6MChunks
System Architecture Documentation

6MChunks Team

January 6, 2026

Contents

Project Overview

Purpose

6MChunks is a real-time news intelligence platform that aggregates, processes, and analyzes news articles from multiple sources using AI-powered agents. The system provides intelligent news search, deep research capabilities, and automated news tracking through ambient agents.

Core Problems Solved

1. **Multi-Source News Aggregation**: Unified ingestion from GNews, NewsAPI, and NewsData.io with normalization
2. **Real-Time Processing**: Stream-based architecture using Kafka and Pathway for low-latency news processing
3. **Semantic Search**: Vector-based retrieval using embeddings for meaning-based article discovery
4. **Intelligent Research**: AI agents that autonomously research topics using multiple tools (database, RAG, web search)
5. **Automated Monitoring**: Background agents that track news based on natural language queries
6. **Market Intelligence**: Integration with market data for financial news correlation

High-Level System Goals

- **Real-Time**: Process and make searchable news articles within seconds of ingestion
- **Scalable**: Handle high-volume news streams with horizontal scaling capabilities
- **Intelligent**: Provide context-aware search and research beyond keyword matching
- **Reliable**: Fault-tolerant architecture with graceful degradation
- **Extensible**: Modular design allowing new data sources and AI capabilities

Key Differentiators

1. **Pathway-Powered Real-Time Processing**: Uses Pathway for streaming data processing and real-time vector search updates
2. **Multi-Tool Agent System**: Agents orchestrate database queries, semantic search, and web search for comprehensive research
3. **Unified Data Model**: Normalizes disparate news source schemas into a consistent format
4. **Embedding Caching**: Intelligent caching prevents redundant embedding generation
5. **Model Rotation**: Automatic failover between multiple LLM providers/models for reliability

System Architecture

High-Level Architecture

The system follows a layered architecture with the following components:

- **Data Sources**: GNews API, NewsAPI, NewsData.io, Market API (Massive)
- **FastAPI Gateway**: Unified REST API (Port 8000) with multiple service routers
- **PostgreSQL Database**: Stores articles in source-specific tables (gnews, newsapi, newsdata) and user data
- **Database Watcher**: Polling service that monitors tables and publishes to Kafka
- **Kafka Message Broker**: Topic `news_articles_topic` for article streaming
- **Embedder Service**: Pathway consumer that generates embeddings using sentence-transformers models
- **Document Store Service**: Pathway server with KNNIndex for vector search
- **Agent Service**: LangGraph-based deep research agent with tool orchestration
- **Frontend**: Next.js React-based UI with chatbot interface

Logical Layers

Data Ingestion Layer

- **News Service**: Fetches articles from external APIs (GNews, NewsAPI, NewsData.io)
- **Market Service**: Fetches US stock market data from Massive API
- **Storage**: Articles stored in PostgreSQL with source-specific tables

World Data Model Layer

- **Normalization**: Database Watcher normalizes articles from different sources into unified schema
- **Metadata Enrichment**: Adds source type, language, category, timestamps
- **Deduplication**: URL-based and content-based deduplication

Embedding & Vectorization Layer

- **Embedder Service**: Generates 384-dimensional embeddings using sentence-transformers
- **Caching**: Disk-based cache prevents redundant embedding generation
- **Stream Processing**: Pathway processes articles in real-time from Kafka

RAG (Pathway-Powered) Layer

- **Document Store**: Pathway KNNIndex for efficient vector similarity search
- **Real-Time Updates**: New embeddings automatically added to search index
- **Query Embedding**: On-the-fly query embedding for semantic search

Agentic AI Layer

- **Deep Research Agent**: LangGraph-based agent with multi-tool orchestration
- **Tool System**: Database queries, RAG search, web search (Tavily)
- **Reasoning**: Iterative research with tool calls and synthesis

APIs / Interfaces Layer

- **FastAPI Gateway**: Unified REST API for all services
- **Authentication**: JWT-based auth with user management
- **Streaming**: Server-Sent Events (SSE) for real-time agent responses

Data Flow

Ingestion Flow

```
External APIs → News Service → PostgreSQL → Database Watcher → Kafka
```

Embedding Flow

```
Kafka → Embedder Service (Pathway) → Embedding Generation → Disk Cache
```

Search Flow

```
User Query → Agent → Tools (DB/RAG/Web) → Document Store → Results
```

Real-Time Update Flow

```
New Article → Database → Watcher → Kafka →
Embedder → Document Store (auto-update)
```

Control Flow

- **Request-Driven**: API endpoints, agent queries, user authentication
- **Event-Driven**: Database watcher → Kafka → Embedder pipeline
- **Scheduled**: Ambient agents execute SQL queries hourly
- **Streaming**: Agent responses via SSE, Pathway stream processing

Service-by-Service Breakdown

News Service (`news_service/`)

Responsibility: Fetch and store news articles from multiple sources.
Inputs:

- API keys for GNews, NewsAPI, NewsData.io
- Keywords for topic-based fetching
- Query parameters (language, date range, etc.)

