AI Focused Orchestration Flow- Revised

1. Objective

We are building a self-hosted AI-powered document Q&A platform with:

- ChromaDB as the primary vector database for scalable, persistent retrieval
- LangChain for context-aware QnA on PDFs
- OpenAI LLM for responses, with rate limiting & WebSockets for streaming
- Hybrid Precompute + On-Demand strategy for optimal performance
- CI/CD with GitHub Actions for modular code delivery

2. System Overview

The system follows microservices architecture with intelligent caching and background processing to ensure both immediate responsiveness and eventual consistency.

3. Component Breakdown

A. Frontend (Next.js)

- Provides an upload UI for PDFs with real-time progress indicators
- Provides a chat interface with WebSocket streaming for responses
- Handles multi-model insights with loading states:
  - "Summarize" → shows cached summary instantly or displays "Generating..." with fallback
  - "Sentiment Analysis" → precomputed results or on-demand processing
  - "Ask a Question" → hybrid retrieval with cached context enhancement

B. Backend (FastAPI)

Modular services:

- **Auth Service**: JWT-based user management with session tracking
- **Doc Service**: PDF ingestion + text extraction using PyMuPDF (fitz) or Unstructured.io
- **Embedding Worker**: Asynchronous embedding generation (Celery + Redis)
- **Precompute Worker**: Background jobs for summary/sentiment generation
- **RAG Service**: LangChain pipeline with hybrid retrieval strategy
- **LLM Service**: OpenAI API wrapper with:
  - Rate limiting (async queue + retry)
  - Streaming output over WebSocket
  - Fallback mechanisms for cache misses
- **Cache Service**: Redis-based caching for precomputed insights
- **Multi-Model Orchestrator**: Intelligent routing with cache-first strategy

C. Vector Database

- **ChromaDB**:
  - Persistent, self-hosted, production-ready
  - Supports metadata queries and filtering
  - Built-in embedding functions with custom model support
  - Horizontal scaling capabilities
  - Docker-friendly with persistent volumes
- **Data Storage**:
  - Document embeddings with chunk-level metadata
  - Document metadata (ID, filename, upload timestamp, processing status)
  - Precomputed insights cache keys

D. Document Processing

Primary: PyMuPDF (fitz)

- Superior text extraction accuracy
- Handles complex layouts, tables, and images
- Metadata extraction (author, creation date, etc.)
- Page-level processing for better chunk attribution

Alternative: Unstructured.io

- Advanced document understanding
- Better handling of complex document structures
- OCR capabilities for scanned documents
- Multiple output formats (text, JSON, HTML)

E. LangChain

- **Text Splitters**: Semantic chunking with overlap optimization
- **Retrievers**: Hybrid retrieval combining:
  - Vector similarity search from ChromaDB
  - Keyword-based search for exact matches
  - Metadata filtering for context relevance
- **Chains**: Enhanced prompt injection with cached context
- **Memory**: Multi-session context with Redis persistence
- **Cache Integration**: LangChain cache backends for repeated queries

F. OpenAI LLM (with Enhanced Resiliency)

- **Rate Limiter**: Token bucket algorithm with burst handling
- **Timeout Handler**: Progressive backoff with circuit breaker pattern
- **WebSocket Streaming**: Chunked response delivery
- **Model Fallback**: Graceful degradation (GPT-4 → GPT-3.5-turbo)

- **Cost Optimization**: Cached responses to reduce API calls

4. Hybrid Precompute + On-Demand Strategy

Document Upload Flow

1. **Immediate Processing**:
   - User uploads PDF
   - Extract text using PyMuPDF
   - Generate embeddings and store in ChromaDB
   - Return upload success to user
2. **Background Precompute Jobs** (Celery):
   - Priority Queue:
   - - High: Document Summary Generation
   - - Medium: Sentiment/Tone Analysis
   - - Low: Advanced Analytics (readability, topics)
3. **Cache Storage**:
   - Store precomputed results in Redis with TTL
   - Update document status in PostgreSQL
   - Create cache keys linked to document ID

Query Processing Flow

1. **Cache-First Strategy**:
   - Query Request → Check Redis Cache → If Hit: Return Instantly

