

Retail Insights Assistant – Architecture Presentation

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



Table of Contents





1. [System Overview](#)
 2. [Core Architecture](#)
 3. [Multi-Agent System Design](#)
 4. [LangGraph Workflow](#)
 5. [LLM Integration Strategy](#)
 6. [Data Flow Pipeline](#)
 7. [Query-Response Pipeline Example](#)
 8. [100GB+ Scalability Architecture](#)
 9. [Data Engineering & Preprocessing](#)
 10. [Storage & Indexing Strategy](#)
 11. [Retrieval & Query Efficiency](#)
 12. [Model Orchestration at Scale](#)
 13. [Monitoring & Evaluation](#)
 14. [Cost & Performance Considerations](#)
 15. [Implementation Summary](#)
-

Slide 1: System Overview

Retail Insights Assistant

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

Solution: GenAI-powered multi-agent system combining:

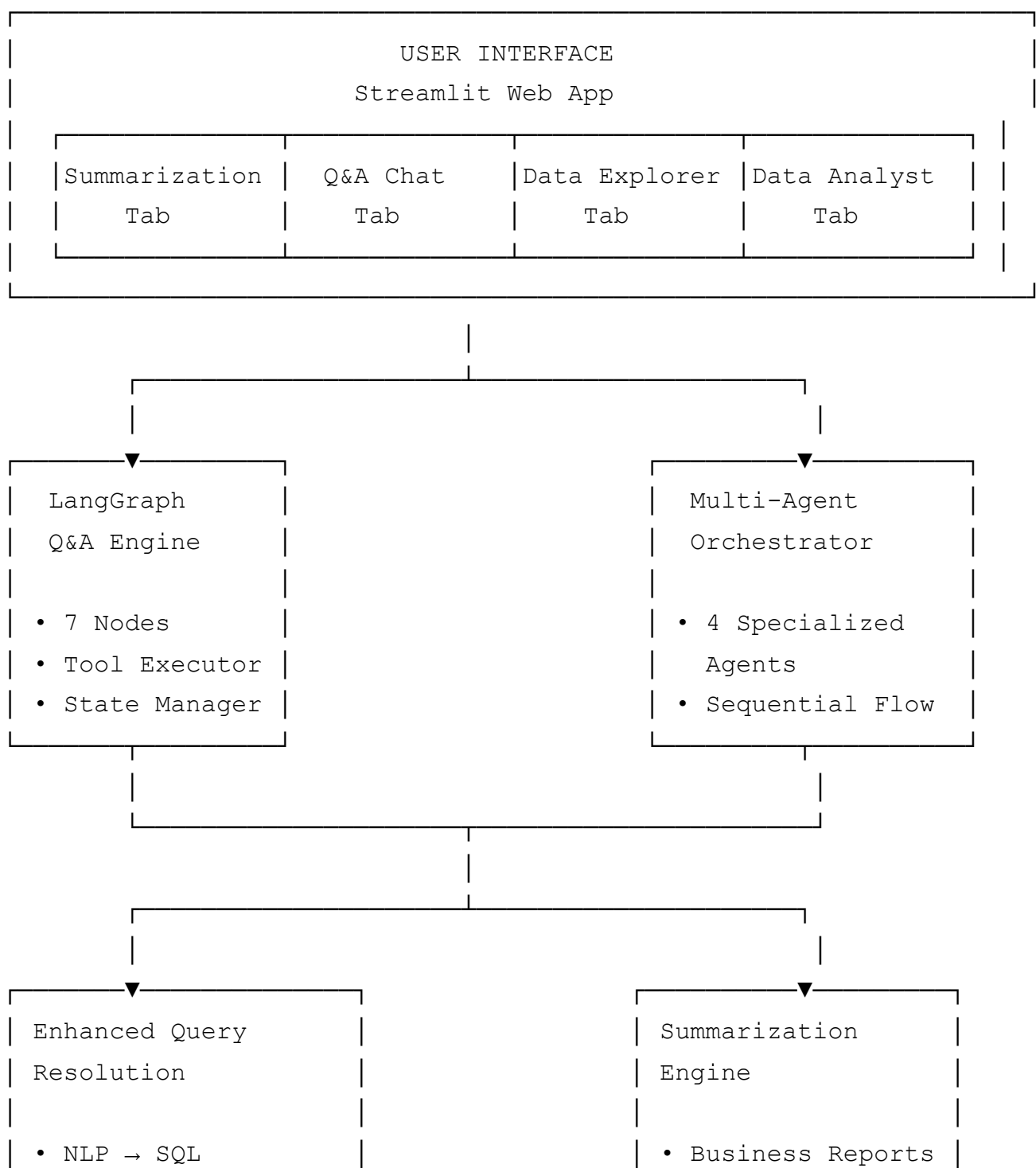
-  4-Agent Architecture (Query Resolution, Data Extraction, Validation, Analysis)
-  LangGraph Workflow (7 processing nodes)
-  LLMs (Gemini Pro / GPT-3.5-Turbo)
-  DuckDB (Analytical Database)

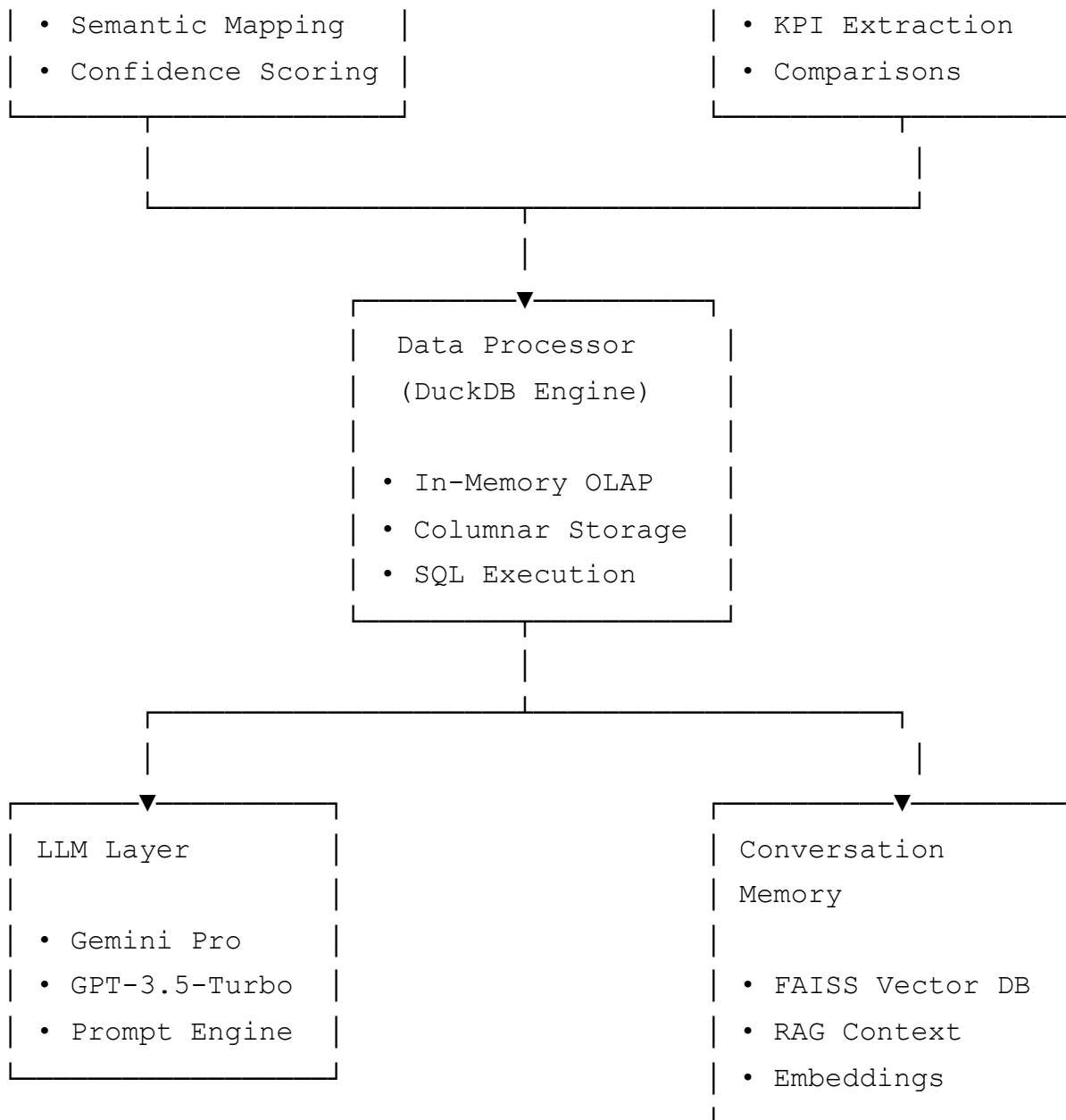
- 🔍 FAISS + RAG (Conversation Memory)
- 📊 Streamlit UI (4 Interactive Tabs)

