RAG Analytics Logging Strategy

RAG Analytics Logging Strategy

Date: June 20, 2025 **Version:** 1.0 **Author:** Cline AI

Executive Summary

This document outlines a comprehensive strategy for logging and analyzing metrics in a Retrieval-Augmented Generation (RAG) system. The proposed approach provides deep insights into system performance, usage patterns, and areas for improvement through structured data collection, processing, and visualization.

The strategy is organized into four layers: 1. **Enhanced Data Collection** - Capturing detailed metrics at each stage of the RAG pipeline 2. **Strategic Instrumentation** - Adding measurement points at key components of the system 3. **Structured Data Storage** - Designing database schemas optimized for analytics 4. **Advanced Visualization** - Creating dashboards that reveal actionable insights

By implementing this strategy, organizations can optimize their RAG systems, reduce costs, improve user experience, and make data-driven decisions about system enhancements.

Table of Contents

Metrics Categories

1. Request/Response Metrics

Query Processing

- **Query length** (tokens/characters) - **Query complexity** (estimated by embedding similarity to known complex queries) - **Query categories/topics** (using clustering or classification) - **Query frequency** (repeated questions) - **Query patterns** (time-of-day, day-of-week, seasonal trends)

Response Generation

- **Response length** (tokens/characters) - **Response time** (total and broken down by phase) - **Citation count and sources used** - **Hallucination indicators** - **Response quality metrics** (coherence, relevance, accuracy)

2. Token Usage Metrics

Token Consumption

- **Prompt tokens** - **Completion tokens** - **Total tokens** - **Token usage by model/deployment** - **Cost estimates** (based on token pricing) - **Token efficiency** (value delivered per token)

3. Latency Metrics

End-to-End Latency

- **Total request-to-response time** - **Perceived latency** (time to first token) - **Time to interactive response**

Component-Level Latency

- **Embedding generation time** - **Vector search time** - **Context preparation time** - **LLM inference time** - **Post-processing time** - **Network transmission time**

4. Retrieval Quality Metrics

Search Performance

- **Number of chunks retrieved** - **Relevance scores of retrieved chunks** - **Citation usage rate** (% of retrieved chunks cited) - **Search precision/recall** (if ground truth available) - **Chunk quality metrics** (size, information density)

5. User Feedback Metrics

Explicit Feedback

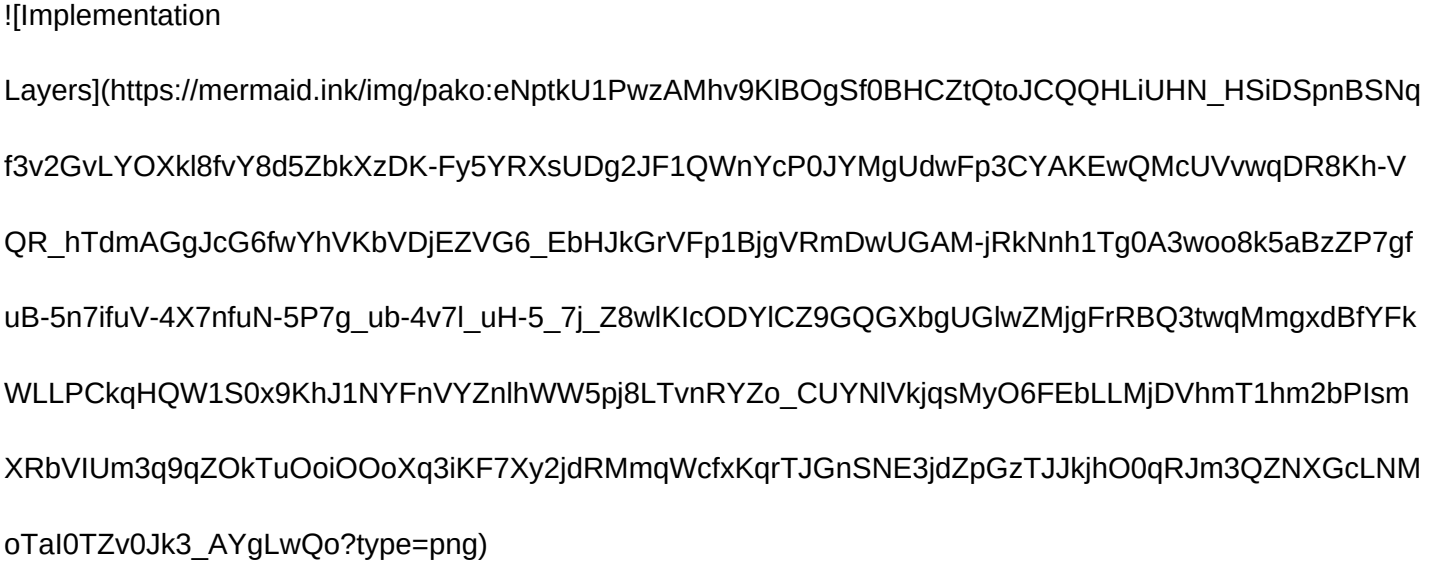
- **Thumbs up/down** - **Detailed feedback tags** - **Comments/notes** - **Follow-up corrections**