     → If Miss: Trigger Fallback
2. **Fallback Mechanism**:
   - Quick on-demand LLM call with reduced context
   - Enqueue background job for full processing
   - Return partial results with "enhancing..." indicator
3. **Background Enhancement**:
   - Process full context in background
   - Update cache when complete
   - Notify frontend via WebSocket of enhanced results

5. Orchestration Flow

Document Ingestion (Hybrid)

```
1. Upload PDF → FastAPI receives file
2. Immediate Response:
   - Quick text extraction (first page preview)
   - Generate basic embeddings
   - Store in ChromaDB with "processing" status
   - Return success to user
3. Background Jobs (Celery):
   - Full document processing with PyMuPDF
   - Complete embedding generation
   - Summary generation → cache in Redis
   - Sentiment analysis → cache in Redis
   - Update status to "ready"
```

User Query (QnA)

1. User question → Frontend chat

2. Backend WebSocket Handler:

- Check cache for similar queries
- If cached: return immediately
- If not: hybrid retrieval flow

3. Hybrid Retrieval

- Query ChromaDB for relevant chunks
- Check for precomputed context enhancements
- Combine cached + real-time context

4. LangChain Processing:

- Build enhanced prompt with cached insights
- Stream response via WebSocket
- Cache response for future queries

Multi-Model Insights

- **Summarization**:
  - Check cache → return instantly if available
  - Fallback → quick summary from first few chunks
  - Background → full document summary generation
- **Sentiment Analysis**:
  - Precomputed → instant results
  - On-demand → quick sentiment on document excerpt

6. Local Orchestration

Services Configuration

```
services:
  frontend:
```

```
  - Next.js with real-time status updates
  - WebSocket client for streaming responses

backend:
  - FastAPI with async workers
  - WebSocket support for real-time updates

chromadb:
  - Persistent vector database
  - Mounted volumes for data persistence
  - Health checks and restart policies

redis:
  - Cache for precomputed insights
  - Celery broker for background jobs
  - Session storage for user contexts

celery-worker:
  - Multiple worker types:
    - embedding-worker (CPU intensive)
    - llm-worker (API calls)
    - precompute-worker (background analytics)

postgres:
  - Document metadata and user data
  - Query logs and analytics
  - LLM response auditing

monitoring:
  - Health check endpoints
  - Metrics collection
```

7. CI/CD Pipeline with GitHub Actions

1. Code Quality:

8. Best Practices

Performance Optimization

- **Lazy Loading**: Load document chunks on-demand
- **Connection Pooling**: Database and Redis connection management
- **Batch Processing**: Group similar operations
- **CDN Integration**: Serve static assets efficiently