Outputs:

- Articles stored in PostgreSQL (`gnews`, `newsapi`, `newsdata` tables)
- Statistics (stored count, updated count, errors)

Internal Workflow:

1. Receive fetch request with keyword
2. Call external API (GNews/NewsAPI/NewsData)
3. Parse and normalize article data
4. Check for existing articles (by URL)
5. Insert new or update existing articles
6. Tag articles with keyword for later retrieval

Dependencies:

- PostgreSQL database
- External news APIs
- SQLAlchemy ORM

Failure Handling:

- API rate limit errors: Logged, request fails gracefully
- Database errors: Transaction rollback, error count incremented
- Network errors: Retry logic, timeout handling

Scalability Considerations:

- Stateless service, can scale horizontally
- Database connection pooling
- Rate limiting per API provider

Database Watcher (`database_watcher/`)

Responsibility: Monitor PostgreSQL tables for new articles and publish to Kafka.

Inputs:

- Polling interval (default: 5 seconds)
- Database connection
- Kafka broker connection

Outputs:

- Normalized article messages to Kafka topic `news_articles_topic`

Internal Workflow:

1. Poll PostgreSQL tables (`gnews`, `newsapi`, `newsdata`)

2. Query articles ordered by `created_at`
3. Normalize articles to unified schema:
    - `article_id`: Prefixed with source type (e.g., `gnews_123`)
    - `source_type`: "gnews", "newsapi", or "newsdata"
    - Standardized fields: title, description, content, url, author, published_at, source_name, category, and language
4. Check if embedding already exists (by cache key)
5. If embedding missing, send to Kafka
6. Track processed article IDs to avoid duplicates

Dependencies:

- PostgreSQL
- Kafka
- Embedding directory (to check for existing embeddings)

Failure Handling:

- Database connection errors: Retry with exponential backoff
- Kafka publish failures: Logged, article skipped (will be retried on next poll)
- Embedding check failures: Assume missing, send to Kafka

Scalability Considerations:

- Single instance recommended (polling-based, not distributed)
- Polling interval configurable for throughput vs. latency tradeoff
- Memory-efficient: Processes articles in batches

Embedder Service (`embedder_service/`)

Responsibility: Generate embeddings for news articles using sentence-transformers.

Inputs:

- Kafka messages with normalized article data
- Embedding model name (default: `sentence-transformers/all-MiniLM-L6-v2`)

Outputs:

- Pickle files in `.embeddings/` directory containing:
    - Embedding vector (384 dimensions)
    - Article metadata
    - Cache key
    - Timestamps

Internal Workflow:

1. Consume from Kafka using Pathway's Kafka reader
2. For each article:
    - Generate cache key from title + content hash
    - Check if embedding exists in cache
    - If cache hit: Return cached embedding, update access time
    - If cache miss: Generate embedding using sentence-transformers
    - Save embedding to disk as pickle file
3. Pathway processes stream continuously

Dependencies:

- Kafka
- Sentence-transformers library
- Disk storage for embeddings
- Pathway framework

Failure Handling:

- Model loading errors: Service fails to start (critical dependency)
- Embedding generation errors: Logged, article skipped
- Disk write errors: Logged, embedding not cached (will regenerate)

Scalability Considerations:

- Stateless processing, can scale horizontally
- Cache reduces redundant computation
- Model loaded once per instance (memory-intensive)
- Pathway handles stream processing efficiently

Document Store Service (`document_store_service/`)

Responsibility: Provide semantic search over embedded articles using Pathway's KNNIndex.

