Retail Insights Assistant - Architecture Presentation

GenAI-Powered Multi-Agent System for 100GB+ Scale Retail Analytics

---

📑 Table of Contents

Slide 1: System Overview

Retail Insights Assistant 🛍️

Problem: Executives need instant, conversational access to 100GB+ retail data

Solution: GenAI-powered multi-agent system combining:

- 🤖 4-Agent Architecture (Query Resolution, Data Extraction, Validation, Analysis)
- 🔄 LangGraph Workflow (7 processing nodes)
- 🧠 LLMs (Gemini Pro / GPT-3.5-Turbo)
- 💾 DuckDB (Analytical Database)
- 🔍 FAISS + RAG (Conversation Memory)
- 📊 Streamlit UI (4 Interactive Tabs)

Key Capabilities:

1. **Summarization Mode**: Auto-generate business intelligence reports
2. **Q&A Mode**: Natural language queries with confidence scoring
3. **Data Explorer**: Interactive data profiling
4. **Data Analyst**: Comprehensive statistical analysis

Slide 2: Core Architecture

```
┌──────────────────────────────────────────────────────────────────┐
│                         USER INTERFACE                           │
│                      Streamlit Web App                           │
│  ┌──────────────┬──────────────┬──────────────┬──────────────┐  │
│  │Summarization │  Q&A Chat    │Data Explorer │Data Analyst  │  │
│  │    Tab       │    Tab       │    Tab       │    Tab       │  │
│  └──────────────┴──────────────┴──────────────┴──────────────┘  │
└──────────────────────────────────────────────────────────────────┘
                              │
        ┌─────────────────────┴─────────────────────┐
        │                                            │
┌───────▼────────┐                         ┌─────────▼─────────┐
│  LangGraph     │                         │  Multi-Agent      │
│  Q&A Engine    │                         │  Orchestrator     │
│                │                         │                   │
│ • 7 Nodes      │                         │ • 4 Specialized   │
│ • Tool Executor│                         │   Agents          │
│ • State Manager│                         │ • Sequential Flow │
└───────┬────────┘                         └─────────┬─────────┘
        │                                            │
        └─────────────────────┬─────────────────────┘
                              │
        ┌─────────────────────┴─────────────────────┐
        │                                            │
┌───────▼─────────────┐                  ┌──────────▼────────┐
│ Enhanced Query      │                  │ Summarization     │
│ Resolution          │                  │ Engine            │
│                     │                  │                   │
│ • NLP → SQL         │                  │ • Business Reports│
│ • Semantic Mapping  │                  │ • KPI Extraction  │
│ • Confidence Scoring│                  │ • Comparisons     │
└───────┬─────────────┘                  └──────────┬────────┘
        │                                            │
        └─────────────────────┬─────────────────────┘
                              │
                    ┌─────────▼──────────┐
                    │  Data Processor    │
                    │  (DuckDB Engine)   │
                    │                    │
                    │ • In-Memory OLAP   │
                    │ • Columnar Storage │
                    │ • SQL Execution    │
                    └─────────┬──────────┘
                              │
        ┌─────────────────────┴─────────────────────┐
        │                                            │
┌───────▼────────┐                       ┌──────────▼────────┐
│ LLM Layer      │                       │ Conversation      │
│                │                       │ Memory            │
│ • Gemini Pro   │                       │                   │
│ • GPT-3.5-Turbo│                       │ • FAISS Vector DB │
│ • Prompt Engine│                       │ • RAG Context     │
└────────────────┘                       │ • Embeddings      │
                                         └───────────────────┘
```

Technology Stack:

- **Orchestration**: LangGraph (StateGraph, ToolExecutor)
- **LLM**: LangChain + Gemini/OpenAI
- **Database**: DuckDB (OLAP-optimized)
- **Vector Store**: FAISS + sentence-transformers
- **UI**: Streamlit with Plotly visualizations

Slide 2.5: User Interface Layer - 4 Interactive Tabs

Streamlit UI Architecture

The application provides **4 specialized tabs**, each optimized for different analytics workflows:

```
┌──────────────────────────────────────────────────────────────────┐
│                    STREAMLIT WEB APPLICATION                      │
│                          (app_new.py)                             │
├──────────────┬──────────────┬──────────────┬─────────────────────┤
│              │              │              │                      │
│  📈 Tab 1    │  💬 Tab 2    │  📋 Tab 3    │  🔬 Tab 4           │
│              │              │              │                      │
│ Automated    │ Conversational│ Data        │ Statistical         │
│ Summaries    │ Q&A          │ Explorer    │ Analyst             │
│              │              │              │                      │
└──────┬───────┴──────┬───────┴──────┬───────┴──────┬──────────────┘
       │              │              │              │
       ▼              ▼              ▼              ▼
  Summarization  LangGraph     DataProcessor  DataAnalyst
     Engine      Multi-Agent     (DuckDB)        Agent
```

Tab 1: 📈 Automated Data Summarization

Purpose: AI-generated business intelligence reports

Core Features:

- **Single Table Summary**:
  - Executive summary with quantified insights
  - Top N analysis (categories, products, regions)

    - Trend identification with growth rates
    - Strategic recommendations
- **Multi-Table Comparison**:
    - Dimensional overlap analysis
    - Integration opportunity scoring
    - Data consolidation roadmap

Output: Markdown report + PDF export capability

Technology:

- `SummarizationEngine` (src/agents/summarization_engine.py)
- External prompts: `summarization_prompt.txt`, `comparative_summarization_prompt.txt`
- PDF generation: ReportLab library

Performance:

- Small datasets (<1K rows): 3-5 seconds
- Large datasets (>100K rows): 10-15 seconds

Tab 2: 💬 Conversational Q&A (Dual Agent Architecture)

Purpose: Natural language queries with intelligent routing

Agent Selection UI

Users choose between two query engines optimized for different complexity levels:

🧠 **LangGraph Agent** (Advanced)

- **Architecture**: 7-node state machine
- **Best For**: Complex, multi-step analytical questions
- **Workflow**:
    1. `analyze_query` → Parse intent
    2. `decompose_query` → Split into sub-queries (if complex)
    3. `extract_data` → Execute SQL queries

4. `validate_results` → Quality checks
5. `refine_query` → Retry optimization (conditional)
6. `llm_analysis` → Generate insights
7. `format_response` → Structure answer

- **Use Cases**:
  - Trend comparisons across dimensions
  - Multi-period analysis
  - Correlation detection
  - Questions requiring reasoning chains

🤖 **Multi-Agent Orchestrator** (Fast)