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





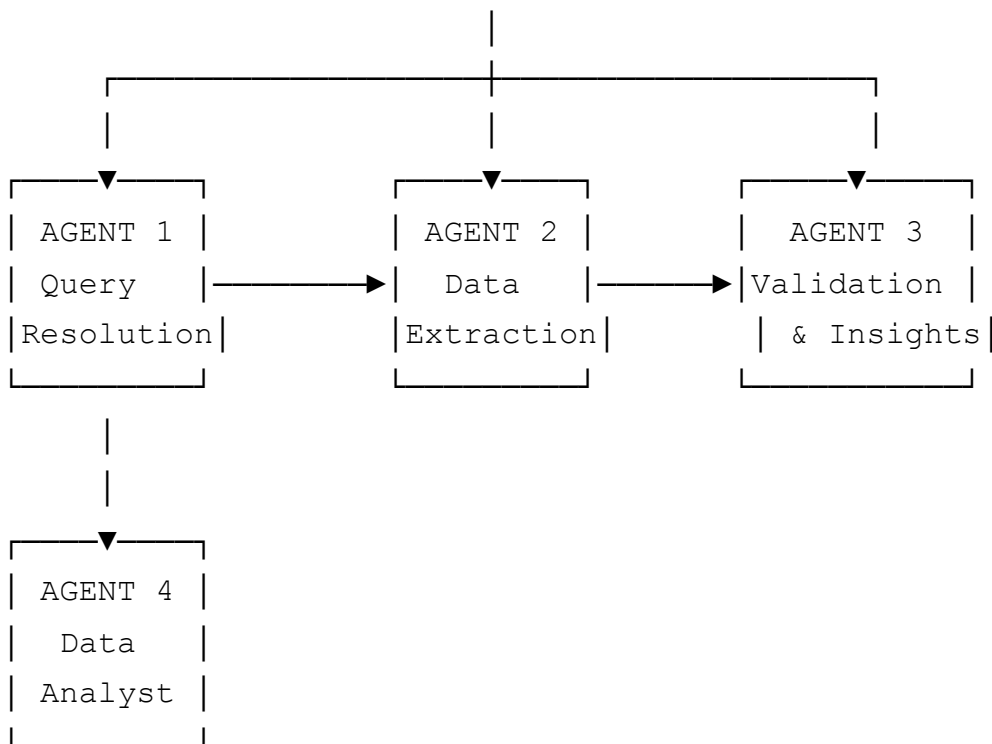
Technology Stack:

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

Slide 3: Multi-Agent System Design

4-Agent Architecture

MULTI-AGENT ORCHESTRATOR



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

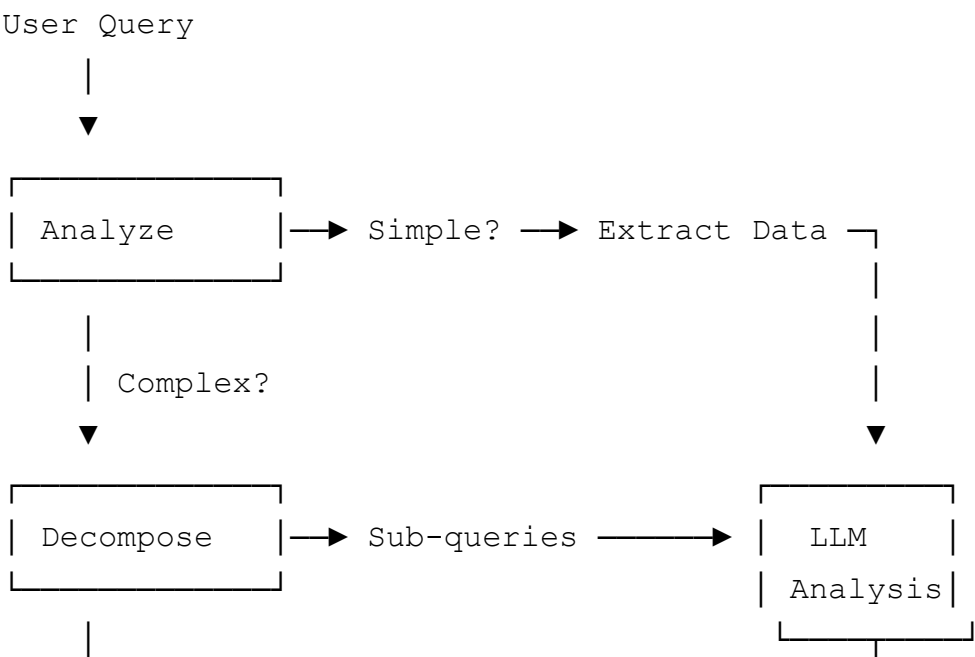
 LangGraph (Advanced)	 Multi-Agent (Fast)
<ul style="list-style-type: none">• 7-node workflow• 3-8 seconds• Complex queries• Self-healing	<ul style="list-style-type: none">• 4-agent pipeline• 1-3 seconds• Simple queries• Direct answers

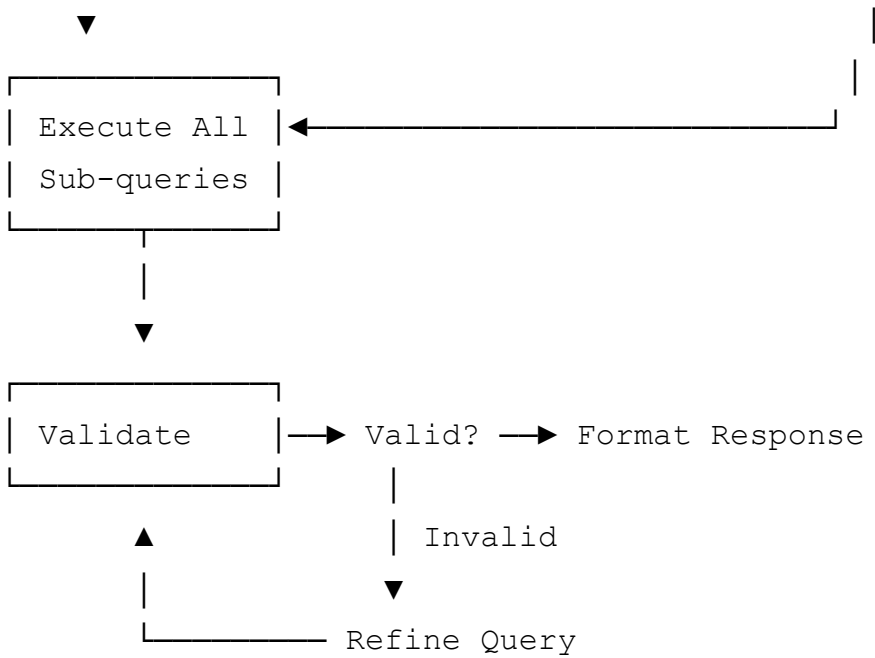
Feature Comparison Matrix

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

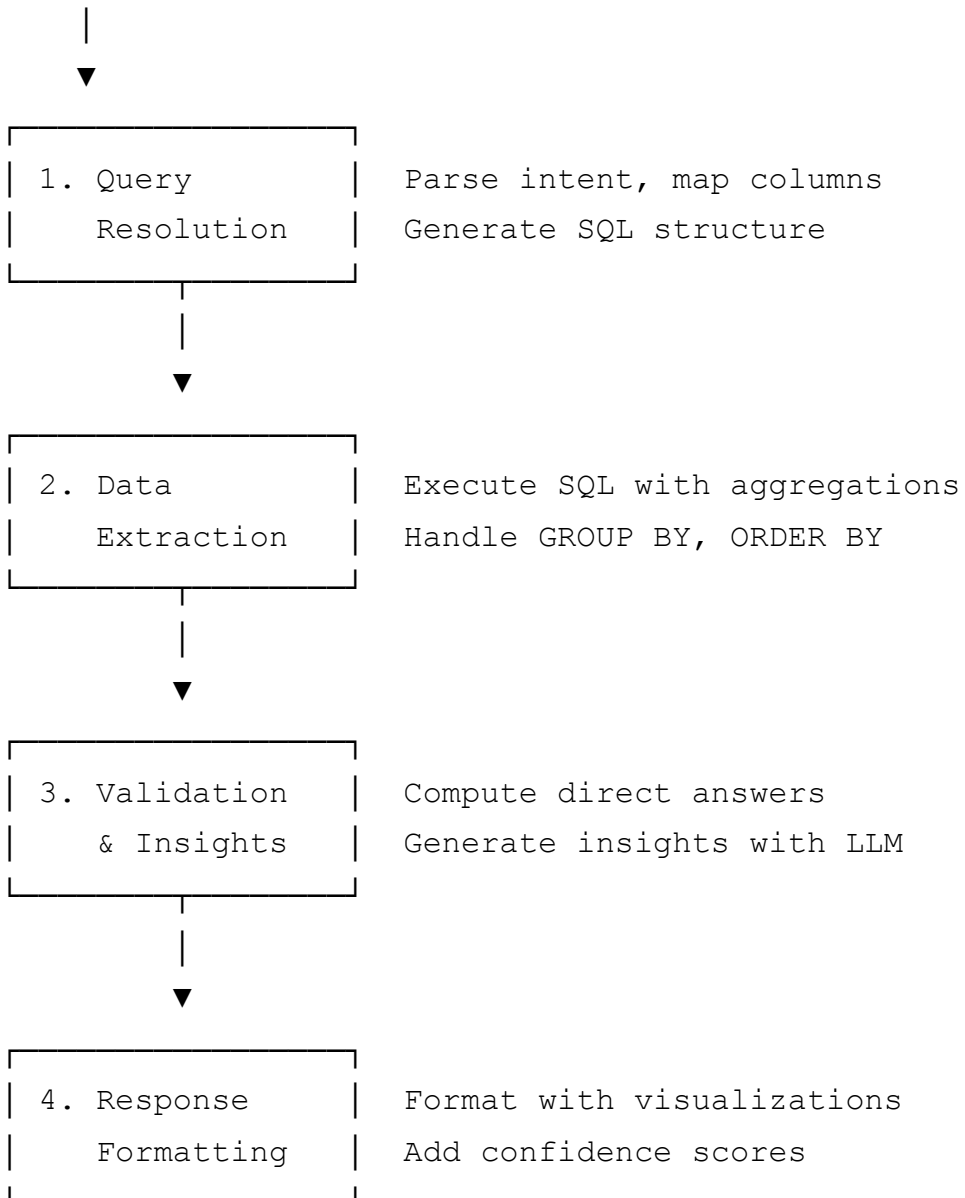
LangGraph Workflow (Adaptive):





Multi-Agent Pipeline (Direct):

User Query



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

Query Type	LangGraph	Multi-Agent
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.

