Multimodal Property RAG System – Technical Analysis Report

Executive Summary

This technical report analyzes the **Multimodal Retrieval-Augmented Generation (RAG)** system designed for property data analytics, focusing on the **dual analytics engine integration** using **Google Gemini** and **OpenAI ChatGPT**, and diagnosing the issue where ChatGPT analytics results are not displayed in the Flask frontend.

Key Findings

- The system successfully integrates **Gemini** and **ChatGPT** for parallel pandas code generation and execution.
- **ChatGPT analytics function correctly** in backend testing but fail to appear in the Flask frontend.
- **Root cause identified:** Missing OpenAI API key in the production environment.
- The architecture demonstrates **robust fault tolerance**, **safe code execution**, and **strong fallback mechanisms**.

---

1. System Architecture Overview

1.1 Core Components

Vector Database Layer

- **Technology:** Pinecone (Serverless)
- **Embeddings:** HuggingFace SentenceTransformer (`all-MiniLM-L6-v2`)
- **Dimension:** 384

- **Records:** 147,666 property vectors
- **Index Name:** `property-data-rag`

Analytics Layer (Dual Engine)

- **Gemini Analytics Agent:** Google Gemini 2.5 Flash
- **ChatGPT Analytics Agent:** OpenAI GPT-4o-mini
- **Function:** Natural language → Pandas code translation
- **Execution:** Safe code execution with strict heuristics and keyword filtering

Response Generation Layer

- **Primary Composer:** GPT-4o-mini
- **Fallback Composer:** Gemini 2.5 Flash
- **Input Context:** Combined output from vector search and both analytics agents

Frontend Layer

- **Framework:** Flask
- **UI Stack:** HTML, CSS, JavaScript (ES6)
- **Theme:** Dark, responsive design with toggle-based output panels
- **Runtime:** AJAX-driven request–response cycle

---

2. Data Pipeline

- Property CSV → EDA Processing → Cleaned Data → Embeddings → Pinecone Index
- ↓
- Analytics DataFrame → Gemini & ChatGPT Agents → Pandas Code Execution
- ↓

- User Query → Vector Search + Dual Analytics → Context Assembly → AI Response

---

3. Technical Implementation Analysis

3.1 Multimodal Analytics Implementation

Gemini Analytics Agent

```
class DataAnalyticsAgent:
    def analyze(self, nl_query: str, top_k: int = 7) -> Dict[str, Any]:
        # Uses Gemini 2.5 Flash for pandas code generation
        # Implements safety filters and fallback heuristics
        return {"context": str, "rows": list, "generated_code": str}
```

Strengths

- Effective fallback heuristics
- Secure execution filters
- Reliable code extraction and structured output

ChatGPT Analytics Agent

```
class ChatGPTAnalyticsAgent:
    def analyze(self, nl_query: str, top_k: int = 7) -> Dict[str, Any]:
        # Uses OpenAI GPT-4o-mini for pandas code generation
        # Implements safety mechanisms and markdown code extraction
```

Highlights

- Dual API strategy (OpenAI SDK + LangChain fallback)
- Enhanced logging for debugging

- Identical prompt structure to Gemini for consistency

3.2 Dual Execution Architecture

```
def analyze_both(self, query: str, top_k: int = 7) -> Dict[str, Any]:
    """Run both Gemini and ChatGPT analytics and return both results."""
    gem = self.analytics_agent.analyze(query, top_k=top_k)
    chg = self.chatgpt_analytics_agent.analyze(query, top_k=top_k)
    return {"gemini": gem, "chatgpt": chg}
```

- **Design Pattern:** Currently sequential; parallel execution planned for future optimization.

3.3 Frontend Integration

Flask API Endpoint

```
@app.route('/api/ask', methods=['POST'])
def api_ask():
    both_analytics = RAG_AGENT.analyze_both(q, top_k=7)
    return jsonify({
        "answer": str(answer),
        "analytics": both_analytics,
        "matches": matches_serializable
    })
```

JavaScript Frontend Rendering

```
const gemCode = j.analytics?.gemini?.generated_code || '';
const chgCode = j.analytics?.chatgpt?.generated_code || '';
document.getElementById('chatgpt-code-content').textContent =
    chgCode || 'No pandas code generated';
```

---

4. Issue Analysis: ChatGPT Analytics Not Displaying

4.1 Symptom

- ChatGPT panel shows “No pandas code generated.”
- Backend tests confirm ChatGPT analytics function correctly.
- Issue isolated to Flask production runtime.

4.2 Root Cause: Missing OpenAI API Key

- ○ OPENAI_API_KEY found: False

Without this key, the OpenAI client fails silently, resulting in empty analytics responses.