- **Architecture**: 4-agent linear pipeline
- **Best For**: Simple aggregations and lookups
- **Workflow**:
  1. QueryResolution → Map NL to SQL spec
  2. DataExtraction → Execute query
  3. Validation → Verify results
  4. Formatting → Structure response
- **Use Cases**:
  - Total revenue, counts, averages
  - Top N queries
  - Simple filtering and grouping
  - Dashboard-style metrics

Common UI Features (Both Agents):

- ✅ Confidence score badges (color-coded: green >0.7, yellow 0.4-0.7, red <0.4)
- ✅ Conversation memory with "Clear History" button
- ✅ Suggested follow-up questions (clickable)
- ✅ Auto-generated visualizations (Plotly charts)
- ✅ Data source selector (specific table vs all tables)

- ✅ Copy response button
- ✅ Export chat history option

Technology:

- LangGraph: `src/graph/langgraph_agent.py`
- Multi-Agent: `src/agents/multi_agent.py`
- Memory: FAISS vector store (RAG pattern)
- Prompts: 6 external files (query_resolution, decomposition, llm_analysis, validation, data_analyst)

Performance:

- Simple queries (Multi-Agent): 1-2 seconds
- Complex queries (LangGraph): 4-8 seconds
- Memory search latency: <100ms

Tab 3: 📋 Data Explorer (AI-Free Exploration)

Purpose: Quick data inspection without LLM costs

Features:

1. **Table Selector**: Dropdown of all loaded tables
2. **Metadata Panel**:
    - Row count, column count
    - Data source file path
    - Load timestamp
3. **Data Preview**: First 100 rows (interactive table)
4. **Column Profiling**:
    - Column names + data types
    - Missing value percentages
    - Unique value counts
5. **Quick Stats**: Min/max/avg for numeric columns

Sub-tabs:

- **Data View**: Raw table display with sorting
- **Visualizations**: Auto-generated charts (histograms, time series)
- **Analytics**: Simple aggregations (sum/avg/count by category)

Technology:

- Direct DuckDB queries (no LLM calls)
- Streamlit native table component
- Plotly Express for visualizations

Performance: <500ms (instant, no API calls)

Use Cases:

- Verify data loaded correctly
- Quick sanity checks
- Understand schema before asking questions
- Spot obvious quality issues

Tab 4: 🔬 Data Analyst (Comprehensive Profiling)

Purpose: Professional statistical analysis and data quality assessment

Report Sections:

1. **Executive Overview**:
    - Business domain detection
    - Data maturity scoring (1-5)
    - Dataset complexity assessment
2. **Statistical Findings**:
    - Distribution analysis (histograms, KDE)
    - Range analysis (min/max/quartiles)
    - Variability metrics (std dev, CV)
    - Correlation matrix
3. **Data Quality Assessment**:

- Completeness score (0-100%)
- Duplicate detection
- Missing value patterns
- Consistency checks

4. **Anomaly Detection** (3 severity levels):
    - 🔴 **Critical**: >3 std deviations (immediate attention)
    - 🟡 **Moderate**: 2-3 std deviations (review recommended)
    - 🟢 **Minor**: 1-2 std deviations (monitor)
5. **Categorical Insights**:
    - Top categories by frequency
    - Category diversity (Shannon entropy)
    - Rare category identification
6. **Recommendations**:
    - Prioritized action items (High/Med/Low)
    - Data cleaning steps
    - Feature engineering opportunities

Interactive Visualizations (4 sub-tabs):

- **Distributions**: Histograms for all numeric columns
- **Categories**: Bar charts for categorical breakdowns
- **Correlation Matrix**: Heatmap showing relationships
- **Box Plots**: Outlier visualization with quartiles

Technology:

- `DataAnalystAgent` (src/agents/multi_agent.py)
- Pandas statistical functions
- Plotly interactive visualizations
- External prompt: `data_analyst_prompt.txt`

Performance:

- Small datasets (<10K rows): 2-3 seconds

- Large datasets (>100K rows): 8-12 seconds

Use Cases:

- Pre-analysis data profiling
- Data quality audits
- Outlier investigation
- Understanding data distributions before modeling

Slide 3: Multi-Agent System Design

4-Agent Architecture

```
┌─────────────────────────────────────────────────────────────┐
│                  MULTI-AGENT ORCHESTRATOR                    │
└─────────────────────────────────────────────────────────────┘
                             │
         ┌───────────────────┼───────────────────┐
         │                   │                   │
    ┌────▼────┐         ┌────▼────┐        ┌─────▼─────┐
    │ AGENT 1 │         │ AGENT 2 │        │  AGENT 3  │
    │ Query   │────────▶│  Data   │───────▶│Validation │
    │Resolution│        │Extraction│        │ & Insights│
    └─────────┘         └─────────┘        └───────────┘
         │
         │
    ┌────▼────┐
    │ AGENT 4 │
    │  Data   │
    │ Analyst │
    └─────────┘
```

Agent Responsibilities

1. QueryResolutionAgent (enhanced_query_resolution.py)

- **Input**: Natural language question + conversation context
- **Processing**:
  - Parse user intent using LLM
  - Map business terms to actual columns (e.g., "revenue" → "Amount")

- Determine query type (summary/analytical/comparison/timeseries)
- Generate structured query specification

- **Output**:

```
{
  "query_type": "analytical",
  "primary_table": "sales_data",
  "entities": ["Category", "Amount"],
  "aggregations": ["sum"],
  "groupby": ["Category"],
  "confidence_score": 0.95
}
```

2. DataExtractionAgent (multi_agent.py)

- **Input**: Structured query specification
- **Processing**:
  - Build optimized SQL query
  - Execute against DuckDB
  - Handle errors and retries
  - Apply filters and aggregations
- **Output**: DataFrame + row count + metadata

3. ValidationAgent (multi_agent.py)

- **Input**: Extracted data + original query
- **Processing**:
  - Validate data quality
  - Compute direct answers from data
  - Generate quantified insights using LLM
  - Calculate confidence scores
- **Output**: Natural language insights with actual numbers

4. DataAnalystAgent (multi_agent.py)

- **Input**: Table name
- **Processing**:
  - Statistical analysis (mean, median, std, quartiles)

- Data quality assessment (completeness, duplicates, missing data)
- Anomaly detection (IQR-based outlier identification)
- Categorical distribution analysis
- LLM-powered insight generation

- **Output**: Comprehensive analysis report with visualizations

Agent Communication:

- Sequential pipeline with state passing
- Error handling with fallback strategies
- Confidence propagation through stages

Slide 3.5: Two Agent Approaches Comparison

Why Two Different Agents?