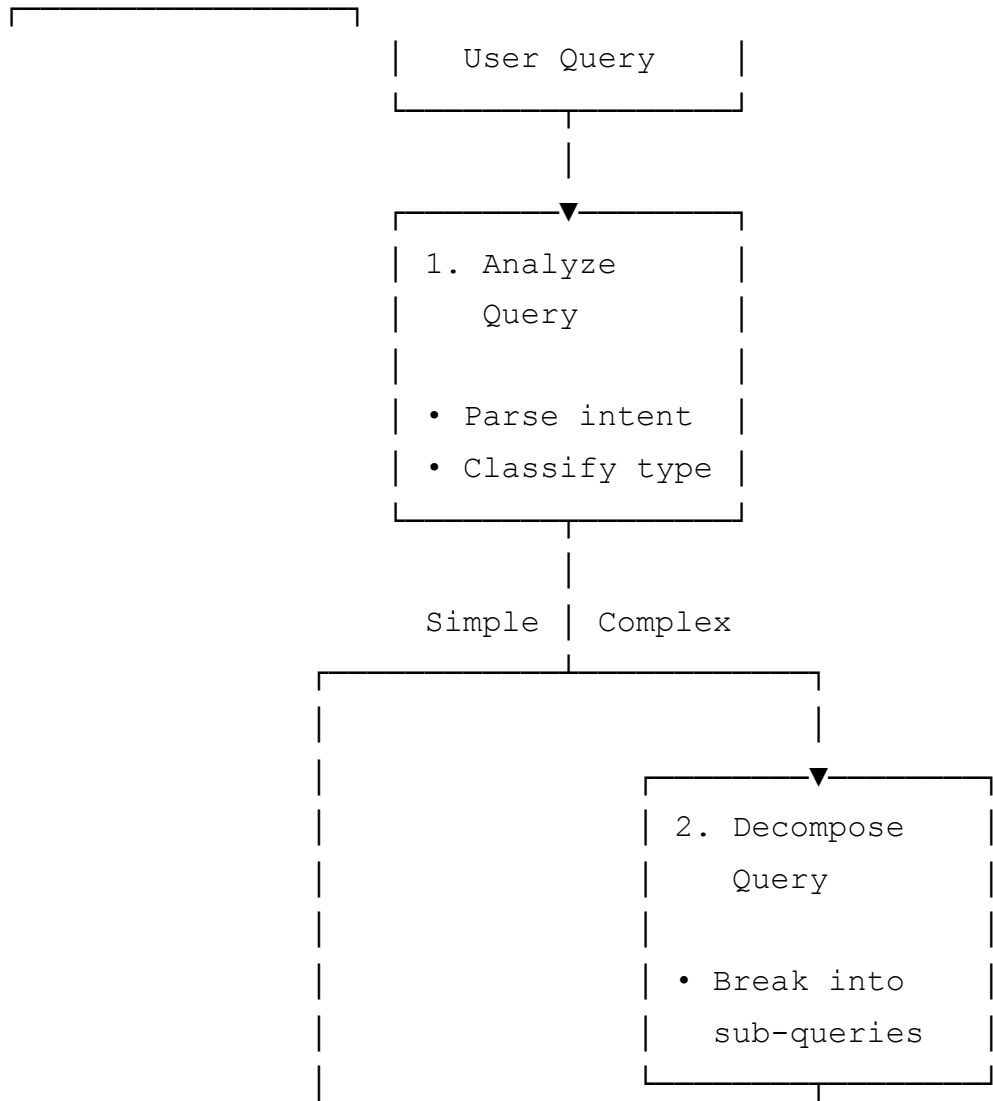
Slide 4: LangGraph Workflow

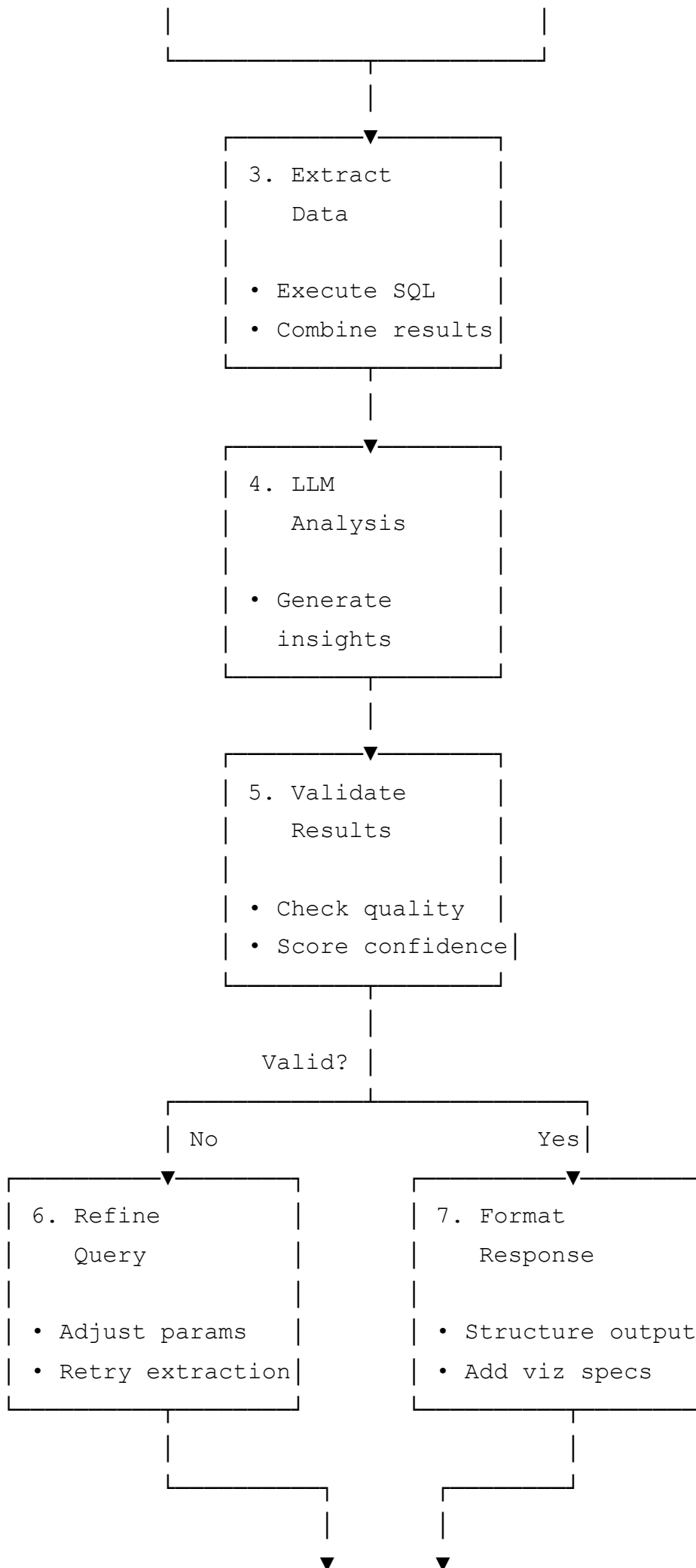
🖼️ LangGraph Workflow Architecture

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





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
}
```

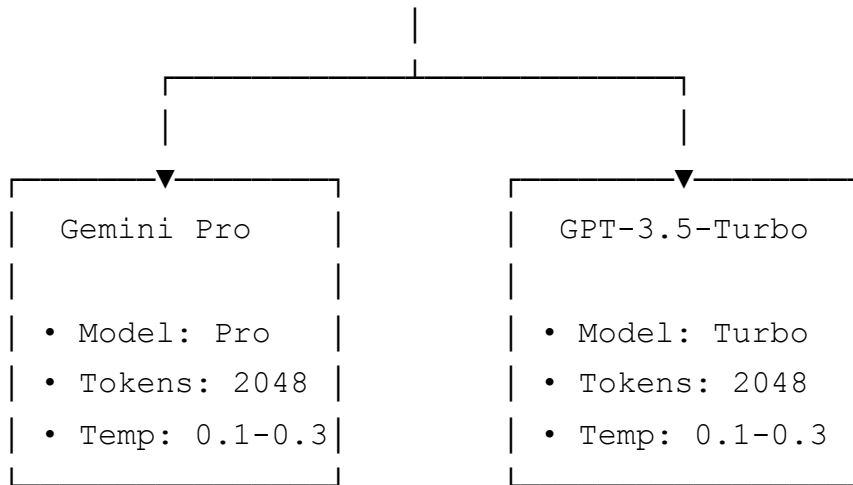
Node Functions:

- 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)
--



Prompt Engineering Strategy

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

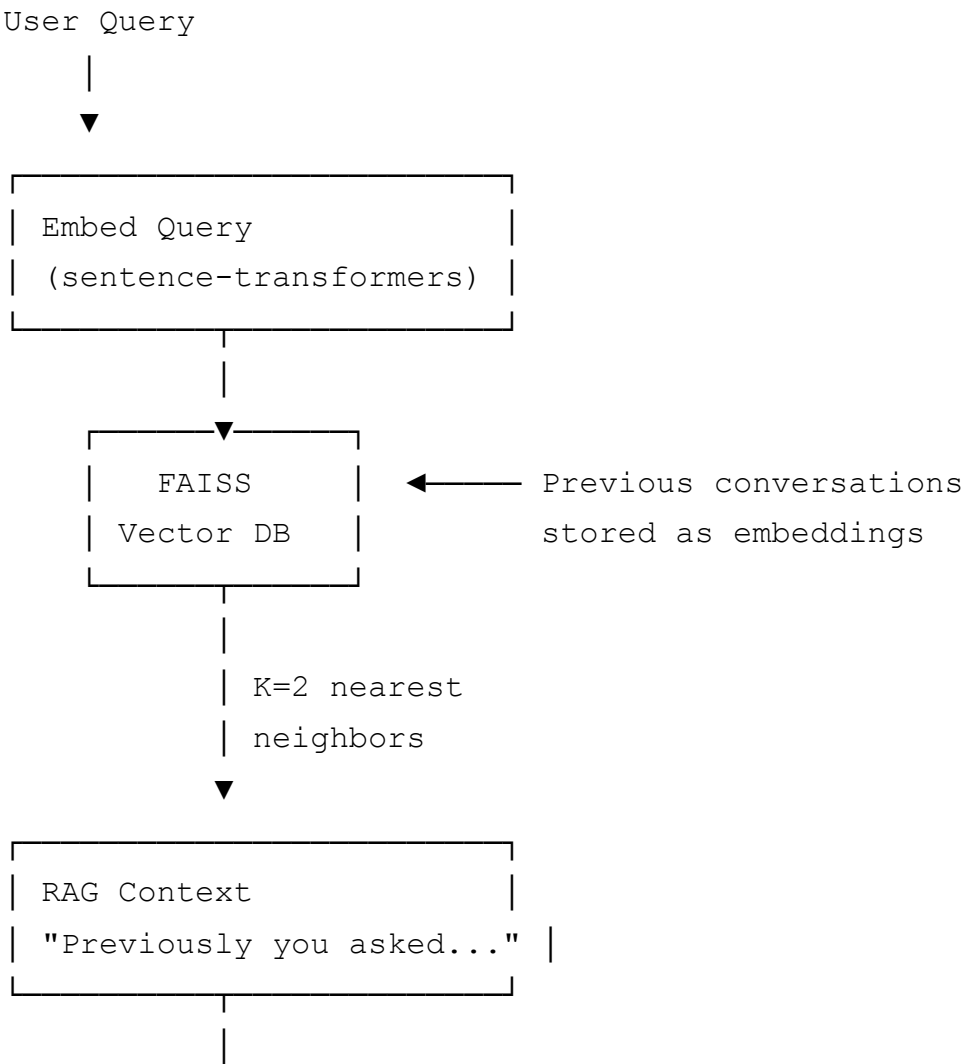
2. Prompt Optimization Techniques:

- **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)

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
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)



| Inject into prompt



Query Resolution +
Historical Context

Benefits:

- Maintains conversation coherence
- Resolves ambiguous follow-up questions
- Reduces redundant queries
- Improves confidence scores

Slide 6: Data Flow Pipeline

End-to-End Query Processing

STEP 1: Data Ingestion

CSV/Excel/JSON Files



Input Loader

- `pandas.read_csv/read_excel/read_json`
- Data validation
- Type inference