4.3 Secondary Causes

1. Improper `.env` loading due to different working directories.
2. Missing error reporting for failed API initialization.
3. Caching of invalid client state (`None`) on startup.

4.4 Code Flow Comparison

Working Path (Backend Test)

- ○ test_rag_system.py → pinecone_rag_setup.py → ChatGPTAnalyticsAgent.analyze()
- ○ → get_openai_client() → OpenAI API → Returns valid pandas code

Failing Path (Flask Frontend)

- ○ app.py → RAG_AGENT.analyze_both() → ChatGPTAnalyticsAgent.analyze()

- → get_openai_client() → None (API key missing) → Empty output

---

5. Technical Recommendations

5.1 Immediate Fixes

Environment Configuration

```
○ # .env file
○ OPENAI_API_KEY=sk-proj-your-key-here
○ PINECONE_API_KEY=your-pinecone-key
○ GEMINI_API_KEY=your-gemini-key
```

Improved Client Initialization

```
○ def get_openai_client():
○     client = setup_openai()
○     if not client:
○         print("[ERROR] OpenAI client not initialized - missing API key")
○     return client
```

Frontend Error Display

```
○ if (!chgCode) {
○   document.getElementById('chatgpt-code-content').textContent =
○       'ChatGPT analytics unavailable - check API configuration';
○ }
```

5.2 Architecture Enhancements

Parallel Execution

```
○ import asyncio
○ from concurrent.futures import ThreadPoolExecutor
```

```
○
○   async def analyze_both_parallel(self, query: str, top_k: int = 7):
○       with ThreadPoolExecutor(max_workers=2) as executor:
○           gem = executor.submit(self.analytics_agent.analyze, query, top_k)
○           chg = executor.submit(self.chatgpt_analytics_agent.analyze, query, top_k)
○       return {"gemini": gem.result(), "chatgpt": chg.result()}
```

Health Check Endpoint

```
○   @app.route('/api/health')
○   def health_check():
○       return jsonify({
○           "pinecone": bool(RAG_AGENT.index),
○           "gemini": bool(RAG_AGENT.model),
○           "openai": bool(get_openai_client())
○       })
```

5.3 Performance Optimizations

Caching Analytics

```
○   from functools import lru_cache
○
○   @lru_cache(maxsize=1000)
○   def cached_analytics(query_hash, agent):
○       # Store recent query results to minimize API calls
○       pass
```

Batch Query Processing

```
○   def batch_analyze(self, queries: List[str], top_k: int = 7):
○       return [self.analyze_both(q, top_k) for q in queries]
```

---

6. Security Considerations

6.1 API Key Management

- Use secret management (AWS Secrets Manager, Azure Key Vault).
- Avoid storing credentials in `.env` in production.
- Implement API key rotation policies.

6.2 Safe Code Execution

```
forbidden = ['os.', 'sys.', 'eval(', 'exec(', 'import ', '__']
for token in forbidden:
    code = code.replace(token, '# removed ')
```

6.3 Input Validation

```
def validate_query(q):
    return len(q) < 500 and "drop" not in q.lower()
```

---

7. Testing Strategy

Unit Tests

```
def test_chatgpt_analytics():
    agent = ChatGPTAnalyticsAgent(test_df)
    res = agent.analyze("2 bedroom under £2000")
    assert res["generated_code"]
```

Integration Tests

```
def test_multimodal_rag():
    r = PropertyRAGAgent().analyze_both("test query")
    assert "gemini" in r and "chatgpt" in r
```

Load Tests

Simulate 10 concurrent queries using threading to measure latency.

---

8. Deployment & Monitoring

Deployment Checklist

1. `.env` configured with all keys
2. Verify individual services (Pinecone, Gemini, OpenAI)
3. Run integration tests
4. Enable logging and monitoring

Logging Example

- logging.info(f"[ANALYTICS] ChatGPT | {duration:.2f}s | {query[:60]}")

Error Recovery

- if not openai_available:
-     return {"chatgpt": {"context": "Analytics unavailable", "rows": [], "generated_code": ""}}

---

9. Conclusion

The **Multimodal Property RAG System** showcases a robust architecture combining **semantic retrieval** and **AI-driven data analytics**. The dual-engine design enhances resilience and analytical diversity, offering improved reliability across varied data contexts.

Key Achievements

- Dual LLM integration (Gemini + ChatGPT)
- Intelligent fallback and error handling
- Safe code execution framework
- Real-time analytics UI

Required Actions

1. Add missing OpenAI API key.
2. Implement real-time health checks.
3. Introduce parallel execution and caching.

Once configured, the system is **production-ready** and capable of scaling for enterprise-grade property analytics.