The system offers **two query processing approaches** optimized for different use cases:

```
┌──────────────────────────────────────────────────────────────┐
│                      Q&A TAB OPTIONS                          │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│  🧠 LangGraph (Advanced)      🤖 Multi-Agent (Fast)          │
│  ─────────────────────        ───────────────────            │
│  • 7-node workflow            • 4-agent pipeline             │
│  • 3-8 seconds                • 1-3 seconds                  │
│  • Complex queries            • Simple queries               │
│  • Self-healing               • Direct answers               │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Feature Comparison Matrix

Capability	LangGraph Agent	Multi-Agent Orchestrator
Architecture	State machine (7 nodes)	Linear pipeline (4 agents)
Query Decomposition	✅ Automatic	❌ Not available

Capability	LangGraph Agent	Multi-Agent Orchestrator
Validation Loops	✅ Multi-stage with retry	✅ Single-stage
Error Recovery	✅ Self-healing	⚠️ Basic error handling
Complex Joins	✅ Multi-table sub-queries	⚠️ Single table focus
Response Time	3-8 seconds	1-3 seconds
Best For	Research & exploration	Dashboards & quick facts

Processing Flow Comparison

LangGraph Workflow (Adaptive):

```
User Query
    │
    ▼
┌─────────────┐
│ Analyze     │──▶ Simple? ──▶ Extract Data ─┐
└─────────────┘                              │
    │                                        │
    │ Complex?                               │
    ▼                                        ▼
┌─────────────┐                          ┌─────────┐
│ Decompose   │──▶ Sub-queries ─────────▶│  LLM    │
└─────────────┘                          │ Analysis│
    │                                     └─────────┘
    ▼                                        │
┌─────────────┐                             │
│ Execute All │◀────────────────────────────┘
│ Sub-queries │
└─────────────┘
       │
       ▼
┌─────────────┐
│ Validate    │──▶ Valid? ──▶ Format Response
└─────────────┘      │
       ▲             │ Invalid
       │             ▼
       └───────── Refine Query
```

Multi-Agent Pipeline (Direct):

```
User Query
    │
    ▼
┌─────────────────┐
│ 1. Query        │  Parse intent, map columns
│    Resolution   │  Generate SQL structure
└─────────────────┘
         │
         ▼
┌─────────────────┐
│ 2. Data         │  Execute SQL with aggregations
│    Extraction   │  Handle GROUP BY, ORDER BY
└─────────────────┘
         │
         ▼
┌─────────────────┐
│ 3. Validation   │  Compute direct answers
│    & Insights   │  Generate insights with LLM
└─────────────────┘
         │
         ▼
┌─────────────────┐
│ 4. Response     │  Format with visualizations
│    Formatting   │  Add confidence scores
└─────────────────┘
```

Use Case Examples

LangGraph Excels At:

- "Compare sales trends between Electronics and Home categories over Q1, and identify correlation with discount rates"
- "Show me products with declining sales but increasing returns, grouped by region"
- "What's the relationship between order size and shipping cost across different fulfillment centers?"

Multi-Agent Excels At:

- "Which category generates the highest total sales?" → Direct GROUP BY + SUM
- "What's the average order value?" → Simple AVG calculation
- "Show top 10 products by revenue" → Quick ORDER BY + LIMIT
- "How many orders in May?" → Fast COUNT with filter

Technical Implementation

LangGraph Agent Features:

- Conditional branching based on query complexity
- State persistence across nodes
- Tool-based architecture (query_database, analyze_data)
- Automatic retry with query refinement
- Sub-query merging for complex analyses

Multi-Agent Orchestrator Features:

- Direct computation patterns (no LLM for simple aggregations)
- Built-in answer templates for common questions
- Optimized SQL generation with GROUP BY/aggregations
- Faster for single-table queries
- Better for real-time dashboards

Performance Metrics (Typical)

Query Type	LangGraph	Multi-Agent
Simple aggregation	4 sec	1.5 sec
Single table filter	3.5 sec	1.2 sec
Group by analysis	5 sec	2 sec
Multi-table join	7 sec	N/A (limited)
Complex decomposition	8 sec	N/A

💡 **Recommendation**: Start with Multi-Agent for quick answers. Switch to LangGraph when queries require breaking down into multiple steps or when initial results need refinement.

Slide 4: LangGraph Workflow

LangGraph Workflow Architecture

Visual representation of the 7-node state machine with conditional routing

📖 **Detailed Documentation:** See LANGGRAPH_VISUALIZATION.md for complete architecture details.

7-Node Processing Pipeline

```
               ┌─────────────────┐
               │   User Query    │
               └────────┬────────┘
                        │
               ┌────────▼────────┐
               │ 1. Analyze      │
               │    Query        │
               │                 │
               │ • Parse intent  │
               │ • Classify type │
               └────────┬────────┘
                        │
                 Simple │ Complex
           ┌────────────┴────────────┐
           │                         │
           │                ┌────────▼────────┐
           │                │ 2. Decompose    │
           │                │    Query        │
           │                │                 │
           │                │ • Break into    │
           │                │   sub-queries   │
           │                └────────┬────────┘
           │                         │
           └─────────────┬───────────┘
                         │
               ┌─────────▼────────┐
               │ 3. Extract       │
               │    Data          │
               │                  │
               │ • Execute SQL    │
               │ • Combine results│
               └─────────┬────────┘
                         │
               ┌─────────▼────────┐
               │ 4. LLM           │
               │    Analysis      │
               │                  │
               │ • Generate       │
               │   insights       │
               └─────────┬────────┘
                         │
               ┌─────────▼────────┐
               │ 5. Validate      │
               │    Results       │
               │                  │
               │ • Check quality  │
               │ • Score confidence│
               └─────────┬────────┘
                         │
                  Valid? │
           ┌─────────────┴──────────────┐
           │ No                      Yes│
┌──────────▼────────┐       ┌──────────▼────────┐
│ 6. Refine         │       │ 7. Format         │
│    Query          │       │    Response       │
│                   │       │                   │
│ • Adjust params   │       │ • Structure output│
│ • Retry extraction│       │ • Add viz specs   │
└──────────┬────────┘       └──────────┬────────┘
           │                           │
```

```
               │                      │
               └───────┐      ┌───────┘
                       │      │
                       ▼      ▼
                  ┌─────────────┐
                  │ Final Response │
                  │             │
                  │ • Summary   │
                  │ • Data      │
                  │ • Visualizations│
                  │ • Follow-ups │
                  └─────────────┘
```

LangGraph State Management

State Schema:

```
{
    "user_query": str,
    "query_analysis": dict,       # From analyze_query
    "decomposed": bool,
    "sub_queries": list,
    "extracted_data": dict,
    "llm_analysis": str,
    "needs_refine": bool,
    "validation_message": str,
    "final_response": dict
}
```

Node Functions:

- Each node is atomic and stateless
- Updates state using immutable patterns
- Conditional edges route based on state flags
- Tool executor handles database operations

Slide 5: LLM Integration Strategy

Dual-LLM Support Architecture