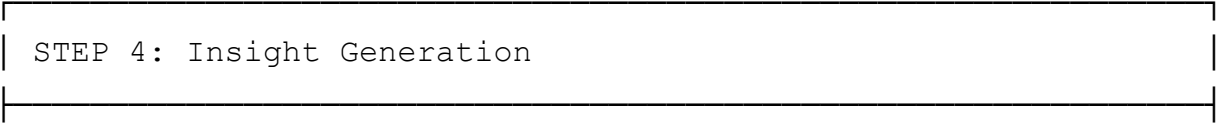
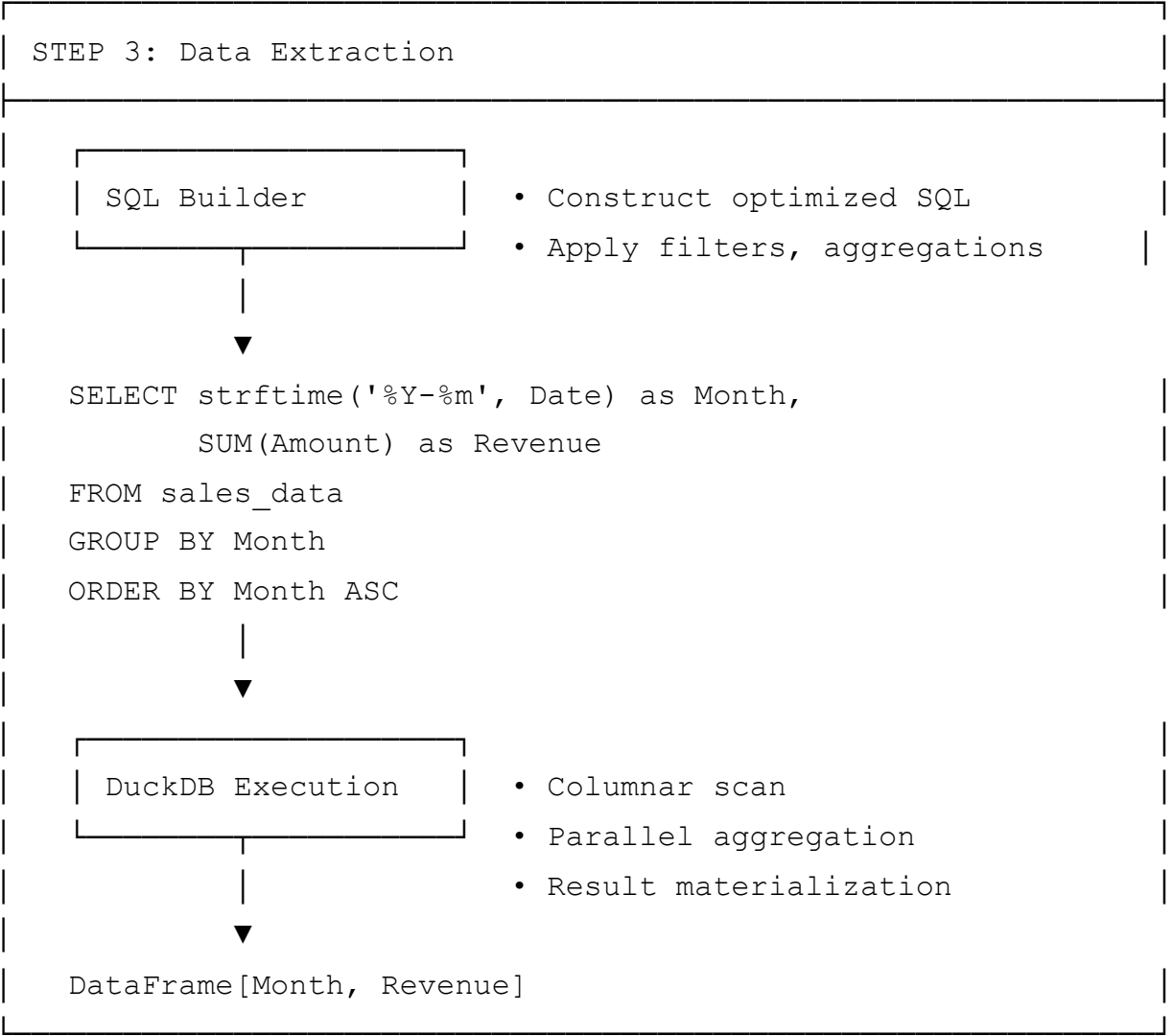
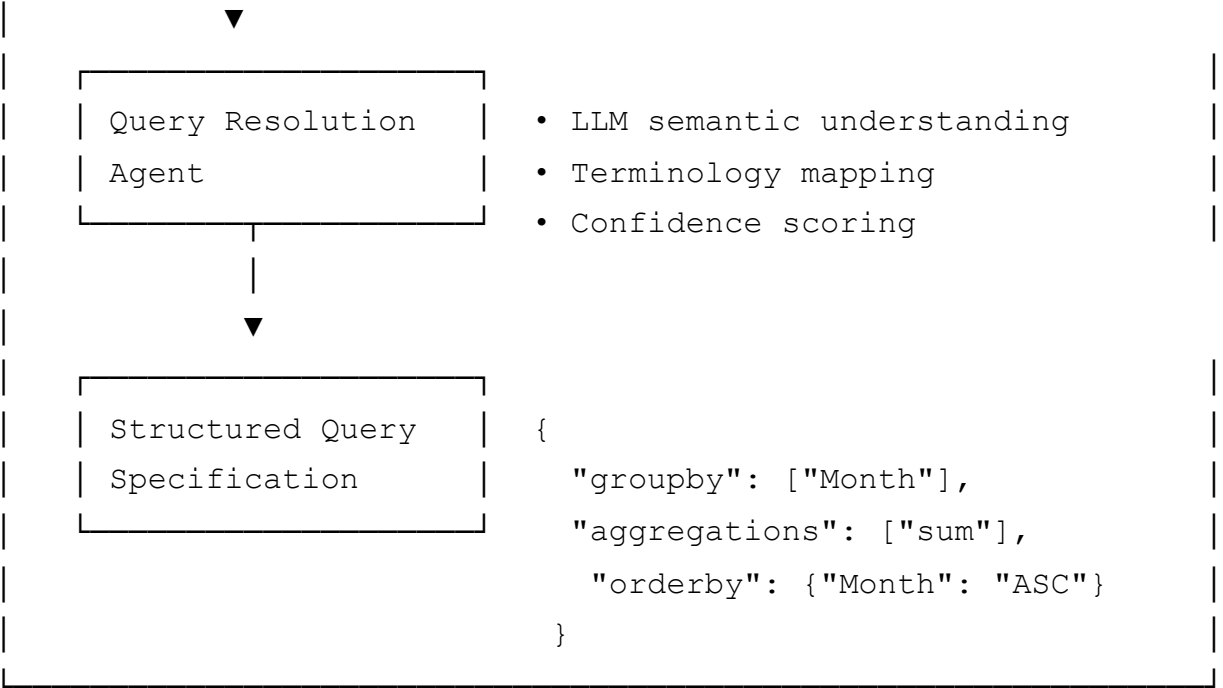


DuckDB
(In-Memory)

- `CREATE TABLE AS SELECT *`
- Columnar storage
- Indexed for OLAP queries

STEP 2: Query Understanding

Natural Language Query: "What is monthly revenue trend?"



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

Extracted Data (sample):

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 the highest total revenue at \$4.2M across all regions,
representing 35% of total sales.

Regional Breakdown:

- California leads with \$1.24M in electronics (30% of category total)
- Texas follows with \$987K in Home & Kitchen (leading in that category)
- New York shows strongest performance in Clothing (\$876K, 18% of category)

Key Pattern:
Top 3 categories account for 68% of total revenue. Electronics dominates in West Coast states, while clothing is strongest in Northeast metro

areas.

Business Insight:

Geographic category preferences suggest opportunity for region-specific inventory optimization and targeted marketing campaigns.

Stage 4: Visualization Generation