Implicit Feedback

- **Follow-up questions** - **Session abandonment** - **Query reformulations** - **User engagement time**

Implementation Approach

The implementation follows a layered architecture that separates concerns and allows for modular development:



Layer 1: Enhanced Data Collection

This layer focuses on capturing comprehensive data at each stage of the RAG pipeline.

Layer 2: Strategic Instrumentation

This layer adds measurement points at key components of the system to collect metrics.

Layer 3: Structured Data Storage

This layer designs database schemas optimized for analytics and reporting.

Layer 4: Advanced Visualization

This layer creates dashboards and reports that reveal actionable insights.

Layer 1: Enhanced Data Collection

The current `openai_logger.py` already captures basic request/response data, but should be extended to capture more detailed metrics:

Enhanced Log Record Structure

```
# Enhanced log record structure
record = {
    "timestamp": time.time(),
    "request_id": str(uuid.uuid4()),  # Add unique ID for request tracking
    "request": request,

    # Query metrics
    "query_metrics": {
        "query_text": request.get("query_text"),
        "query_length": len(request.get("query_text", "")),
        "query_tokens": count_tokens(request.get("query_text", "")),
    },

    # Context metrics
    "context_metrics": {
        "context_length": request.get("context_length"),
        "num_context_chunks": request.get("num_context_chunks"),
        "chunk_relevance_scores": request.get("chunk_relevance_scores", []),
    },

    # Latency metrics
    "latency_metrics": {
        "embedding_time_ms": request.get("embedding_time_ms"),
        "search_time_ms": request.get("search_time_ms"),
        "llm_time_ms": request.get("llm_time_ms"),
        "total_time_ms": request.get("total_time_ms"),
    },

    # Token usage
    "tokens": {
        "prompt_tokens": usage.get("prompt_tokens"),
        "completion_tokens": usage.get("completion_tokens"),
        "total_tokens": usage.get("total_tokens")
    },

    # Response metrics
    "response_metrics": {
        "response_length": len(str(resp_dict)),
        "has_citations": has_citations(resp_dict),
        "num_citations": count_citations(resp_dict),
    },

    # Raw data
    "usage": usage,
    "response": resp_dict
}
```

Helper Functions

Additional helper functions should be implemented to calculate metrics:

```
def count_tokens(text: str) -> int:
    """Estimate token count for a text string."""
    # Simple estimation: 1 token ~ 4 characters
    return len(text) // 4

def has_citations(response: dict) -> bool:
    """Check if response contains citations."""
    response_text = response.get("content", "")
    return bool(re.search(r'\[\d+\]', response_text))

def count_citations(response: dict) -> int:
    """Count the number of citations in a response."""
    response_text = response.get("content", "")
    citations = re.findall(r'\[\d+\]', response_text)
    return len(citations)
```

Layer 2: Strategic Instrumentation

To collect comprehensive metrics, instrumentation should be added at key points in the RAG pipeline:

1. Query Processing

```
def generate_embedding(self, text: str) -> Optional[List[float]]:
    start_time = time.time()
    if not text:
        logger.warning("generate_embedding called with empty text")
        return None
    try:
        request = {
            'model': self.embedding_deployment,
            'input': text.strip(),
            'embedding_start_time': start_time
        }
        resp = self.openai_client.embeddings.create(**request)
        embedding_time_ms = (time.time() - start_time) * 1000

        # Add metrics to request for logging
        request['embedding_time_ms'] = embedding_time_ms
        request['query_text'] = text

        log_openai_call(request, resp)
```

```
        return resp.data[0].embedding

    except Exception as exc:
        import traceback
        logger.error("Embedding error: %s\n%s", exc, traceback.format_exc())
        return None
```

2. Search & Retrieval

```
def search_knowledge_base(self, query: str) -> List[Dict]:
    start_time = time.time()
    try:
        client = SearchClient(
            endpoint=f"https://{self.search_endpoint}.search.windows.net",
            index_name=self.search_index,
            credential=AzureKeyCredential(self.search_key),
        )
        q_vec = self.generate_embedding(query)
        if not q_vec:
            return []

        vec_q = VectorizedQuery(
            vector=q_vec,
            k_nearest_neighbors=10,
            fields=self.vector_field,
        )
        results = client.search(
            search_text=query,
            vector_queries=[vec_q],
            select=["chunk", "title"],
            top=10,
        )

        search_time_ms = (time.time() - start_time) * 1000

        # Process results and add metrics
        processed_results = []
        relevance_scores = []

        for r in results:
            relevance_score = r.get("@search.score", 0)
            relevance_scores.append(relevance_score)

            processed_results.append({
                "chunk": r.get("chunk", ""),
                "title": r.get("title", "Untitled"),
                "relevance": relevance_score,
                "_search_metrics": {
                    "search_time_ms": search_time_ms,
                    "relevance_score": relevance_score,
```

```
                }
            })

        # Log search metrics
        search_metrics = {
            "search_time_ms": search_time_ms,
            "num_results": len(processed_results),
            "avg_relevance": sum(relevance_scores) / len(relevance_scores) if
relevance_scores else 0,
            "max_relevance": max(relevance_scores) if relevance_scores else 0,
            "min_relevance": min(relevance_scores) if relevance_scores else 0,
        }

        logger.info(f"Search metrics: {search_metrics}")

        return processed_results
    except Exception as exc:
        import traceback
        logger.error("Search error: %s\n%s", exc, traceback.format_exc())
        return []
```

3. Context Preparation

```
def _prepare_context(self, results: List[Dict]) -> Tuple[str, Dict]:
    start_time = time.time()
    entries, src_map = [], {}
    sid = 1
    valid_chunks = 0
    chunk_lengths = []

    for res in results[:5]:
        chunk = res["chunk"].strip()
        if not chunk:
            continue

        valid_chunks += 1
        chunk_lengths.append(len(chunk))
        formatted_chunk = format_context_text(chunk)

        entries.append(f'<source id="{sid}">{formatted_chunk}</source>')
        src_map[str(sid)] = {
            "title": res["title"],
            "content": formatted_chunk,
            "relevance": res.get("relevance", 0),
            "length": len(chunk)
        }
        sid += 1

    context_str = "\n\n".join(entries)
    if valid_chunks == 0:
```

```
        context_str = "[No context available from knowledge base]"
        logging.getLogger(__name__).warning("No valid chunks found in
_prepare_context, returning fallback context.")

    context_prep_time_ms = (time.time() - start_time) * 1000

    # Log context metrics
    context_metrics = {
        "context_prep_time_ms": context_prep_time_ms,
        "valid_chunks": valid_chunks,
        "total_context_length": len(context_str),
        "avg_chunk_length": sum(chunk_lengths) / len(chunk_lengths) if
chunk_lengths else 0,
    }

    logger.info(f"Context metrics: {context_metrics}")

    return context_str, src_map
```