```
┌─────────────────────────────────────────────────┐
│                LLM Abstraction Layer             │
│                   (llm_config.py)                │
└─────────────────────────────────────────────────┘
                         │
          ┌──────────────┴──────────────┐
          │                             │
┌─────────▼────────┐          ┌─────────▼────────┐
│  Gemini Pro      │          │  GPT-3.5-Turbo   │
│                  │          │                  │
│ • Model: Pro     │          │ • Model: Turbo   │
│ • Tokens: 2048   │          │ • Tokens: 2048   │
│ • Temp: 0.1-0.3│          │ • Temp: 0.1-0.3  │
└──────────────────┘          └──────────────────┘
```

Prompt Engineering Strategy

0. Externalized Prompt Management (Production Best Practice):

````
All LLM prompts are externalized to prompts/ folder:

├── prompts/
│   ├── query_resolution_prompt.txt       # Query intent analysis
│   ├── query_decomposition_prompt.txt    # Complex query breakdown
│   ├── llm_analysis_prompt.txt           # LLM-based insights
│   ├── validation_analyst_prompt.txt     # Result validation
│   ├── data_analyst_prompt.txt           # Statistical analysis
│   ├── summarization_prompt.txt          # Business reports
│   ├── comparative_summarization_prompt.txt # Multi-table comparison
│   └── comparison_prompt.txt             # Table comparison

Benefits:
✓ Version control on prompts independently from code
✓ A/B testing without code deployment
✓ Non-technical teams can optimize prompts
✓ Consistent prompt loading via src/utils/prompt_loader.py
✓ Template variable substitution with .format()

Usage:
```python
from src.utils.prompt_loader import load_prompt

# Load prompt (automatically adds .txt extension)
prompt = load_prompt("summarization_prompt")

# With variable substitution
template = load_prompt("llm_analysis_prompt")
formatted = template.format(
    user_query=query,
    data_display=data,
    stats_info=statistics
)
````
````

1. Structured Prompt Templates:

```
System Role Definition
     │
     ├─▶ Elite-level persona (e.g., "senior retail analytics expert")
     ├─▶ Clear mission statement
     └─▶ Output format specification

Context Injection
     │
     ├─▶ Database schema
     ├─▶ Available data summary
     ├─▶ Conversation history (RAG)
     └─▶ Business domain knowledge

Instructions
     │
     ├─▶ Step-by-step reasoning requirements
     ├─▶ Quality criteria (✓ DO's and ✗ DON'Ts)
     ├─▶ Output format (JSON/Markdown/Structured)
     └─▶ Examples (few-shot learning)

Output Specification
     │
     ├─▶ Required sections
     ├─▶ Formatting rules
     └─▶ Validation criteria
```

2. Prompt Optimization Techniques:

- **Specificity**: Require actual numbers, ban vague language
- **Structure**: Numbered sections, bullet points, emojis
- **Examples**: Few-shot prompts for complex tasks
- **Constraints**: Explicit DO/DON'T lists
- **Validation**: Output format enforcement (JSON schemas)
- **Externalization**: All prompts loaded from prompts/ folder (zero hardcoded)

3. LLM Usage by Component:

Component	LLM Usage	Temperature	Why
Query Resolution	High	0.1	Need deterministic SQL generation
Data Extraction	None	N/A	Pure SQL execution
Validation	Medium	0.1	Consistent insight generation

Component	LLM Usage	Temperature	Why
LLM Analysis	High	0.2	Detailed pattern analysis
Summarization	High	0.3	Creative business insights
Data Analyst	High	0.3	Professional report writing

Conversation Memory (RAG Pattern)

Architecture:

1. Embed user query (sentence-transformers)
2. Search FAISS vector DB (K=2 nearest neighbors from conversation history)
3. Inject context: "Previously you asked..."
4. Query Resolution uses historical context

Benefits: Conversation coherence, resolves follow-ups, reduces redundancy, improves confidence

---

Slide 6: Data Flow Pipeline

End-to-End Query Processing

STEP 1: Data Ingestion

- CSV/Excel/JSON → pandas read functions
- Data validation + type inference
- Load into DuckDB (in-memory columnar storage, OLAP-indexed)

STEP 2: Query Understanding

- Natural language → Query Resolution Agent (LLM)
- Semantic mapping, terminology resolution
- Output: Structured query spec with confidence score

STEP 3: Data Extraction

```
├───────────────────────────────────────────────────────────────┤
│                                                               │
│   ┌─────────────────────┐                                     │
│   │ SQL Builder         │  • Construct optimized SQL          │
│   └─────────────────────┘  • Apply filters, aggregations     │
│              │                                                 │
│              ▼                                                 │
│   SELECT strftime('%Y-%m', Date) as Month,                    │
│          SUM(Amount) as Revenue                               │
│   FROM sales_data                                             │
│   GROUP BY Month                                              │
│   ORDER BY Month ASC                                          │
│              │                                                 │
│              ▼                                                 │
│   ┌─────────────────────┐                                     │
│   │ DuckDB Execution    │  • Columnar scan                    │
│   └─────────────────────┘  • Parallel aggregation             │
│              │             • Result materialization           │
│              ▼                                                 │
│   DataFrame[Month, Revenue]                                   │
└───────────────────────────────────────────────────────────────┘

┌───────────────────────────────────────────────────────────────┐
│ STEP 4: Insight Generation                                    │
├───────────────────────────────────────────────────────────────┤
│   DataFrame + Original Query                                  │
│              │                                                 │
│              ▼                                                 │
│   ┌─────────────────────┐                                     │
│   │ Validation Agent    │  • Compute statistics               │
│   │                     │  • Identify trends                  │
│   └─────────────────────┘  • Generate direct answers          │
│              │                                                 │
│              ▼                                                 │
│   ┌─────────────────────┐                                     │
│   │ LLM Analysis        │  • Pattern recognition              │
│   │                     │  • Business insights                │
│   └─────────────────────┘  • Quantified findings              │
│              │                                                 │
│              ▼                                                 │
│   "Monthly revenue grew from $125K in Jan to $310K in         │
│    Dec, a 148% increase. Peak month was October ($340K),      │
│    driven by holiday sales. Q4 accounts for 42% of annual     │
│    revenue."                                                  │
└───────────────────────────────────────────────────────────────┘