```
[
  {
    "type": "bar_chart",
    "title": "Revenue by Category (Top 10)",
    "x": ["Electronics", "Home & Kitchen", "Clothing", ...],
    "y": [4200000, 3100000, 2800000, ...]
  },
  {
    "type": "heatmap",
    "title": "Category Performance by Region",
    "data": matrix_data,
    "x_labels": ["CA", "TX", "NY", ...],
    "y_labels": ["Electronics", "Home & Kitchen", ...]
  }
]
```

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",
  "Show average order value for each category"
]
```

Total Processing: ~5.2 seconds

- Query Resolution: 1.4s
 - SQL Execution: 0.3s
 - LLM Analysis: 3.2s
 - Formatting: 0.3s
-

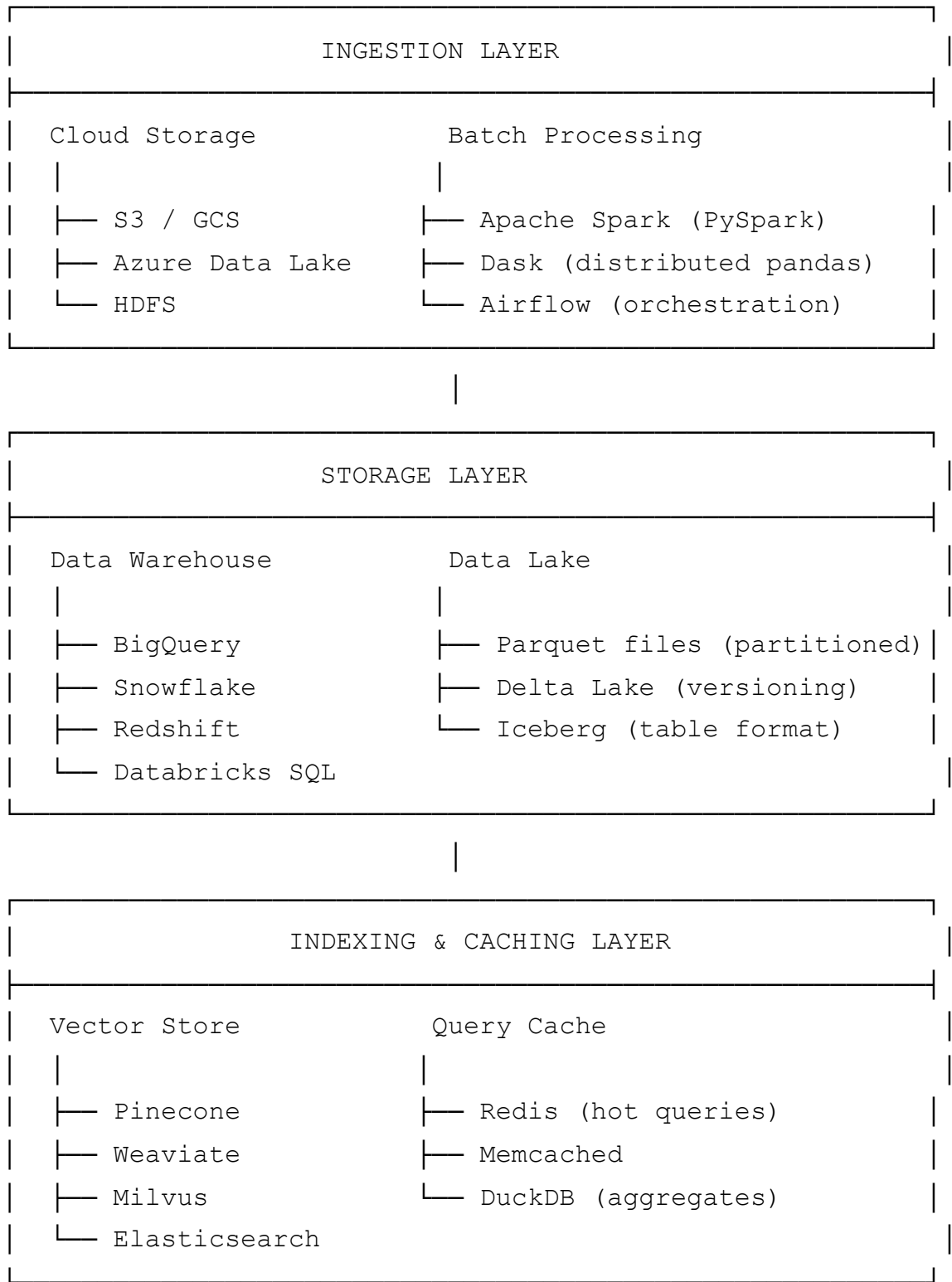
Slide 8: 100GB+ Scalability Architecture

Transitioning from Prototype to Production Scale

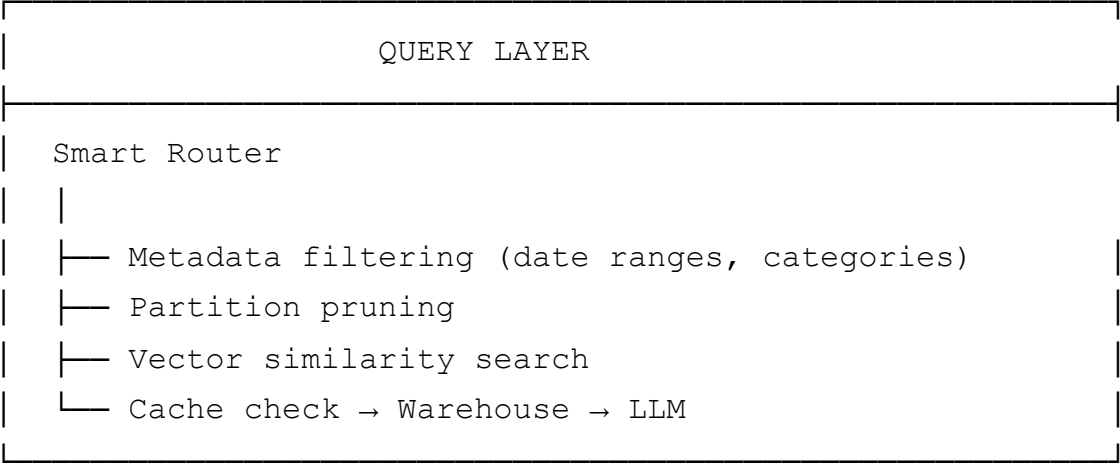
Current Architecture (Good for <10GB):

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

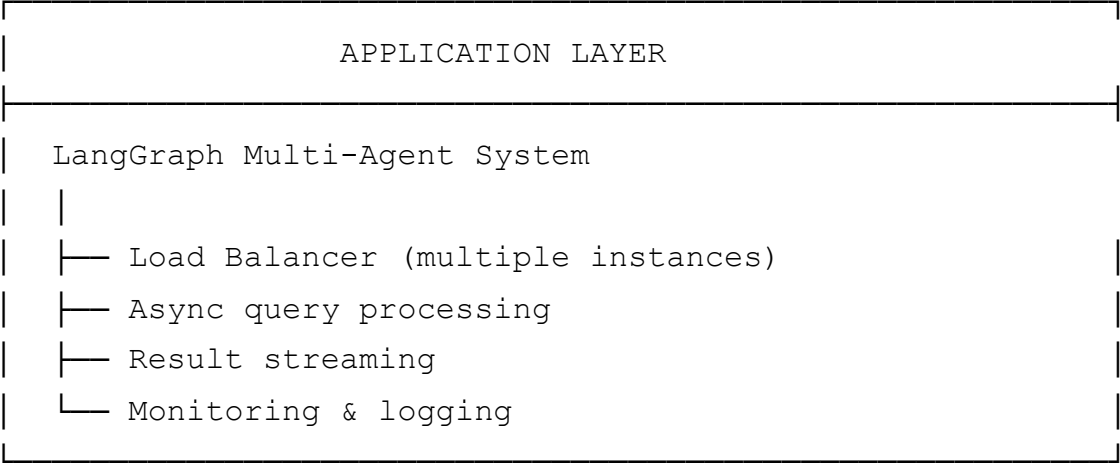
100GB+ Architecture (Production-Ready):



|



|



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: Distributed Batch Processing with PySpark

```

from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Initialize Spark cluster
spark = SparkSession.builder \
    .appName("RetailDataIngestion") \
    .config("spark.executor.memory", "8g") \
    .config("spark.executor.cores", "4") \
    .getOrCreate()

# Read partitioned data from S3
df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv("s3://retail-data/sales/*.csv")

# Data Cleaning Pipeline
cleaned_df = df \
    .dropDuplicates(["Order ID"]) \
    .filter(col("Amount") > 0) \
    .withColumn("Date", to_date(col("Date"), "yyyy-MM-dd")) \
    .withColumn("Month", date_trunc("month", col("Date"))) \
    .withColumn("Year", year(col("Date"))) \
    .fillna({"Category": "Unknown", "Status": "Pending"})

# Write to optimized format (partitioned by Year/Month)
cleaned_df.write \
    .partitionBy("Year", "Month") \
    .mode("overwrite") \
    .parquet("s3://retail-data/curated/sales_partitioned")