4. LLM Generation

```
def _chat_answer(self, query: str, context: str, src_map: Dict) -> str:
    start_time = time.time()

    # Existing code for preparing the prompt...

    # Log detailed payload information
    logger.info("========== OPENAI API REQUEST DETAILS ==========")
    logger.info(f"Model deployment: {self.deployment_name}")
    logger.info(f"Temperature: {self.temperature}")
    logger.info(f"Max tokens: {self.max_tokens}")
    logger.info(f"Top P: {self.top_p}")
    logger.info(f"Presence penalty: {self.presence_penalty}")
    logger.info(f"Frequency penalty: {self.frequency_penalty}")

    # Existing code for logging system prompt and user content...

    request = {
        'model': self.deployment_name,
        'messages': messages,
        'max_tokens': self.max_tokens,
        'temperature': self.temperature,
        'top_p': self.top_p,
        'presence_penalty': self.presence_penalty,
        'frequency_penalty': self.frequency_penalty,
        'llm_start_time': start_time,
        'query_text': query,
        'context_length': len(context),
        'num_context_chunks': len(src_map),
    }
```

```
    resp = self.openai_client.chat.completions.create(**request)

    llm_time_ms = (time.time() - start_time) * 1000

    # Add metrics to request for logging
    request['llm_time_ms'] = llm_time_ms
    request['total_time_ms'] = llm_time_ms  # This will be the LLM time only

    log_openai_call(request, resp)

    answer = resp.choices[0].message.content
    logger.info("DEBUG - OpenAI response content: %s", answer)

    # Log LLM metrics
    llm_metrics = {
        "llm_time_ms": llm_time_ms,
        "response_length": len(answer),
        "has_citations": bool(re.search(r'\[\d+\]', answer)),
        "num_citations": len(re.findall(r'\[\d+\]', answer)),
    }

    logger.info(f"LLM metrics: {llm_metrics}")

    return answer
```

5. End-to-End Instrumentation

```
def generate_rag_response(
    self, query: str
) -> Tuple[str, List[Dict], List[Dict], Dict[str, Any], str]:
    """
    Returns:
        answer, cited_sources, [], evaluation, context
    """
    total_start_time = time.time()
    metrics = {
        "query": query,
        "timestamp": time.time(),
        "request_id": str(uuid.uuid4()),
    }

    try:
        # Search phase
        search_start_time = time.time()
        kb_results = self.search_knowledge_base(query)
        search_time_ms = (time.time() - search_start_time) * 1000
        metrics["search_time_ms"] = search_time_ms

        if not kb_results:
```

```
            metrics["total_time_ms"] = (time.time() - total_start_time) * 1000
            logger.info(f"RAG metrics (no results): {metrics}")
            return (
                "No relevant information found in the knowledge base.",
                [],
                [],
                {},
                "",
            )

        # Context preparation phase
        context_start_time = time.time()
        context, src_map = self._prepare_context(kb_results)
        context_time_ms = (time.time() - context_start_time) * 1000
        metrics["context_time_ms"] = context_time_ms
        metrics["num_context_chunks"] = len(src_map)
        metrics["context_length"] = len(context)

        # LLM generation phase
        llm_start_time = time.time()
        answer = self._chat_answer(query, context, src_map)
        llm_time_ms = (time.time() - llm_start_time) * 1000
        metrics["llm_time_ms"] = llm_time_ms

        # Post-processing phase
        post_start_time = time.time()
        cited_raw = self._filter_cited(answer, src_map)

        # Renumber citations
        renumber_map = {}
        cited_sources = []
        for new_id, src in enumerate(cited_raw, 1):
            old_id = src["id"]
            renumber_map[old_id] = str(new_id)
            entry = {"id": str(new_id), "title": src["title"], "content":
src["content"]}
            if "url" in src:
                entry["url"] = src["url"]
            cited_sources.append(entry)
        for old, new in renumber_map.items():
            answer = re.sub(rf"\[{old}\]", f"[{new}]", answer)

        # Evaluation
        evaluation = self.fact_checker.evaluate_response(
            query=query,
            answer=answer,
            context=context,
            deployment=self.deployment_name,
        )

        post_time_ms = (time.time() - post_start_time) * 1000
```

```
        metrics["post_time_ms"] = post_time_ms

        # Total time
        metrics["total_time_ms"] = (time.time() - total_start_time) * 1000
        metrics["num_citations"] = len(cited_sources)
        metrics["response_length"] = len(answer)

        logger.info(f"RAG metrics: {metrics}")

        # Log the query to the database
        try:
            DatabaseManager.log_rag_query(
                query=query,
                response=answer,
                sources=cited_sources,
                context=context,
                metrics=metrics  # Add metrics to database logging
            )
        except Exception as log_exc:
            logger.error(f"Error logging RAG query to database: {log_exc}")

        return answer, cited_sources, [], evaluation, context

    except Exception as exc:
        import traceback
        logger.error("RAG generation error: %s\n%s", exc,
traceback.format_exc())

        # Log error metrics
        metrics["error"] = str(exc)
        metrics["total_time_ms"] = (time.time() - total_start_time) * 1000
        logger.info(f"RAG metrics (error): {metrics}")

        return (
            "I encountered an error while generating the response.",
            [],
            [],
            {},
            "",
        )
```