Inputs:

- Embedding pickle files from `.embeddings/` directory
- Search queries (text)

Outputs:

- Ranked list of articles with similarity scores
- REST API responses (JSON)

Internal Workflow:

1. **Startup**:
    - Load all embedding pickle files from disk

- Build Pathway KNNIndex with embeddings
- Load query embedder model (sentence-transformers)

2. **Search Request**:
   - Receive query text via REST API
   - Embed query using query embedder
   - Query KNNIndex for k nearest neighbors
   - Format results with metadata
   - Return JSON response
3. **Real-Time Updates**:
   - Pathway automatically updates index when new embeddings detected
   - No manual reload required

Dependencies:

- Embedding directory (shared volume with embedder service)
- Pathway framework
- Sentence-transformers for query embedding

Failure Handling:

- Index build failures: Service fails to start
- Query embedding errors: Returns empty results
- File I/O errors: Logged, service continues with existing index

Scalability Considerations:

- Single instance recommended (index in memory)
- Index size limited by available RAM
- Query latency: ∼100–500ms depending on index size
- Pathway handles concurrent queries efficiently

Agent Service (`agents/`)

Responsibility: Provide AI-powered deep research using multi-tool orchestration.

Inputs:

- User query (natural language)
- Optional document references
- Max iterations (default: 10)

Outputs:

- Streaming response (SSE) with research findings
- Tool call results

- Final synthesized answer
- Source citations

Internal Workflow:

1. Initialize LangGraph state with user query
2. Agent receives system prompt with tool definitions
3. Agent decides to call tools (query_db, query_rag, query_search)
4. Tools execute and return results
5. Agent analyzes results and decides next action:
    - Call more tools if needed
    - Synthesize final answer if sufficient information
6. Stream intermediate and final responses via SSE
7. Track iteration count to prevent infinite loops

Tool Definitions:

- **query_db**: Execute SQL queries on news database (keyword-based search)
- **query_rag**: Semantic search via document store (meaning-based search)
- **query_search**: Web search via Tavily API (latest information)

Dependencies:

- Arli AI LLM API (with model rotation)
- PostgreSQL database
- Document store service
- Tavily API (web search)

Failure Handling:

- LLM API errors: Rotate to next model, retry
- Tool execution errors: Logged, agent continues with available tools
- Timeout errors: Return partial results
- Max iterations reached: Return current findings

Scalability Considerations:

- Stateless agent instances
- Model rotation handles rate limits
- Tool calls are async for better throughput
- Streaming reduces perceived latency

Reranker Service (`reranker/`)

Responsibility: Re-rank search results using Qwen3-Reranker model (hosted on Modal).

Inputs:

- Query text
- List of documents from document store
- Top N parameter

Outputs:

- Re-ranked documents with relevance scores

Internal Workflow:

1. Receive search query and k documents
2. Fetch k documents from document store (if not provided)
3. Send query + documents to Modal reranker endpoint
4. Receive re-ranked results with scores
5. Return top N results

Dependencies:

- Modal reranker endpoint (external service)
- Document store service

Failure Handling:

- Modal endpoint unavailable: Returns error, falls back to unranked results
- Timeout errors: Returns partial results

Scalability Considerations:

- Stateless service
- Modal handles model inference scaling
- Can be called independently or as part of agent workflow

Market Service (`market/`)

Responsibility: Fetch and store US stock market data from Massive API.

Inputs:

- Date for market snapshot
- Massive API key

Outputs:

- Market snapshots stored in PostgreSQL (`market_snapshots` table)
- Daily aggregated stock data (open, high, low, close, volume)

Internal Workflow:

1. Receive date parameter
2. Call Massive API for daily market snapshot
3. Parse response with stock ticker data
4. Upsert into database (update if exists, insert if new)
5. Return statistics (stored, updated counts)

Dependencies:

- Massive API (external)
- PostgreSQL database

Failure Handling:

- API errors: Logged, request fails gracefully
- Database errors: Transaction rollback

Scalability Considerations:

- Stateless service
- Daily snapshots (low frequency)
- Batch upsert operations

Authentication Service (`auth/`)

Responsibility: User authentication, registration, and JWT token management.

Inputs:

- User credentials (email, password)
- Registration data (full_name, email, password)

Outputs:

- JWT tokens for authenticated users
- User profile data
- Onboarding status

Internal Workflow:

1. **Registration**:
    - Validate email format
    - Hash password (SHA-256 with salt)
    - Create user record in PostgreSQL
    - Return user data (no password)
2. **Login**:
    - Verify credentials

- Generate JWT token with user claims
- Return token and user profile

3. **Token Validation**:
   - Decode JWT
   - Verify signature and expiration
   - Extract user ID and claims

Dependencies:

- PostgreSQL database (`users` table)
- JWT library (PyJWT)

Failure Handling:

- Invalid credentials: Return 401 Unauthorized
- Expired tokens: Return 401 with error message
- Duplicate email: Return 400 Bad Request

Scalability Considerations:

- Stateless authentication (JWT)
- Password hashing is CPU-intensive but acceptable for registration rate
- Database indexes on email for fast lookups

Chatroom Service (`chatroom/`)

Responsibility: Manage user chat threads and messages.