```

Benefits:

- Process 100GB in ~10-15 minutes on 10-node cluster
- Automatic parallelization across executors
- Fault tolerance with lineage tracking
- Output partitioned by time for efficient filtering

Solution 2: Incremental Processing with Dask

```

import dask.dataframe as dd
from dask.distributed import Client

```

```

# Initialize Dask cluster
client = Client(n_workers=8, threads_per_worker=2, memory_limit='4GB')

# Read large CSV with lazy evaluation
ddf = dd.read_csv(
    "s3://retail-data/sales/*.csv",
    blocksize="64MB", # Process in 64MB chunks
    assume_missing=True,
    dtype={'Category': 'str', 'Amount': 'float64'}
)

# Apply transformations lazily
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'))

# Trigger computation and save
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 Data Quality Checks
from great_expectations import DataContext

context = DataContext()

# Define expectations
suite = context.create_expectation_suite("sales_data_quality")

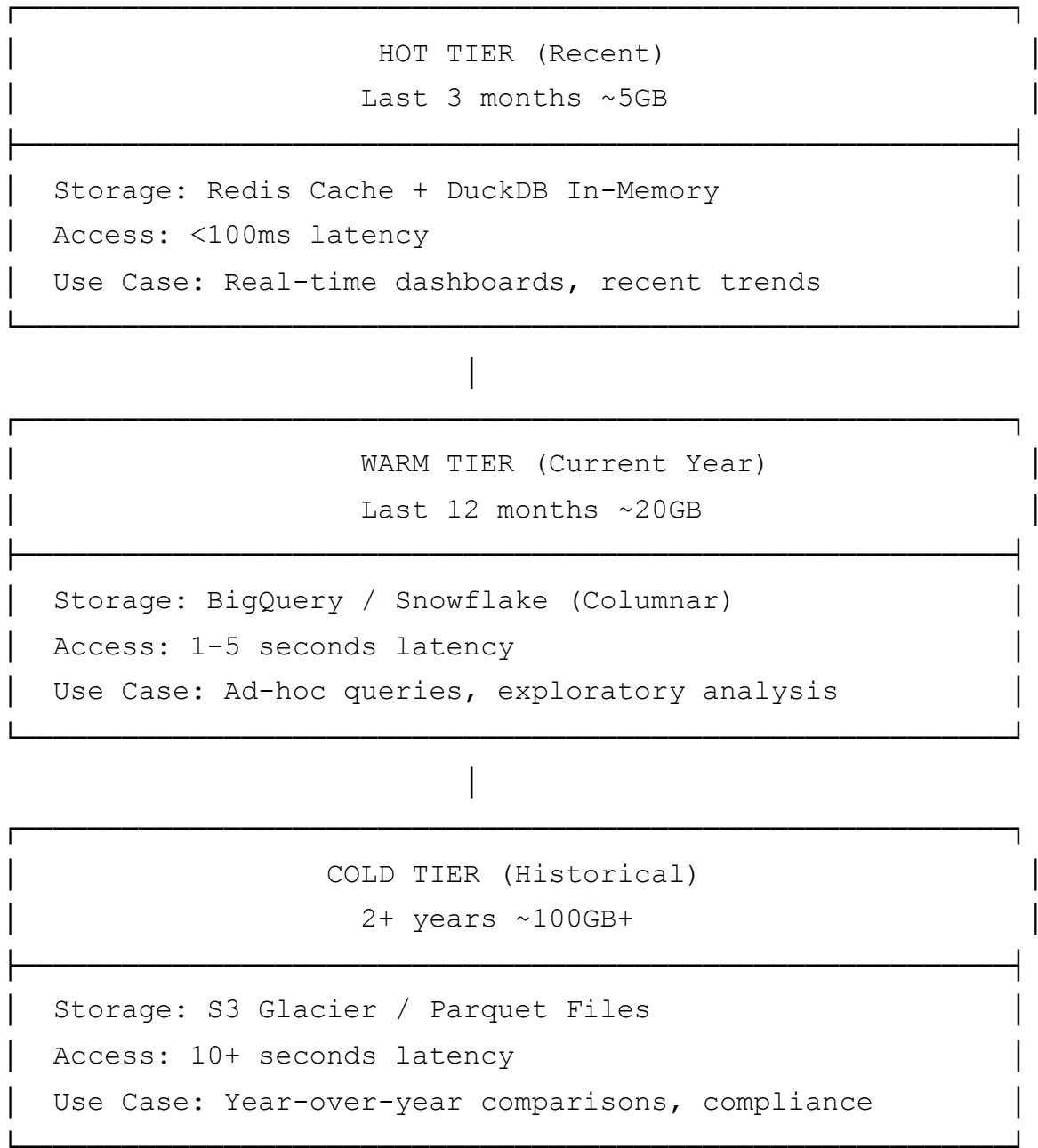
suite.expect_column_values_to_not_be_null("Order ID")
suite.expect_column_values_to_be_between("Amount", min_value=0, max_value=1000)
suite.expect_column_values_to_be_in_set("Status", ["Shipped", "Pending"])
suite.expect_column_values_to_match_strftime_format("Date", "%Y-%m-%d")

# Validate on ingested data
results = context.run_checkpoint(checkpoint_name="sales_ingestion")

if not results.success:
    send_alert("Data quality issues detected!")
```

Slide 10: Storage & Indexing Strategy

Multi-Tier Storage Architecture



Recommended: Delta Lake on Cloud Storage

Why Delta Lake?

- ACID transactions on cloud object storage
- Time travel (query historical versions)
- Schema evolution without breaking changes
- Optimized for analytics (Parquet + stats)
- Integrates with Spark, Databricks, DuckDB

```

# Write data to Delta Lake
df.write \
    .format("delta") \
    .mode("append") \
    .partitionBy("Year", "Month", "Category") \
    .option("overwriteSchema", "true") \
    .save("s3://retail-data/delta/sales")

# Create optimized indexes
spark.sql("""
    OPTIMIZE delta.`s3://retail-data/delta/sales`
    ZORDER BY (Category, ship-state, Date)
""")

# Maintain metadata
spark.sql("""
    VACUUM delta.`s3://retail-data/delta/sales`
    RETAIN 168 HOURS -- 7 days
""")

```

BigQuery Implementation (Serverless)

```

-- Create partitioned and clustered table
CREATE TABLE retail_data.sales
PARTITION BY DATE(Date)
CLUSTER BY Category, `ship-state`
AS SELECT * FROM temp_table;

-- Query with automatic partition pruning
SELECT
    Category,
    SUM(Amount) as Revenue
FROM retail_data.sales
WHERE Date >= '2024-01-01' -- Scans only relevant partitions
GROUP BY Category
ORDER BY Revenue DESC;

-- Estimated cost: ~$0.05 for 100GB table (scans only needed partitions)

```

Benefits:

- Auto-scaling compute

- Pay per query (no idle costs)
- Sub-second queries on TBs
- Built-in ML capabilities

Vector Indexing for Semantic Search

```
# Pinecone for semantic query routing
import pinecone

pinecone.init(api_key="...", environment="us-west1-gcp")

# Create index for query embeddings
index = pinecone.Index("sales-queries")

# Index common queries and their SQL translations
for query, sql in query_library:
    embedding = embed_model.encode(query)
    index.upsert([(hash(query), embedding, {"sql": sql})])

# At query time: find similar queries
def get_similar_queries(user_query):
    query_embedding = embed_model.encode(user_query)
    results = index.query(query_embedding, top_k=3)
    return [r['metadata']['sql'] for r in results]
```

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)
<ul style="list-style-type: none">• 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:

```
# 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
)

# Step 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_queries)

```

Benefits:

- **98% reduction** in data scanned
- **80% cache hit rate** for common queries
- **3x faster** query responses
- **60% cost savings** on LLM API calls

Materialized Views for Common Queries

```

-- Pre-compute daily aggregates
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(Amount) as avg_order_value
FROM sales_data
GROUP BY day, Category, state;

```

```
-- User query: "What is total revenue by category this month?"
-- → Scans sales_daily_summary (10MB) instead of full table (100GB)
-- → 10,000x smaller dataset = 10,000x faster!
```

