

"""

Step 3: Generate Explanation

response = openai.ChatCompletion.create(

  model="gpt-4",

  messages=[{"role": "user", "content": prompt}]

)

print(response['choices'][0]['message']['content'])

---

6. Tips

- Chunk long queries into CTEs and explain block by block

- Add color-coded display in frontend (e.g., highlight JOIN, GROUP BY)

---

7. Extensions

- Auto-generate documentation from SQL
- Voice-based input for SQL review on the go
- Translate SQL to flowcharts

Final Output Example

> **Explanation:** This query finds customer names and counts how many orders they placed since Jan 1, 2023. It only includes those who placed more than 5 orders. It joins `customers` and `orders` tables using customer ID.

Project 21: Synthetic Data Generator

---

1. Problem Statement

Create a Gen AI agent that generates synthetic datasets based on schema definitions and sample data, preserving realistic distributions and relationships.

---

2. Why It Matters

Synthetic data helps in testing, privacy-safe training, and debugging without exposing real customer data. LLMs can intelligently generate rows that "look like" real data.

---

3. Architecture Diagram

Gowtham SB
www.linkedin.com/in/sbgowtham/
User provides schema (column names + types) or sample data

↓

LLM uses patterns and examples to generate synthetic rows

↓

Returns downloadable CSV/Parquet file with fake data

---

4. Tech Stack

Component	Tool
Interface	FastAPI or Streamlit
LLM	GPT-4 / Claude / Mistral
Output Format	CSV / JSON / Parquet
Optional	Faker library, Pandas

---

5. Step-by-Step Implementation

Step 1: Input Example

```
{
  "columns": ["name", "email", "signup_date", "purchase_amount"],
  "types": ["string", "email", "date", "float"]
}
```

Step 2: Prompt the LLM

```
prompt = f"""
Generate 100 rows of synthetic data with the following schema:
Columns: {columns}
Types: {types}
Ensure the data looks realistic and reflects real-world patterns.
"""
```

Step 3: Generate Output

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
data = response['choices'][0]['message']['content']
```

---

6. Tips

- Use `Faker` for fields like names, addresses, etc.
- Use Pandas to format and validate generated output
- Add export options: CSV, JSON, Excel

---

7. Extensions

- Add noise injection or label flipping for ML testing
- Generate classification datasets with labels
- Support conditional data (e.g., generate based on date range)

Final Output Example

name	email	signup_date	purchase_amount
Jane Doe	jane@gmail.com	2023-04-10	120.50
John Lee	john@company.com	2023-03-01	89.90

Project 22: Data Lake Explorer Assistant

---

1. Problem Statement

Create a Gen AI-powered assistant that lets users navigate and explore large data lakes (like S3, GCS, or ADLS) using natural language commands.

---

2. Why It Matters

Data lakes contain massive folder structures with thousands of files. Searching or exploring them manually is difficult. AI can simplify discovery for both technical and non-technical users.

Gowtham SB
www.linkedin.com/in/sbgowtham/

3. Architecture Diagram

User types natural language command (e.g., "list all parquet files in January folder")

↓

LLM parses intent and generates code (boto3, gsutil, etc.)

↓

Backend executes and fetches results from data lake

↓

Returns result in readable format or JSON

---

4. Tech Stack

Component	Tool
Cloud Storage	AWS S3, GCS, ADLS
Access	Boto3 / Google Cloud SDK / Azure SDK
LLM	GPT-4 / Claude / Mistral
Interface	FastAPI or Streamlit

---

5. Step-by-Step Implementation

Step 1: User Query

"Show all folders in s3://data-lake/logs/2024/"

Step 2: Prompt Template

prompt = f"""

Generate Python boto3 code to list all folders and files under this S3 path: {user_input}. Just return the list, no delete or write operation.

"""

Step 3: Execute and Return Output

s3 = boto3.client('s3')

response = s3.list_objects_v2(Bucket='data-lake', Prefix='logs/2024/')

folders = set(obj['Key'].split('/')[2] for obj in response['Contents'])

---

6. Tips

- Always enforce read-only mode in generated code
- Add folder/file counts to each result
- Paginate results if folder is large

---

7. Extensions

- Add preview of file content (first 5 rows of CSV/JSON)
- Allow natural-language filtering: "Only list folders >100MB"
- Generate pre-signed URLs for secure preview/download

Final Output Example

Request: "List all parquet files in February 2024" **Response:**

- s3://data-lake/logs/2024/02/events-01.parquet
- s3://data-lake/logs/2024/02/errors-aggregated.parquet

Project 23: Security Risk Detector for Data Pipelines

---

1. Problem Statement

Build a Gen AI tool that analyzes data pipeline configurations and scripts (Airflow DAGs, Spark jobs, shell scripts) and detects potential security issues such as exposed secrets, unsafe permissions, or misconfigured IAM policies.

---

2. Why It Matters

Data pipelines often touch sensitive sources and destinations. Even a small misconfiguration can expose private data. This tool provides fast and intelligent static security reviews.

---

3. Architecture Diagram

User uploads pipeline config or code (YAML, Python, JSON)

↓

LLM reviews content with static rules and heuristics

↓

Returns a report highlighting risks, severity, and suggestions

---

4. Tech Stack

Component	Tool
Input Types	Python (Airflow), YAML (K8s), Terraform, Bash
LLM	GPT-4 / Claude / Mistral
Backend	FastAPI + Static Analysis Logic

---

5. Step-by-Step Implementation

Step 1: User Upload

file_content = uploaded_file.read().decode("utf-8")

Step 2: Prompt Example

prompt = f"""

Review the following Airflow DAG and highlight any security issues, such as exposed credentials, broad IAM roles, or missing retries.

Code:

{file_content}

"""

Step 3: LLM Response

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
report = response['choices'][0]['message']['content']
```