Reliability

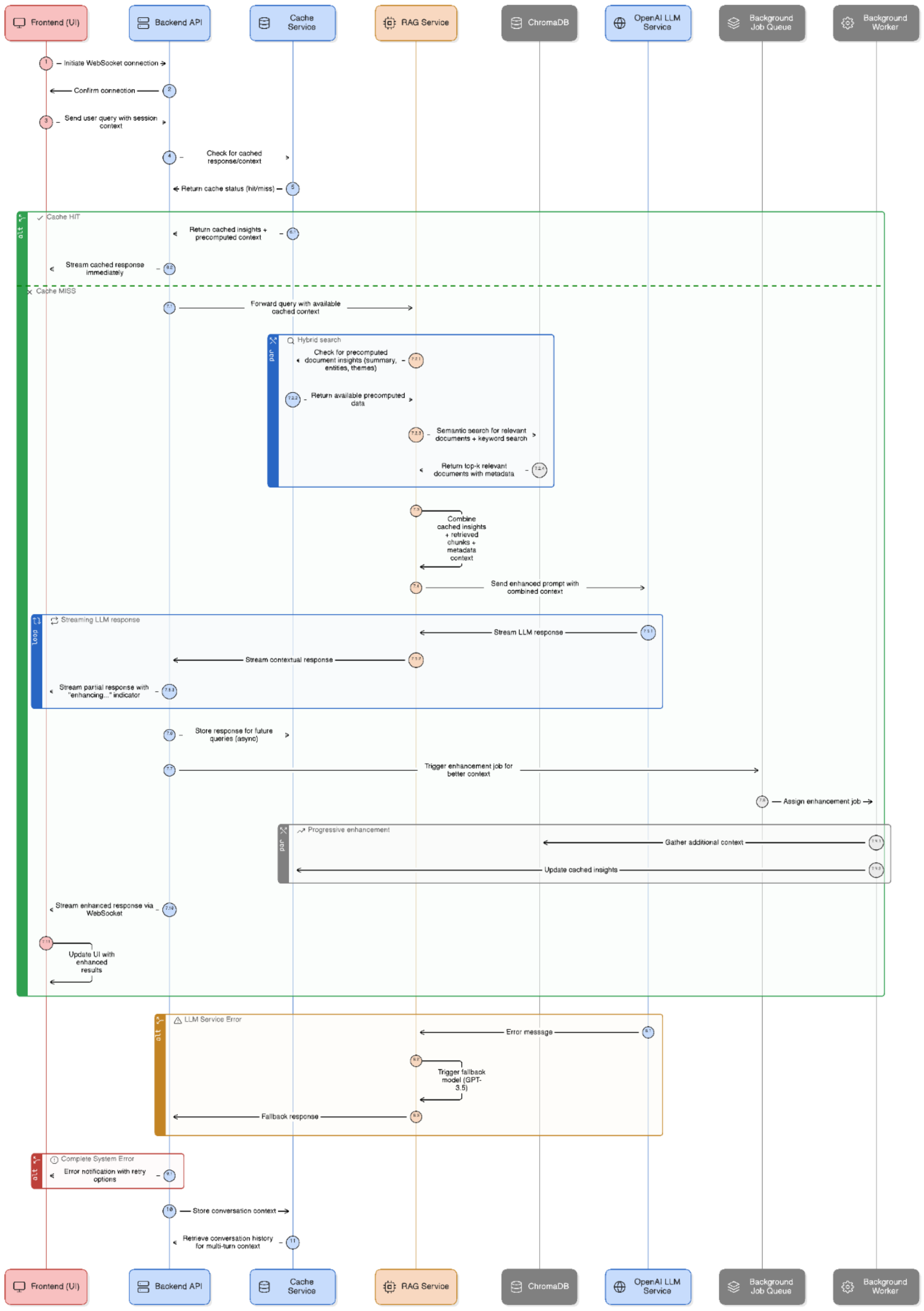
- **Circuit Breakers**: Prevent cascade failures
- **Health Checks**: Monitor service availability
- **Graceful Degradation**: Fallback strategies for each component
- **Data Backup**: Automated ChromaDB and PostgreSQL backups

Monitoring & Observability

- **Structured Logging**: JSON logs with correlation IDs
- **Metrics Collection**: Response times, cache hit rates, error rates
- **Distributed Tracing**: Track requests across services
- **Alerting**: Automated notifications for issues

Security

- **API Rate Limiting**: Per-user quotas
- **Input Validation**: Sanitize uploads and queries
- **Secrets Management**: Encrypted environment variables
- **RBAC**: Role-based access control



Frontend (UI)
Backend API
Cache Service
RAG Service
ChromaDB
OpenAI LLM Service
Background Job Queue
Background Worker
1 Initiate WebSocket connection
2 Confirm connection
3 Send user query with session context
4 Check for cached response/context
5 Return cache status (hit/miss)
alt
Cache HIT
Return cached insights + precomputed context
Stream cached response immediately
Cache MISS
Forward query with available cached context
par
Hybrid search
Check for precomputed document insights (summary, entities, themes)
Return available precomputed data
Semantic search for relevant documents + keyword search
Return top-k relevant documents with metadata
Combine cached insights + retrieved chunks + metadata context
Send enhanced prompt with combined context
loop
Streaming LLM response
Stream LLM response
Stream contextual response
Stream partial response with "enhancing..." indicator
Store response for future queries (async)
Trigger enhancement job for better context
Assign enhancement job
par
Progressive enhancement
Gather additional context
Update cached insights
Stream enhanced response via WebSocket
Update UI with enhanced results
alt
LLM Service Error
Error message
Trigger fallback model (GPT-3.5)
Fallback response
alt
Complete System Error
Error notification with retry options
10 Store conversation context
11 Retrieve conversation history for multi-turn context
Frontend (UI)
Backend API
Cache Service
RAG Service
ChromaDB
OpenAI LLM Service
Background Job Queue
Background Worker

eraser