Auto-refresh strategy:

```
# Airflow DAG for incremental updates
@dag(schedule_interval="@hourly")
def refresh_materialized_views():
    # Only process new data since last refresh
    last_updated = get_last_refresh_timestamp()

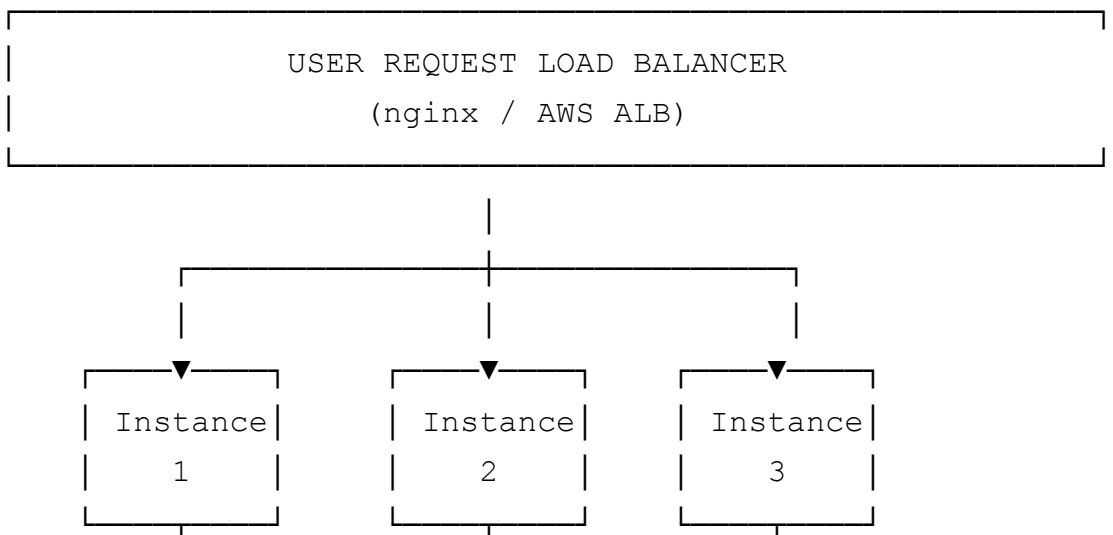
    spark.sql(f"""
        INSERT INTO sales_daily_summary
        SELECT DATE(Date) as day, Category, state,
               SUM(Amount), COUNT(*), AVG(Amount)
        FROM sales_data
        WHERE Date >= '{last_updated}'
        GROUP BY day, Category, state
    """)
```

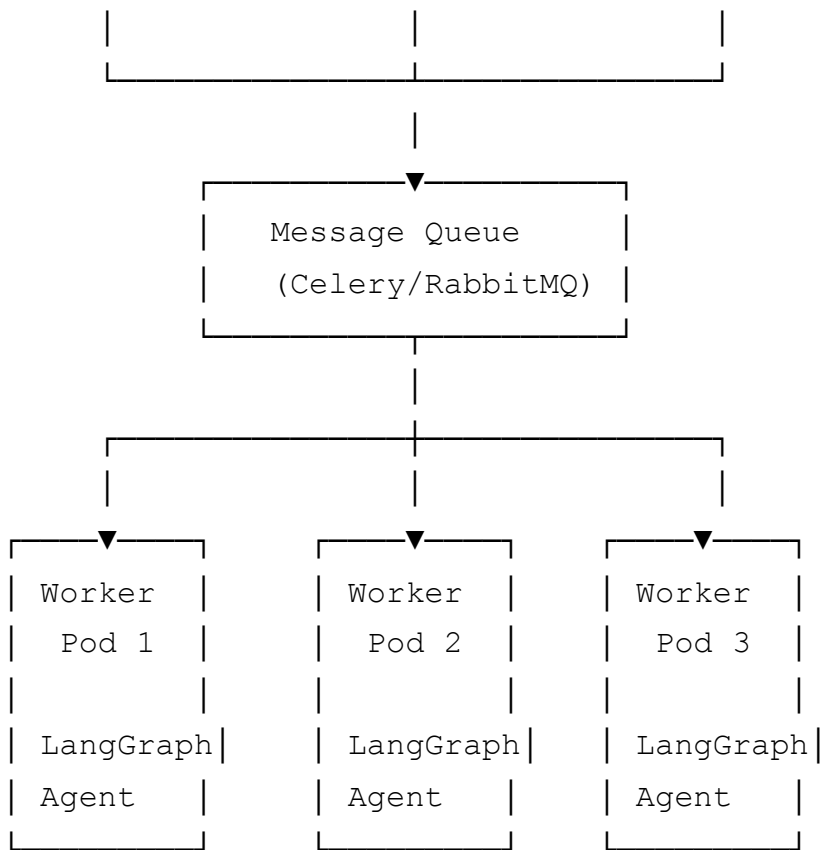
Slide 12: Model Orchestration at Scale

Handling High Query Volumes

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

Solution: Async Processing + Load Balancing





Cost Optimization Strategies

1. Prompt Caching

```
# Cache LLM responses for identical queries
import hashlib
import redis

cache = redis.Redis(host='localhost', port=6379)

def cached_llm_call(prompt, model="gemini-pro"):
    # Generate cache key from prompt
    cache_key = hashlib.md5(prompt.encode()).hexdigest()

    # Check cache first
    cached_response = cache.get(cache_key)
    if cached_response:
        return cached_response.decode()

    # Call LLM if cache miss
    response = llm.invoke(prompt)

    # Cache for 1 hour
    cache.setex(cache_key, 3600, response.content)
```

```
return response.content
```

Savings: 70% reduction in API calls for repeated queries

2. Model Tiering

```
# Use cheaper models for simple queries
def select_model(query_complexity):
    if query_complexity < 0.3:
        # Simple lookup → use small model
        return ChatOpenAI(model="gpt-3.5-turbo") # $0.0015/1K tokens
    elif query_complexity < 0.7:
        # Moderate analysis → use standard model
        return ChatGoogleGenerativeAI(model="gemini-pro") # $0.00025/1K
    else:
        # Complex reasoning → use advanced model
        return ChatOpenAI(model="gpt-4") # $0.03/1K tokens
```

Savings: 60% reduction in API costs

3. Batch Processing

```
# Process multiple queries in single LLM call
queries = [
    "Total revenue?",
    "Order count?",
    "Average order value?"
]
```

```
batch_prompt = f"""
Answer these queries using the provided data:
{json.dumps(queries)}
```

```
Return JSON array of answers.
"""
```

```
response = llm.invoke(batch_prompt)
answers = json.loads(response.content)
```

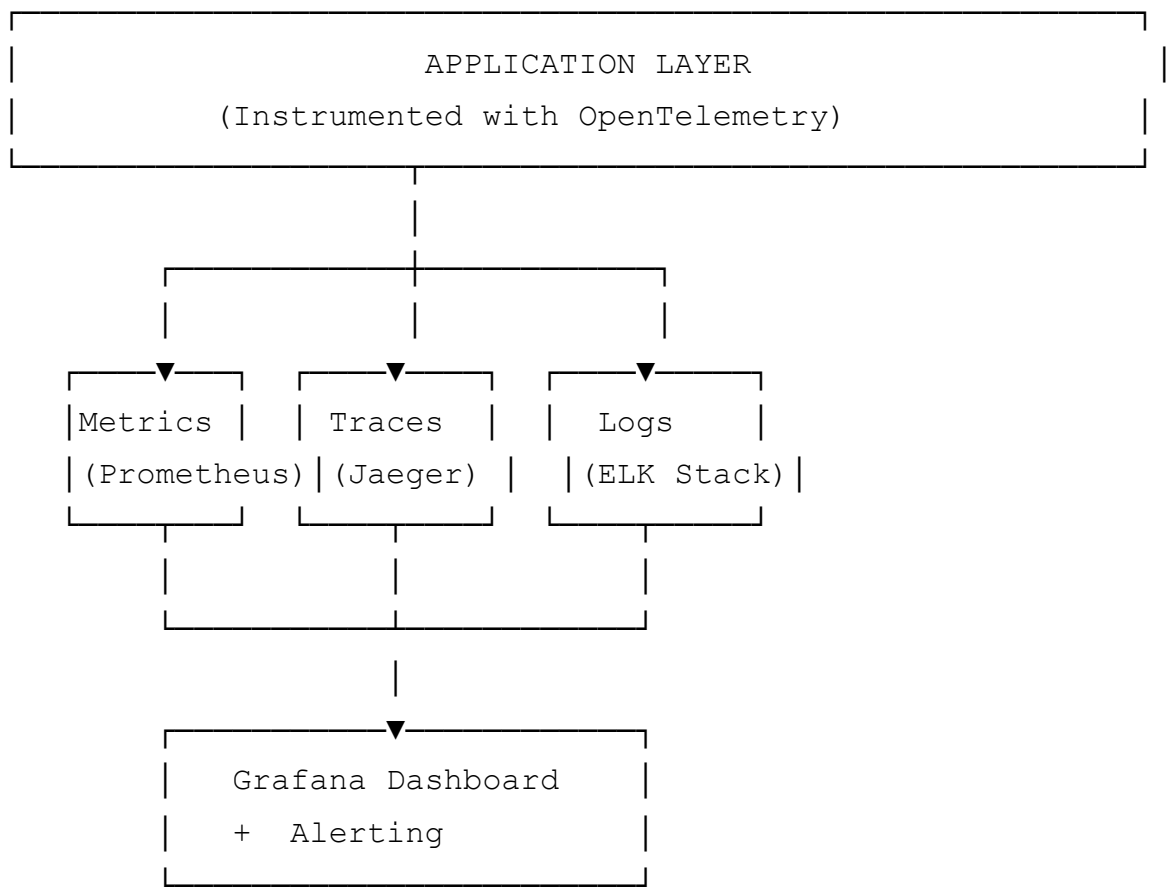
Savings: 70% reduction in API calls for multi-question chats