---

6. Tips

- Highlight `os.environ`, hardcoded strings, wide `*:*` permissions
- Check for retry settings, failure alerts, IAM role scoping

---

7. Extensions

- Add CVSS-style scoring
- GitHub integration to run on pull requests
- Fix suggestions with code examples

Final Output Example

- **Issue:** `aws_access_key_id` is hardcoded
- **Risk:** Secrets in code
- **Fix:** Use AWS Secrets Manager or environment variable injection

---

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 24: Business KPI Analyzer

---

1. Problem Statement

Create a Gen AI tool that takes input reports (Excel, CSV, JSON) and generates a natural-language summary of business KPIs such as revenue, churn, conversion rate, etc., highlighting trends, spikes, and anomalies.

---

2. Why It Matters

Decision-makers prefer insights over raw numbers. This tool helps turn tabular data into readable, insightful KPI summaries automatically — saving time for analysts.

---

3. Architecture Diagram

User uploads monthly report (CSV/Excel)

↓

Pandas summarizes KPIs using statistical rules

↓

LLM generates a business summary in plain English

↓

Output returned as HTML, Markdown, or email-ready report

---

4. Tech Stack

Component	Tool
Data Handling	Pandas
File Input	CSV, Excel
LLM	GPT-4 / Claude / Mistral
Output Format	Markdown / Email / PDF

---

5. Step-by-Step Implementation

Step 1: Ingest data file

```
import pandas as pd
file = pd.read_csv("report.csv")
summary = file.describe()
```

Step 2: Prepare prompt

```
prompt = f"""
Here's the monthly KPI summary:
{summary}
Generate a business summary with highlights on trends and outliers.
"""
```