┌───────────────────────────────────────────────────────────────┐
│ STEP 5: Response Formatting                                   │
├───────────────────────────────────────────────────────────────┤
│   ┌─────────────────────┐                                     │
│   │ Response Formatter  │  • Structure output                 │
│   │                     │  • Generate visualizations          │
│   └─────────────────────┘  • Suggest follow-ups               │
│              │                                                 │
│              ▼                                                 │
│   {                                                           │
│     "summary": "Monthly revenue analysis...",                 │
│     "data_preview": [...],                                    │
│     "visualizations": [                                       │
│       {"type": "line_chart", "data": {...}},                 │
```

```
│       {"type": "bar_chart", "data": {...}}                      │
│     ],                                                           │
│     "suggested_followups": [                                     │
│       "Which category drove Q4 growth?",                         │
│       "Compare this year vs last year"                           │
│     ],                                                           │
│     "confidence_score": 0.92                                     │
│   }                                                              │
└──────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────┐
│ STEP 6: UI Rendering                                             │
├──────────────────────────────────────────────────────────────────┤
│   Streamlit Interface                                            │
│   ├── Summary text with markdown                                 │
│   ├── Interactive Plotly charts                                  │
│   ├── Data table preview                                         │
│   ├── Confidence score badge                                     │
│   └── Suggested follow-up buttons                                │
└──────────────────────────────────────────────────────────────────┘
**Performance Metrics** (Current Implementation):
- Query Resolution: ~1-2 seconds
- Data Extraction: ~0.1-0.5 seconds (DuckDB)
- LLM Analysis: ~2-4 seconds (API latency)
- Visualization: ~0.2 seconds
- **Total End-to-End**: ~4-7 seconds per query
```

Slide 7: Query-Response Pipeline Example

Real Query Walkthrough

User Query: *"Which category generates the highest revenue, and what is the breakdown by region?"*

Stage 1: Query Resolution

```
# Output from QueryResolutionAgent
{
    "query_type": "analytical",
    "primary_table": "sales_data",
    "entities": ["Category", "ship-state", "Amount"],
    "aggregations": ["sum"],
    "groupby": ["Category", "ship-state"],
    "orderby": {"Amount": "DESC"},
    "limit": 20,
    "parsed_intent": "Revenue by category with regional breakdown",
    "confidence_score": 0.94,
    "suggested_visualizations": ["bar_chart", "heatmap"]
}
```

Stage 2: Data Extraction

Generated SQL:

```sql
SELECT Category, "ship-state" as Region, SUM(Amount) as Revenue, COUNT(
FROM sales_data
GROUP BY Category, "ship-state"
ORDER BY Revenue DESC
LIMIT 20
```

Sample Results:

Category	Region	Revenue	OrderCount
Electronics	CA	$1,245,320	1,523
Home & Kitchen	TX	$987,450	2,104
Clothing	NY	$876,230	3,421

Stage 3: Validation & Insights

Direct Answer: Electronics generates highest revenue ($4.2M, 35% of total)

Regional Breakdown:

- CA leads: $1.24M electronics (30% of category)
- TX: $987K Home & Kitchen (category leader)
- NY: $876K Clothing (18% of category, Northeast strong)

Key Pattern: Top 3 categories = 68% revenue. Electronics dominates West Coast, Clothing in Northeast.

Business Insight: Geographic preferences suggest region-specific inventory + marketing opportunities.

Stage 4: Visualization

Generates:

1. Bar chart: Revenue by Category (top 10)
2. Heatmap: Category Performance × Region matrix

Stage 5: Follow-up Suggestions

- "What is the monthly trend for electronics revenue?"
- "Which products in electronics category are top sellers?"
- "Compare California vs Texas performance by category"

Processing Time: 5.2s (Resolution: 1.4s | SQL: 0.3s | LLM: 3.2s | Format: 0.3s)

Slide 8: 100GB+ Scalability Architecture

Transitioning from Prototype to Production Scale

Current Architecture (Good for <10GB):

```
Files (CSV) → pandas → DuckDB (in-memory) → Streamlit
```

100GB+ Architecture (Production-Ready):

```
┌─────────────────────────────────────────────────────────┐
│                   INGESTION LAYER                        │
├─────────────────────────────────────────────────────────┤
│  Cloud Storage              Batch Processing             │
│  │                          │                            │
│  ├── S3 / GCS              ├── Apache Spark (PySpark)   │
│  ├── Azure Data Lake       ├── Dask (distributed pandas)│
│  └── HDFS                  └── Airflow (orchestration)  │
└─────────────────────────────────────────────────────────┘
                            │
┌─────────────────────────────────────────────────────────┐
│                   STORAGE LAYER                          │
├─────────────────────────────────────────────────────────┤
│  Data Warehouse             Data Lake                    │
│  │                          │                            │
│  ├── BigQuery               ├── Parquet files (partitioned)│
│  ├── Snowflake              ├── Delta Lake (versioning)  │
│  ├── Redshift               └── Iceberg (table format)   │
│  └── Databricks SQL                                      │
└─────────────────────────────────────────────────────────┘
                            │
┌─────────────────────────────────────────────────────────┐
│                 INDEXING & CACHING LAYER                 │
├─────────────────────────────────────────────────────────┤
│  Vector Store              Query Cache                   │
│  │                         │                             │
│  ├── Pinecone              ├── Redis (hot queries)      │
│  ├── Weaviate              ├── Memcached                │
│  ├── Milvus                └── DuckDB (aggregates)      │
│  └── Elasticsearch                                       │
└─────────────────────────────────────────────────────────┘
                            │
┌─────────────────────────────────────────────────────────┐
│                   QUERY LAYER                            │
├─────────────────────────────────────────────────────────┤
│  Smart Router                                            │
│  │                                                       │
│  ├── Metadata filtering (date ranges, categories)       │
│  ├── Partition pruning                                  │
│  ├── Vector similarity search                           │
│  └── Cache check → Warehouse → LLM                      │
└─────────────────────────────────────────────────────────┘
                            │
┌─────────────────────────────────────────────────────────┐
│                 APPLICATION LAYER                        │
├─────────────────────────────────────────────────────────┤
│  LangGraph Multi-Agent System                            │
│  │                                                       │
│  ├── Load Balancer (multiple instances)                 │
│  ├── Async query processing                             │
│  ├── Result streaming                                   │
│  └── Monitoring & logging                               │
└─────────────────────────────────────────────────────────┘
```

Key Scaling Principles:

1. **Separation of Concerns**: Storage ≠ Compute ≠ Indexing

2. **Lazy Loading**: Retrieve only relevant data subsets
3. **Materialized Views**: Pre-compute common aggregations
4. **Distributed Processing**: Parallelize across nodes
5. **Tiered Caching**: Hot/Warm/Cold data strategy

Slide 9: Data Engineering & Preprocessing

Handling 100GB+ Ingestion at Scale

Challenge: Processing massive CSV files without OOM errors

Solution 1: Spark (Distributed Batch)

- 10-node cluster processes 100GB in 10-15 minutes
- Deduplication, filtering, date parsing
- Output: Partitioned Parquet (Year/Month to S3)
- **Benefits**: Parallel execution, fault tolerance, automatic partitioning

Solution 2: Dask (Incremental)

- Pandas-like API with lazy evaluation
- 64MB chunks for memory efficiency
- Integrates with existing pandas code
- **Benefits**: Python-native, scales to multi-TB

Solution 3: Streaming (Kafka → Spark → Delta)

- Real-time ingestion from Kafka topic
- Spark Streaming with 10-min watermark
- Delta Lake for ACID writes, checkpointing
- **Benefits**: Near real-time (seconds latency), exactly-once processing

Trigger computation and save