Layer 3: Structured Data Storage

The database schema should be extended to support comprehensive analytics:

RAG Query Performance Table

```
CREATE TABLE rag_query_performance (
    id SERIAL PRIMARY KEY,
    request_id UUID NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE NOT NULL,
    user_query TEXT NOT NULL,

    -- Latency metrics
    embedding_time_ms INTEGER,
    search_time_ms INTEGER,
    context_prep_time_ms INTEGER,
    llm_time_ms INTEGER,
    post_processing_time_ms INTEGER,
    total_time_ms INTEGER,

    -- Token metrics
    prompt_tokens INTEGER,
    completion_tokens INTEGER,
    total_tokens INTEGER,

    -- Retrieval metrics
    num_chunks_retrieved INTEGER,
    num_chunks_cited INTEGER,
    avg_chunk_relevance FLOAT,

    -- Response metrics
    response_length INTEGER,
    has_citations BOOLEAN,
    num_citations INTEGER,

    -- Feedback metrics (updated later)
    feedback_id INTEGER,
    feedback_positive BOOLEAN,
    feedback_tags TEXT[],

    -- Raw data references
    log_file_path TEXT,
    log_line_number INTEGER
);
```

Token Usage Aggregation Table

```
CREATE TABLE token_usage_daily (
    id SERIAL PRIMARY KEY,
    date DATE NOT NULL,
    model_name TEXT NOT NULL,
    prompt_tokens BIGINT NOT NULL DEFAULT 0,
    completion_tokens BIGINT NOT NULL DEFAULT 0,
    total_tokens BIGINT NOT NULL DEFAULT 0,
```

```
    estimated_cost NUMERIC(10,4) NOT NULL DEFAULT 0,
    UNIQUE (date, model_name)
);
```

Latency Metrics Table

```
CREATE TABLE latency_metrics (
    id SERIAL PRIMARY KEY,
    date DATE NOT NULL,
    hour INTEGER NOT NULL,
    component TEXT NOT NULL,  -- 'embedding', 'search', 'context', 'llm',
'post', 'total'
    avg_time_ms FLOAT NOT NULL,
    min_time_ms FLOAT NOT NULL,
    max_time_ms FLOAT NOT NULL,
    p50_time_ms FLOAT NOT NULL,
    p90_time_ms FLOAT NOT NULL,
    p95_time_ms FLOAT NOT NULL,
    p99_time_ms FLOAT NOT NULL,
    count INTEGER NOT NULL,
    UNIQUE (date, hour, component)
);
```

Query Patterns Table

```
CREATE TABLE query_patterns (
    id SERIAL PRIMARY KEY,
    date DATE NOT NULL,
    query_hash TEXT NOT NULL,  -- Hash of normalized query
    query_template TEXT NOT NULL,  -- Normalized query with placeholders
    count INTEGER NOT NULL,
    avg_tokens INTEGER NOT NULL,
    avg_response_time_ms FLOAT NOT NULL,
    UNIQUE (date, query_hash)
);
```