Step 3: Send to LLM

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
kpi_summary = response['choices'][0]['message']['content']
```

---

6. Tips

- Include previous month comparison for better trend detection
- Add charts (bar, line, pie) to enhance visualization
- Add threshold rules (e.g., "if churn > 10%, flag it")

---

7. Extensions

- Auto-email the KPI report to leadership
- Integrate with BI dashboards for auto-summary
- Add multilingual support

Final Output Example

> **Summary:** Revenue increased by 12% MoM, primarily driven by growth in APAC. Churn rate slightly rose to 6.1%. Marketing spend efficiency improved by 9%.

Gowtham SB
www.linkedin.com/in/sbgowtham/

Project 25: Data Science Notebook Explainer

---

1. Problem Statement

Create a Gen AI assistant that can analyze Jupyter or Colab notebooks and generate a structured summary explaining each code block, its purpose, and the overall workflow.

---

2. Why It Matters

Notebooks are messy and undocumented by default. AI-generated walkthroughs improve code sharing, review, onboarding, and reproducibility in data science workflows.

---

3. Architecture Diagram

User uploads .ipynb notebook file

↓

Python backend parses code, markdown, and outputs

↓

LLM summarizes each cell with purpose and outputs

↓

Returns structured document or HTML walkthrough

---

4. Tech Stack

Component	Tool
File Parser	nbformat, nbconvert
LLM	GPT-4 / Claude / Mistral
Backend	FastAPI / Streamlit
Output	Markdown / PDF / Web HTML

---

5. Step-by-Step Implementation

Step 1: Read notebook file

```
import nbformat

nb = nbformat.read("model_training.ipynb", as_version=4)
```

Step 2: Extract code + markdown

```
cells = [(cell.cell_type, cell.source) for cell in nb.cells]
```

Step 3: Prompt LLM per cell or per block

```
for cell_type, content in cells:
    if cell_type == 'code':
        prompt = f"Explain the following Python code from a data science project:

{content}"
```

Step 4: Generate structured summary

```
response = openai.ChatCompletion.create(
  model="gpt-4",
  messages=[{"role": "user", "content": prompt}]
)
print(response['choices'][0]['message']['content'])
```

---

6. Tips

- Include inline comments in cells for better LLM results
- Add visual summaries: charts, dependencies, and feature flow

---

7. Extensions

- Export HTML-based walkthrough
- Link variables to plots or final outputs
- Add reviewer mode to critique and improve code

Final Output Example

Notebook Summary:

- Cell 1: Loads libraries
- Cell 2: Reads `sales.csv` and cleans missing values

- Cell 3: Trains a RandomForest and evaluates accuracy
- Cell 4: Plots feature importance using Seaborn

Gen AI resume for Data Engineers

Gowtham SB

📍 Chennai, India | 📞 +91-9876543210 | 📧 ravi.kumar@example.com | 🌐 linkedin.com/in/sbgowtham | github.com/

🚀 Career Objective

Results-driven Data Engineer with 5+ years of experience in building robust data pipelines, lakehouse systems, and cloud-native platforms. Recently pivoted into Generative AI with strong hands-on skills in LangChain, LLMs, prompt engineering, and RAG pipelines. Looking to contribute to an AI-first organization leveraging data and intelligence at scale.

🧠 Key Skills

- **Languages**: Python, SQL, PySpark, Bash
- **Data Engineering**: Airflow, Spark, Snowflake, BigQuery, Redshift, Kafka, dbt
- **Gen AI Tools**: OpenAI, LangChain, Pinecone, LlamaIndex, FAISS
- **AI/ML**: Hugging Face Transformers, Scikit-learn, Pandas, RAG architecture

Gowtham SB
www.linkedin.com/in/sbgowtham/

- **Cloud**: AWS (S3, Lambda, Glue), GCP (BigQuery, VertexAI), Azure (Data Factory)
- **DevOps**: Docker, GitHub Actions, CI/CD pipelines

🧰 Technical Certifications

- Google Cloud Professional Data Engineer
- DeepLearning.AI Generative AI with LLMs Specialization

🏢 Work Experience

Senior Data Engineer – Gen AI Focus
XXXXX
Feb 2022 – Present

- Designed and deployed a **retrieval-augmented generation (RAG)** architecture using LangChain, OpenAI, and Pinecone to power internal document Q&A system for 500+ employees.
- Migrated traditional dashboards to AI-powered **KPI summarizer** that ingests CSV reports and generates business reports using GPT-4.
- Built an **auto-metadata tagger** for Snowflake tables using schema descriptions + sample data + LLM inference.
- Mentored 3 junior engineers on transitioning from traditional pipelines to LLM-enhanced data workflows.

Gowtham SB
www.linkedin.com/in/sbgowtham/
Data Engineer
XXXXX
Jan 2019 – Jan 2022

- Developed batch + near real-time pipelines on GCP using BigQuery, Pub/Sub, Dataflow.
- Built customer churn features using PySpark + Airflow, saving 22% ETL runtime with dynamic partitioning.
- Integrated dbt for transformation layer with data quality tests and documented metrics.

🔧 **Gen AI Project Highlight**

📌 **Project: LLM-Based Data Quality Analyzer for Data Lakes**

- Built a Gen AI tool using FastAPI + OpenAI + Pandas that accepts CSV/Parquet files and returns:
    - Natural language summary of data quality issues (nulls, skewed data, outliers)
    - Suggestions for cleaning, fixing schemas, and possible reasons for anomalies
- Added Streamlit frontend with drag-drop uploader and exportable Markdown summary
- Used by QA team for validating incoming data in analytics pipeline

Tech Used: GPT-3.5, LangChain, FastAPI, Pandas, Streamlit, Docker

---

Gowtham SB
www.linkedin.com/in/sbgowtham/

🎓 Education

B.Tech – Information Technology
XXXXX | 2014 – 2018 | CGPA: 8.1 / 10

🏆 Achievements

- Built a side project: "TalkToData" — a GPT-powered SQL generator for PostgreSQL and BigQuery
- 1st place in internal LLM Hackathon (2023) for building an autonomous CSV assistant

---

📁 Additional Projects (GitHub)

- AI Resume Reviewer Bot (OpenAI + Streamlit)
- Vector Search Engine using FAISS + Local PDF RAG
- Chatbot for SQL Glossary via LangChain

Gowtham SB
www.linkedin.com/in/sbgowtham/

Target Audience

- Data Engineers who want to **add Gen AI to their skill set**
- Cloud Data Engineers (AWS, Azure, GCP)
- Data Platform Engineers
- AI Enthusiasts who understand data workflows
- Students who want portfolio projects

Gowtham SB
www.linkedin.com/in/sbgowtham/

About the Author

Gowtham SB is a **Data Engineering expert, educator, and content creator** with a passion for **big data technologies, as well as cloud and Gen AI** . With years of experience in the field, he has worked extensively with **cloud platforms, distributed systems, and data pipelines**, helping professionals and aspiring engineers master the art of data engineering.

Beyond his technical expertise, Gowtham is a **renowned mentor and speaker**, sharing his insights through engaging content on **YouTube and LinkedIn**. He has built one of the **largest Tamil Data Engineering communities**, guiding thousands of learners to excel in their careers.

Through his deep industry knowledge and hands-on approach, Gowtham continues to **bridge the gap between learning and real-world implementation**, empowering individuals to build **scalable, high-performance data solutions**.

Socials

🎥**YouTube** - https://www.youtube.com/@dataengineeringvideos

📸**Instagram** - https://instagram.com/dataengineeringtamil

📸**Instagram** - https://instagram.com/thedatatech.in

🤝**Connect for 1:1** - https://topmate.io/dataengineering/

💼**LinkedIn** - https://www.linkedin.com/in/sbgowtham/

🌐**Website** - https://codewithgowtham.blogspot.com

💻**GitHub** - http://github.com/Gowthamdataengineer

💬**Whats App** - https://lnkd.in/g5JrHw8q

📧**Email** - atozknowledge.com@gmail.com

📱 **All My Socials** - https://lnkd.in/gf8k3aCH