```
cleaned_ddf = ddf \
    .drop_duplicates(subset=['Order ID']) \
    .query("Amount > 0") \
    .assign(Date=lambda df: dd.to_datetime(df['Date'])) \
    .assign(Month=lambda df: df['Date'].dt.to_period('M'))

cleaned_ddf.to_parquet(
    "s3://retail-data/curated/sales_dask",
    partition_on=['Year', 'Month'],
    compression='snappy'
)
```

Benefits:

- Pandas-like API with distributed computing
- Scales to multi-TB datasets
- Suitable for Python-heavy workflows
- Integrates with existing pandas code

Solution 3: Streaming Ingestion (Real-time Data)

```
# Kafka → Spark Streaming → Delta Lake
from pyspark.sql.streaming import StreamingQuery

# Read from Kafka topic
streaming_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "retail-sales") \
    .load()

# Parse JSON and clean
parsed_df = streaming_df \
    .selectExpr("CAST(value AS STRING) as json") \
    .select(from_json(col("json"), schema).alias("data")) \
    .select("data.*") \
    .withWatermark("timestamp", "10 minutes")

# Write to Delta Lake with ACID guarantees
query = parsed_df.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "s3://checkpoints/sales") \
    .start("s3://retail-data/delta/sales")
```

Benefits:

- Handle continuous data streams
- Near real-time analytics (seconds latency)
- Exactly-once processing guarantees

Data Quality Pipeline

Automated checks with Great Expectations:

- Schema validation (column types, required fields)
- Value ranges (Amount > 0, valid dates)
- Referential integrity (Order IDs exist)
- Statistical anomalies (revenue >3 std dev flagged)

Implementation: Run checks in Airflow DAG before materializing views

---

Slide 10: Storage & Indexing Strategy

Multi-Tier Storage Architecture

Tier	Data Age	Size	Storage	Latency	Use Case
Hot	Last 3 months	~5GB	Redis + DuckDB (in-memory)	<100ms	Real-time dashboards
Warm	Last 12 months	~20GB	BigQuery/Snowflake (columnar)	1-5s	Ad-hoc queries
Cold	2+ years	100GB+	S3 Glacier / Parquet	10+s	YoY comparisons, compliance

Recommended: Delta Lake on Cloud Storage

Key Features:

- ACID transactions on object storage
- Time travel (query historical versions)

- Schema evolution, auto-optimization
- Parquet format with statistics

Optimization:

- Partition: Year/Month/Category (pruning)
- ZORDER clustering on frequent filters
- OPTIMIZE for compaction, VACUUM (7-day retention)

BigQuery Implementation (Serverless)

Configuration:

- Partitioned by Date (auto-pruning)
- Clustered by Category + ship-state
- Pay-per-query (~$0.05 for 100GB with partition pruning)

Benefits: Auto-scaling, no idle costs, sub-second queries on TBs, built-in ML

Vector Indexing for Semantic Search

Pinecone Strategy:

1. Index common queries with embeddings
2. Store SQL translations as metadata
3. At query time: Embed → Find similar (top_k=3, threshold >85%)
4. If match: Return cached SQL | Else: Process with LLM

Result: 60-80% cache hit rate, significant LLM cost reduction

---

Slide 11: Retrieval & Query Efficiency

Smart Query Router (Pre-filtering before LLM)

```
User Query: "Show me electronics sales in California for Q1 2024"
     |
     ▼
┌──────────────────────────────────────────────────────────┐
│ METADATA EXTRACTION (No LLM needed)                      │
├──────────────────────────────────────────────────────────┤
│ • Temporal: Q1 2024 → 2024-01-01 to 2024-03-31          │
│ • Category: electronics → "Electronics"                  │
│ • Geography: California → ship-state = 'CA'              │
└──────────────────────────────────────────────────────────┘
     |
     ▼
┌──────────────────────────────────────────────────────────┐
│ PARTITION PRUNING                                        │
├──────────────────────────────────────────────────────────┤
│ Original Data: 100GB across 48 months                    │
│ After Filtering: 2GB (3 months × CA only)                │
│ Reduction: 98% → 50x faster queries                      │
└──────────────────────────────────────────────────────────┘
     |
     ▼
┌──────────────────────────────────────────────────────────┐
│ QUERY EXECUTION (on 2GB only)                            │
├──────────────────────────────────────────────────────────┤
│ SELECT Month, SUM(Amount) as Revenue                     │
│ FROM sales_partitioned                                   │
│ WHERE Year = 2024 AND Month IN (1,2,3)                   │
│   AND Category = 'Electronics'                           │
│   AND `ship-state` = 'CA'                                │
│ GROUP BY Month                                           │
└──────────────────────────────────────────────────────────┘
```

RAG (Retrieval-Augmented Generation) Pattern

Current Implementation:

- FAISS for conversation memory
- Sentence-BERT embeddings
- K=2 nearest previous queries

100GB Scale Enhancement:

```python
# Hybrid search: Metadata + Vector similarity
class ScalableQueryRetriever:
    def __init__(self):
        self.metadata_index = ElasticsearchIndex()
        self.vector_index = PineconeIndex()
        self.sql_cache = RedisCache()

    def retrieve_relevant_subset(self, user_query):
        # Step 1: Extract filters from query
        filters = extract_metadata(user_query)
        # {"date_range": "2024-Q1", "category": "Electronics"}

        # Step 2: Get relevant partitions
        partitions = self.metadata_index.filter(filters)
        # ["s3://data/2024/01/*", "s3://data/2024/02/*", ...]