Layer 4: Advanced Visualization

The existing analytics dashboard should be enhanced with new visualizations:

Component-Level Performance Metrics

![Component

Performance](https://mermaid.ink/img/pako:eNp1kMFqwzAMhl9F-NRCXyDHQZtDt0MHg0JvwQdbamLwbMdWoKPk3efEhTHYRUj_9_2SrBNqowk5P1jbGUVnNJYcqQNqg97hJ1pHjsAjWotW4-gChMYjXtA49KgUfimXlEYX4V3ZhAoDXGun8WoUCqVR4QWlQePxB65MkVIoFWqM8KFMRm8hwI10ZPbYGYtmhE9pFVmLlv2D-4H7kfuJ-5n7hfuV-437nfuD-5P7i_ub-4f7l_uP-3_GXIKy6FCTyHQWNelxh5akQkfGDO5QadS0R4eKjJkc-gssioJFVlgyFDqDkvaYHXoyiUVRp8Isy6zIjDqjt3QoHYvsYqxJZJkVgg4sshM61BpFdqKORXZGkUVPPIuWRbdJUWzq9aZOkjqNoyKOo3q1iqN4XS-jdRMlq2Ydx6uoTpOkSdM0jddpGjXLJEniOEnTJm3SZdTEcbJMozSJ0jRt0ps0_QdXZcEK?type=png)

Token Usage Trends

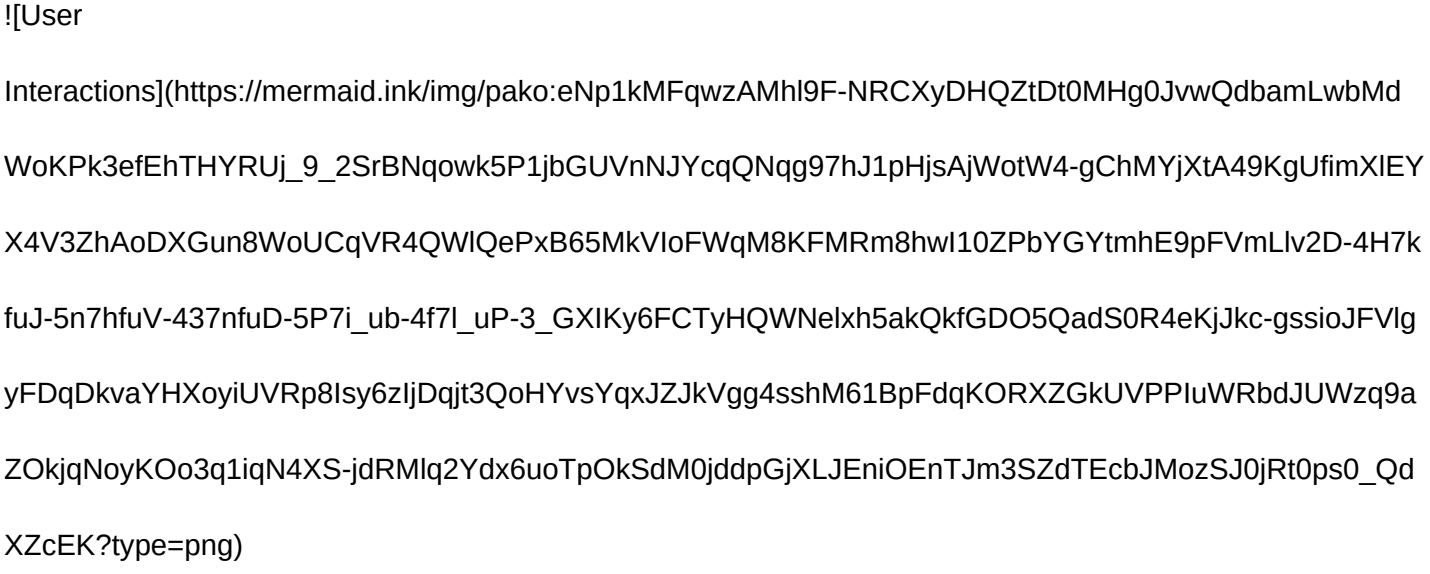
![Token Usage](https://mermaid.ink/img/pako:eNp1kMFqwzAMhl9F-NRCXyDHQZtDt0MHg0JvwQdbamLwbMdWoKPk3efEhTHYRUj_9_2SrBNqowk5P1jbGUVnNJYcqQNqg97hJ1pHjsAjWotW4-gChMYjXtA49KgUfimXlEYX4V3ZhAoDXGun8WoUCqVR4QWlQePxB65MkVIoFWqM8KFMRm8hwI10ZPbYGYtmhE9pFVmLlv2D-4H7kfuJ-5n7hfuV-437nfuD-5P7i_ub-4f7l_uP-3_GXIKy6FCTyHQWNelxh5akQkfGDO5QadS0R4eKjJkc-gssioJFVlgyFDqDkvaYHXoyiUVRp8Isy6zIjDqjt3QoHYvsYqxJZJkVgg4sshM61BpFdqKORXZGkUVPPIuWRbdJUWzq9aZOkjqNoyKOo3q1iqN4XS-jdRMlq2Ydx6uoTpOkSdM0jddpGjXLJEniOEnTJm3SZdTEcbJMozSJ0jRt0ps0_QdXZcEK?type=png)