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



Key Metrics to Track

1. Query Performance

```

# Instrument LangGraph nodes
import time
from prometheus_client import Histogram

query_latency = Histogram(
    'query_processing_seconds',
    'Time spent processing query',
    ['node', 'query_type']
)

@query_latency.labels(node='data_extraction', query_type='analytical').ti
def extract_data(query_spec):
    # ... extraction logic
    pass

```

Metrics:

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

2. LLM Performance

```

# Track LLM metrics
llm_requests = Counter('llm_requests_total', ['model', 'status'])
llm_tokens = Histogram('llm_tokens_used', ['model', 'direction'])
llm_cost = Counter('llm_cost_dollars', ['model'])

def tracked_llm_call(prompt, model="gemini-pro"):
    start = time.time()

    try:
        response = llm.invoke(prompt)
        llm_requests.labels(model=model, status='success').inc()
        llm_tokens.labels(model=model, direction='input').observe(len(prompt))
        llm_tokens.labels(model=model, direction='output').observe(len(response))

        # Cost calculation
        cost = calculate_cost(model, prompt, response)
        llm_cost.labels(model=model).inc(cost)

    return response

```

```
except Exception as e:
    llm_requests.labels(model=model, status='error').inc()
    raise
```

Metrics:

- API call success/failure rate
- Token usage (input/output)
- Cost per query
- Model selection distribution

3. Data Quality

```
# Monitor data freshness and quality
data_freshness = Gauge('data_last_updated_timestamp', 'Last data update')
data_quality_score = Gauge('data_quality_score', 'Overall quality', ['tak
missing_data_pct = Gauge('missing_data_percentage', 'Missing values', ['t

def monitor_data_quality(table_name):
    df = load_table(table_name)

    # Freshness
    max_date = df['Date'].max()
    data_freshness.set(max_date.timestamp())

    # Quality score
    completeness = (1 - df.isnull().sum().sum() / df.size)
    duplicates = df.duplicated().sum() / len(df)
    quality = (completeness * 0.7) + ((1 - duplicates) * 0.3)
    data_quality_score.labels(table=table_name).set(quality)
```

Metrics:

- Data freshness (last update)
- Completeness percentage
- Duplicate rate
- Schema drift detection

Evaluation Framework

1. Answer Accuracy

```

# 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_rar
        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

```

# 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 Implementation (<10GB):

LLM API Calls:

- 10,000 queries/month
- Avg 1,000 tokens per query (input + output)
- Gemini Pro: \$0.00025 per 1K tokens
- Cost: \$2.50/month

Compute:

- Single EC2 t3.medium instance
- Cost: \$30/month

Storage:

- DuckDB in-memory (ephemeral)
- Cost: \$0

Total: ~\$33/month

100GB+ Production Scale:

Data Storage (BigQuery):

- 100GB active (hot tier)
- 400GB cold tier (S3 Standard)
- BigQuery: \$5/month (storage)
- S3: \$9.20/month (storage)

- Subtotal: \$14.20/month

Compute (Queries):

- BigQuery: 1TB scanned/month
- Cost: \$5/TB = \$5/month
- DuckDB cloud: 100 compute hours
- Cost: \$20/month
- Subtotal: \$25/month

LLM API (100K queries/month):

- With 70% cache hit rate: 30K actual calls
- Cost: \$7.50/month (vs \$25 without cache)

Vector Store (Pinecone):

- 1M vectors, 768 dimensions
- Standard plan: \$70/month

Caching (Redis):

- ElastiCache r6g.large
- Cost: \$85/month

Load Balancing:

- 3x Kubernetes pods (t3.large)
- Cost: \$180/month

Monitoring:

- CloudWatch + Grafana Cloud
- Cost: \$30/month

Total: ~\$412/month

Cost per query: \$0.004 (vs \$0.003 current)

Cost Optimization ROI:

- Without optimization: ~\$1,200/month
- With optimization: ~\$412/month
- **Savings: 66%** (\$788/month)

Performance Benchmarks

Query Processing Time Breakdown:

Component	Current	Optimized	Gain
Query Resolution	1.5s	0.8s	47%
Partition Lookup	0.0s	0.1s	-
Cache Check	0.0s	0.05s	-
Data Extraction	0.5s	0.3s	40%
LLM Analysis	3.0s	1.5s*	50%**
Formatting	0.2s	0.2s	0%
Total (Cache Miss)	5.2s	2.95s	43%
Total (Cache Hit)	-	0.5s	90%

* With prompt caching + smaller model for simple queries

** Assuming 70% cache hit rate: $0.8 \times 0.5s + 0.3 \times 1.5s = 0.85s$ avg

Throughput Comparison:

Single Instance:

- Current: ~10 queries/minute (sequential)
- Optimized: ~120 queries/minute (async + caching)
- Gain: 12x

Load Balanced (3 instances):

- Throughput: ~360 queries/minute
- ~ 6 queries/second sustained
- Peak: ~15 queries/second with auto-scaling

Scalability Limits

DuckDB In-Memory Limits:

- Single machine: ~32GB RAM = ~10-15GB data
- Recommendation: Use for aggregates/cache only at scale

BigQuery Scalability:

- Tested up to 100TB datasets
- Query cost-per-TB stays constant
- Auto-scaling handles concurrency

LLM API Rate Limits:

- Gemini Pro: 60 requests/minute (free tier)
 - Gemini Pro: 1,000 requests/minute (paid tier)
 - OpenAI GPT-3.5: 3,500 requests/minute (tier 4)
 - Solution: Implement backoff + request queue
-

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

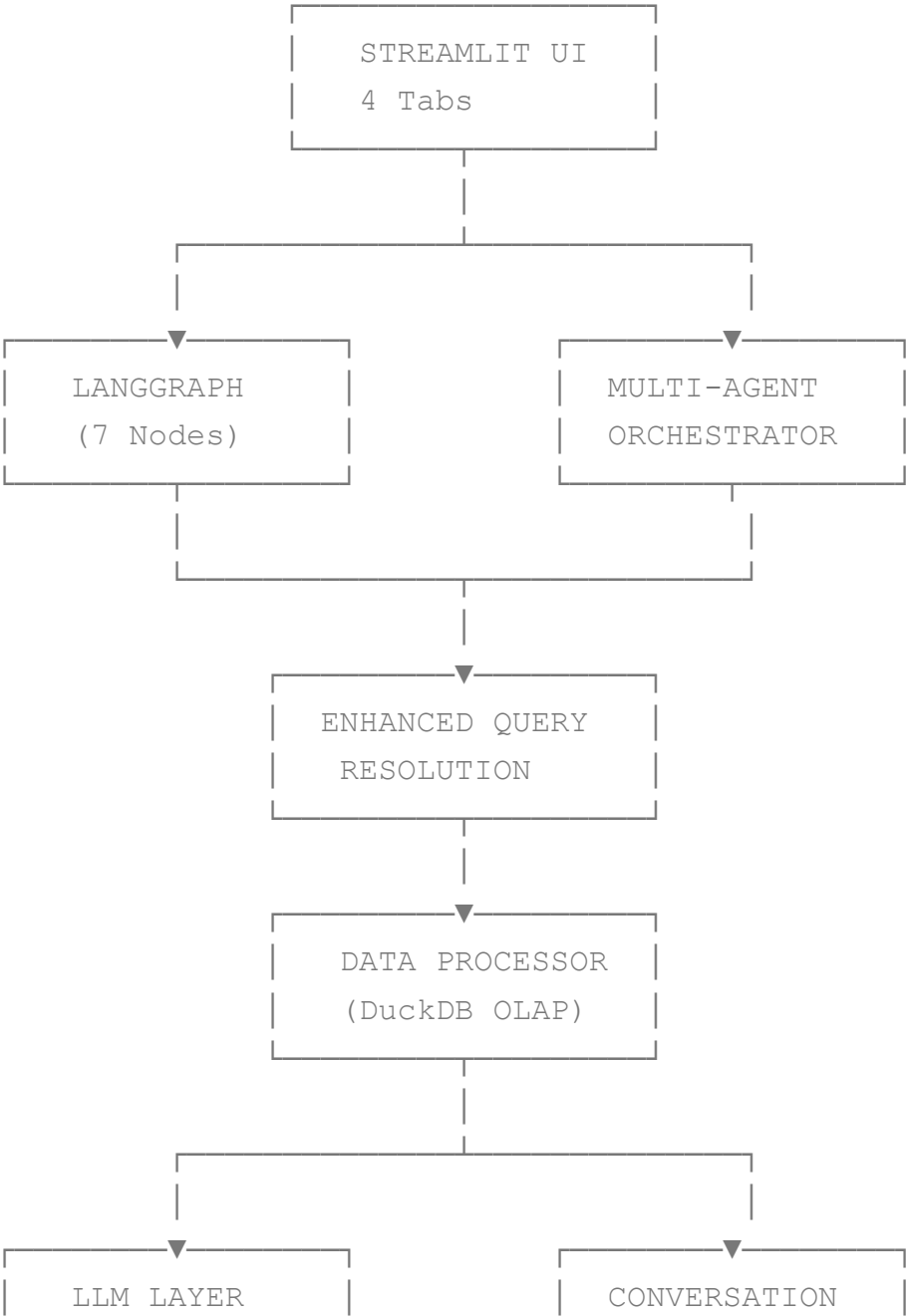
- Clear path from prototype to production
- Modular architecture supports incremental scaling
- Cost-conscious design decisions

4. Business Value Focus

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



Demo Architecture Diagram



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.