        # Step 3: Check if similar query cached
        cached_sql = self.sql_cache.get_similar(user_query)
        if cached_sql:
            return execute_on_partitions(cached_sql, partitions)

        # Step 4: Vector search for similar queries
        similar_queries = self.vector_index.query(
            embed(user_query),
            filter={"partitions": partitions},
            top_k=5
        )

        # Use LLM only if no match
        if similar_queries.score > 0.9:
            return similar_queries[0].result
        else:
            return llm_query_resolution(user_query, context=similar_que
```

Benefits: 98% less data scanned | 80% cache hit rate | 3× faster | 60% LLM cost savings

Materialized Views for Common Queries

Pre-compute daily aggregates:

```sql
CREATE MATERIALIZED VIEW sales_daily_summary AS
SELECT DATE(Date) as day, Category, `ship-state` as state,
       SUM(Amount) as total_revenue, COUNT(*) as order_count, AVG(Amoun
FROM sales_data
GROUP BY day, Category, state;
```

Result: Query "total revenue by category this month" scans 10MB view vs 100GB table (10,000× faster)

Auto-refresh: Airflow DAG runs hourly, processes only new data since last refresh

---

Slide 12: Model Orchestration at Scale

Handling High Query Volumes

Problem: 1000 concurrent users × 5s LLM latency = bottleneck

Solution: Async Processing + Load Balancing

```
┌─────────────────────────────────────────────────────────┐
│              USER REQUEST LOAD BALANCER                  │
│                  (nginx / AWS ALB)                       │
└─────────────────────────────────────────────────────────┘
                         │
         ┌───────────────┼───────────────┐
         │               │               │
    ┌────▼────┐     ┌────▼────┐     ┌────▼────┐
    │ Instance│     │ Instance│     │ Instance│
    │    1    │     │    2    │     │    3    │
    └────┬────┘     └────┬────┘     └────┬────┘
         │               │               │
         └───────────────┼───────────────┘
                         │
              ┌──────────▼──────────┐
              │   Message Queue     │
              │  (Celery/RabbitMQ)  │
              └──────────┬──────────┘
                         │
         ┌───────────────┼───────────────┐
         │               │               │
    ┌────▼────┐     ┌────▼────┐     ┌────▼────┐
    │ Worker  │     │ Worker  │     │ Worker  │
    │  Pod 1  │     │  Pod 2  │     │  Pod 3  │
    │         │     │         │     │         │
    │ LangGraph│    │ LangGraph│    │ LangGraph│
    │ Agent   │     │ Agent   │     │ Agent   │
    └─────────┘     └─────────┘     └─────────┘
```

Cost Optimization Strategies

1. Prompt Caching (Redis)

Strategy: Hash prompt → Check cache → Return if hit → Else call LLM + cache result (TTL: 1hr)

Implementation: MD5 hash as cache key, store response in Redis

Savings: 70% reduction in API calls for repeated queries

2. Model Tiering (Route by Complexity)

Strategy: Analyze query complexity → Route to appropriate model tier

Tiers:

- Simple lookups (<0.3): GPT-3.5-Turbo ($0.0015/1K tokens)
- Moderate analysis (0.3-0.7): Gemini Pro ($0.00025/1K tokens)
- Complex reasoning (>0.7): GPT-4 ($0.03/1K tokens)

Savings: 60% reduction in API costs

3. Batch Processing

Strategy: Combine multiple queries into single LLM call, parse JSON array response

Use Cases: Dashboard rendering (10 charts), report generation (multiple metrics)

Savings: 70% reduction in API calls for multi-question scenarios

SLA Targets for Production

Metric	Target	Current	Gap
Query Latency (p95)	< 3 seconds	~5-7 seconds	Need optimization
Throughput	100 req/sec	~10 req/sec	Need load balancing
Cache Hit Rate	> 80%	~0% (no cache)	Need Redis
LLM API Cost	< $0.01/query	~$0.03/query	Need caching
Data Scan Reduction	> 95%	~0% (full scan)	Need partitioning
Uptime	99.9%	NA	Need monitoring

Slide 13: Monitoring & Evaluation

Observability Stack

```
┌──────────────────────────────────────────────────────────────┐
│                      APPLICATION LAYER                        │
│              (Instrumented with OpenTelemetry)               │
└──────────────────────────────────────────────────────────────┘
                      │
              ┌───────┼─────────────┐
              │       │             │
         ┌────▼──┐ ┌───▼───┐  ┌────▼───┐
         │Metrics │ │ Traces │ │  Logs  │
         │(Prometheus)│(Jaeger) │ │(ELK Stack)│
         └───────┘ └───────┘  └────────┘
              │        │            │
              └────────┼────────────┘
                       │
              ┌────────▼─────────────┐
              │   Grafana Dashboard  │
              │   +  Alerting        │
              └──────────────────────┘
```

Key Metrics to Track

1. Query Performance

Instrumentation: Prometheus Histogram decorators on LangGraph nodes

Metrics:

- Query latency (p50, p95, p99) per node
- Node processing time breakdown
- SQL execution time
- LLM API latency

Alerts: Latency >10s, error rate >5%

2. LLM Performance

Instrumentation: Counter + Histogram wrappers on llm.invoke() calls

Metrics:

- API call success/failure rate by model
- Token usage (input/output) per model

- Cost per query and daily totals
- Model selection distribution (tiering effectiveness)

Alerts: Cost spike >$50/day, error rate >10%

3. Data Quality

Instrumentation: Scheduled checks on data freshness and completeness

Metrics:

- Data freshness (last updated timestamp)
- Quality score = (completeness × 0.7) + ((1 - duplicates) × 0.3)
- Missing data percentage per table/column

Evaluation Framework

1. Answer Accuracy

```python
# Human-in-the-loop evaluation
test_queries = [
    {
        "query": "What is total revenue?",
        "expected_answer_range": [1000000, 2000000],
        "query_type": "summary"
    },
    # ... more test cases
]

def evaluate_accuracy():
    results = []
    for test in test_queries:
        response = agent.process_query(test["query"])

        # Extract number from response
        predicted_value = extract_number(response["summary"])
        expected_range = test["expected_answer_range"]

        is_correct = expected_range[0] <= predicted_value <= expected_r
        results.append({
            "query": test["query"],
            "correct": is_correct,
            "confidence": response["confidence_score"]
        })

    accuracy = sum(r["correct"] for r in results) / len(results)
    return accuracy
```

Target: >90% accuracy on test set

2. Response Latency Targets

Query Type	p50	p95	p99
Simple (cached)	0.5s	1s	2s
Analytical	2s	4s	6s
Complex (multi-step)	4s	8s	12s

3. User Satisfaction