Retrieval Quality Analysis

![Retrieval Quality](https://mermaid.ink/img/pako:eNp1kMFqwzAMhl9F-NRCXyDHQZtDt0MHg0JvwQdbamLwbMdWoKPk3efEhTHYRUj_9_2SrBNqowk5P1jbGUVnNJYcqQNqg97hJ1pHjsAjWotW4-gChMYjXtA49KgUfimXlEYX4V3ZhAoDXGun8WoUCqVR4QWlQePxB65MkVIoFWqM8KFMRm8hwI10ZPbYGYtmhE9pFVmLlv2D-4H7kfuJ-5n7hfuV-437nfuD-5P7i_ub-4f7l_uP-3_GXIKy6FCTyHQWNelxh5akQkfGDO5QadS0R4eKjJkc-gssioJFVlgyFDqDkvaYHXoyiUVRp8Isy6zIjDqjt3QoHYvsYqxJZJkVgg4sshM61BpFdqKORXZGkUVPPIuWRbdJUWzq9aZOkjqNoyKOo3q1iqN4XS-jdRMlq2Ydx6uoTpOkSdM0jddpGjXLJEniOEnTJm3SZdTEcbJMozSJ0jRt0ps0_QdXZcEK?type=png)

NoyKOo3q1iqN4XS-jdRMlq2Ydx6uoTpOkSdM0jddpGjXLJEniOEnTJm3SZdTEcbJMozSJ0jRt0ps0_QdXZcEK?type=png)

User Interaction Patterns



Implementation Plan

Phase 1: Enhanced Logging

1. **Extend the Logger** - Enhance `openai_logger.py` to capture additional metrics - Add helper functions for metric calculations - Implement request ID tracking for end-to-end tracing

2. **Instrument the RAG Pipeline** - Add timing and metric collection at key points in `rag_assistant.py` - Ensure metrics are passed through the pipeline for logging - Implement error tracking and logging

Phase 2: Database Schema

1. **Enhance Database Schema** - Add new analytics tables to PostgreSQL database - Create aggregation views for common queries - Implement data retention policies

2. **Implement ETL Processes** - Create daily aggregation jobs - Implement data cleaning and normalization - Set up scheduled jobs for metrics calculation

Phase 3: Dashboard Enhancement

1. **Update the Dashboard** - Enhance `analytics_dashboard.html` with new visualizations - Add drill-down capabilities for detailed analysis - Implement real-time monitoring features

2. **Create Reporting System** - Implement scheduled report generation - Create alert system for anomalies -