```python
# Collect feedback inline
def render_response(response):
    st.write(response["summary"])

    col1, col2 = st.columns(2)
    with col1:
        if st.button("👍 Helpful"):
            log_feedback(response["query_id"], rating=1)
    with col2:
        if st.button("👎 Not Helpful"):
            log_feedback(response["query_id"], rating=0)
```

Target: >85% positive feedback

Slide 14: Cost & Performance Considerations

Cost Breakdown (Monthly Estimates)

Current (<10GB): $33/month

- LLM API: $2.50 (10K queries, Gemini Pro)
- Compute: $30 (EC2 t3.medium)
- Storage: $0 (in-memory)

Production (100GB+): $432/month

- Storage: $14.20 (BigQuery $5, S3 $9.20)
- Compute: $235 (BigQuery $5, Kubernetes $210, DuckDB $20)
- LLM API: $25 (100K queries, 70% cache hit → 30K calls @ Gemini Pro)
- Caching/Indexing: $155 (Redis $85, Pinecone $70)
- Monitoring: $50 (CloudWatch/Grafana/ALB)

Key Optimizations:

Optimization	Savings/Month
Prompt caching (70% hit rate)	$242

Optimization	Savings/Month
Partitioning (90% less scan)	$45
Materialized views	$35
Model tiering (Gemini vs GPT-4)	$150
Auto-scaling (3-10 pods vs fixed 10)	$990

ROI: Without optimizations = $1,895/month → With optimizations = $432/month (**77% savings**)

Performance Targets

Metric	Target	Measurement
Availability	99.9%	Max 43 min downtime/month
Latency (p95)	<3s	95% queries
Throughput	100 queries/sec	Peak load (1000+ users)
Accuracy	>90%	Weekly eval
Cache hit rate	>80%	Cost efficiency

Slide 15: Implementation Summary

✅ Completed Features

Core Requirements (Assignment):

- ✅ Multi-agent system (4 agents: Query Resolution, Data Extraction, Validation, Data Analyst)
- ✅ LangGraph orchestration (7-node workflow with tools)
- ✅ Summarization mode (business intelligence reports)
- ✅ Conversational Q&A mode (with memory and RAG)
- ✅ CSV/Excel/JSON support
- ✅ Prompt engineering (elite-level prompts across all agents)

- ✅ Confidence scoring
- ✅ Streamlit UI (4 tabs)
- ✅ Data visualization (Plotly charts)
- ✅ PDF export

Technology Stack:

- ✅ LLM: Gemini Pro + GPT-3.5-Turbo (dual support)
- ✅ Framework: LangChain + LangGraph
- ✅ Database: DuckDB (OLAP-optimized)
- ✅ Vector Store: FAISS + sentence-transformers
- ✅ UI: Streamlit with 4 interactive tabs

Advanced Features:

- ✅ Conversation memory with RAG
- ✅ Query decomposition for complex questions
- ✅ Automatic visualization generation
- ✅ Suggested follow-up questions
- ✅ Data quality assessment
- ✅ Statistical analysis with anomaly detection

📋 Scalability Roadmap (100GB+)

Phase 1: Foundation (Months 1-2)

- [ ] Implement PySpark data processing pipeline
- [ ] Set up BigQuery data warehouse
- [ ] Deploy to cloud (AWS/GCP/Azure)
- [ ] Add Redis caching layer
- [ ] Implement partition pruning

Phase 2: Optimization (Months 3-4)

- [ ] Deploy Pinecone for vector search
- [ ] Build materialized views for common queries
- [ ] Implement async query processing

- [ ] Add Kubernetes auto-scaling
- [ ] Set up monitoring (Prometheus + Grafana)

Phase 3: Production (Months 5-6)

- [ ] Load balancing across multiple instances
- [ ] Implement query queue (Celery)
- [ ] Add user authentication (OAuth2)
- [ ] Cost optimization (prompt caching, model tiering)
- [ ] Comprehensive alerting and SLOs

Expected Results:

- **Scale**: Handle 100GB+ datasets
- **Performance**: <3 second p95 latency
- **Cost**: <$500/month for 100K queries
- **Reliability**: 99.9% uptime

🎯 Key Differentiators

1. Production-Ready Multi-Agent System

- Not just a chatbot — intelligent agent workflow
- LangGraph for complex orchestration
- Tool-based execution with retry logic

2. Intelligent Query Understanding

- Semantic mapping (business terms → SQL)
- Context-aware query resolution
- Confidence scoring at every stage

3. Scalability-First Design

- Clear path from prototype to production
- Modular architecture supports incremental scaling
- Cost-conscious design decisions

4. Business Value Focus

- Quantified insights with actual numbers
- Actionable recommendations
- Executive-ready reports

📊 Demo Architecture Diagram

```
                ┌──────────────────┐
                │  STREAMLIT UI    │
                │  4 Tabs          │
                └────────┬─────────┘
                         │
          ┌──────────────┴──────────────┐
          │                             │
┌─────────▼─────────┐         ┌─────────▼─────────┐
│  LANGGRAPH        │         │  MULTI-AGENT      │
│  (7 Nodes)        │         │  ORCHESTRATOR     │
└─────────┬─────────┘         └─────────┬─────────┘
          │                             │
          └──────────────┬──────────────┘
                         │
               ┌─────────▼─────────┐
               │  ENHANCED QUERY   │
               │   RESOLUTION      │
               └─────────┬─────────┘
                         │
               ┌─────────▼─────────┐
               │   DATA PROCESSOR  │
               │   (DuckDB OLAP)   │
               └─────────┬─────────┘
                         │
          ┌──────────────┴──────────────┐
          │                             │
┌─────────▼─────────┐         ┌─────────▼─────────┐
│  LLM LAYER        │         │  CONVERSATION     │
│ Gemini/OpenAI     │         │  MEMORY (FAISS)   │
└───────────────────┘         └───────────────────┘

  SCALABILITY: 100GB+ Architecture

┌──────────────────────────────────────────┐
│ CLOUD STORAGE: S3 / GCS / Azure Data Lake │
│ DATA WAREHOUSE: BigQuery / Snowflake     │
│ PROCESSING: PySpark / Dask                │
│ CACHE: Redis / Memcached                 │
│ VECTOR DB: Pinecone / Weaviate           │
│ COMPUTE: Kubernetes (Auto-scaling)       │
│ MONITORING: Prometheus + Grafana         │
└──────────────────────────────────────────┘
```

📚 References & Resources

Documentation:

- See `README.md` for setup instructions
- See `SCALABILITY_DESIGN.md` for detailed 100GB+ architecture
- See `SCREENSHOTS_GUIDE.md` for UI walkthrough

Key Technologies:

- LangChain: https://python.langchain.com/
- LangGraph: https://langchain-ai.github.io/langgraph/
- DuckDB: https://duckdb.org/
- Streamlit: https://docs.streamlit.io/

END OF PRESENTATION

This architecture supports both current prototype (<10GB) and production-scale (100GB+) deployments with clear migration path.