Inputs:

- User ID
- Thread/message data
- Public/private thread flags

Outputs:

- Chat threads with messages
- Public thread listings

Internal Workflow:

1. Create chat thread for user
2. Store messages with thread association
3. Support public/private threads
4. Retrieve thread history

Dependencies:

- PostgreSQL database (`chatrooms`, `messages` tables)

- Auth service (for user validation)

Failure Handling:

- Database errors: Transaction rollback
- Invalid user: Return 404

Scalability Considerations:

- Stateless service
- Indexed queries on user_id and thread_id
- Message pagination for large threads

Ambient Agent Service (`ambient_agent/`)

Responsibility: Automated background agents that track news based on natural language queries.

Inputs:

- Natural language query (e.g., "Send me news related to Donald Trump")
- Agent name
- User ID

Outputs:

- SQL query generated from natural language
- AI-generated description
- Execution results stored in database
- Hourly execution of queries

Internal Workflow:

1. **Agent Creation**:
   - User provides natural language query
   - LLM converts query to SQL (with schema context)
   - LLM generates description of the query
   - SQL query validated and post-processed (fixes UNION ALL syntax)
   - Agent stored in database
2. **Scheduled Execution** (hourly):
   - Scheduler retrieves all active agents
   - For each agent:
     - Execute SQL query on news database
     - Store results in `agent_execution_results` table
     - Update agent's `last_executed_at` timestamp
     - Handle errors gracefully (store error message)

3. **Execution History**:
    - Users can retrieve execution history
    - View results, error messages, execution times

Dependencies:

- PostgreSQL database
- Arli AI LLM (for SQL generation and description)
- APScheduler for background tasks

Failure Handling:

- SQL syntax errors: Post-processing fixes common issues, errors stored in execution results
- Database transaction errors: Rollback before storing error results
- LLM failures: Model rotation attempts, fallback to error state

Scalability Considerations:

- Background scheduler runs in single instance
- SQL queries can be optimized by users
- Execution results stored with JSONB for flexible schema
- Hourly execution limits database load

Provider Service (`provider/`)

Responsibility: Provide AI assistant capabilities with RAG search integration.

Inputs:

- User query
- Optional document context
- RAG search parameters

Outputs:

- AI-generated responses
- RAG search results
- Reasoning content (if model supports it)

Internal Workflow:

1. Receive user query
2. Optionally perform RAG search for context
3. Format documents as context for LLM
4. Call Arli AI with context
5. Return response with reasoning (if available)

Dependencies:

- Arli AI LLM API (with model rotation)
- Document store service (for RAG)
- Redis cache (optional)

Failure Handling:

- LLM API errors: Model rotation, retry logic
- RAG search failures: Continue without context
- Rate limit errors: Rotate to next model

Scalability Considerations:

- Stateless service
- Model rotation handles rate limits
- Caching reduces redundant RAG calls

Simulator Engine (`simulator_engine/`)

Responsibility: Generate simulated news articles for testing and development.

Inputs:

- Production interval (default: 5 seconds)
- Kafka topic configuration

Outputs:

- Simulated news articles published to Kafka

Internal Workflow:

1. Generate random news articles with realistic structure
2. Publish to Kafka topic at configured interval
3. Used for testing pipeline without external API dependencies

Dependencies:

- Kafka broker

Failure Handling:

- Kafka connection errors: Retry with backoff

Scalability Considerations:

- Single instance sufficient for testing
- Configurable production rate

Input Engine (`input_engine/`)

Responsibility: Consume news from Kafka using Pathway and forward to embedder service.

Inputs:

- Kafka messages from `news_topic`

Outputs:

- Processed news to `processed_news_topic` (legacy)
- CSV output for monitoring

Internal Workflow:

1. Consume from Kafka using Pathway Kafka reader
2. Process and format articles
3. Forward to embedder service topic
4. Write to CSV for monitoring

Dependencies:

- Kafka
- Pathway framework

Note: This service is part of the legacy pipeline. The current architecture uses Database Watcher → Kafka → Embedder Service directly.

World Data Model

Data Schema Philosophy

The system uses a **multi-source, unified schema** approach:

1. **Source-Specific Tables**: Each news source (GNews, NewsAPI, NewsData.io) has its own PostgreSQL table to preserve source-specific fields
2. **Unified Normalization**: Database Watcher normalizes articles into a common schema for downstream processing
3. **Keyword Tagging**: All articles are tagged with keywords during ingestion for efficient retrieval
4. **Metadata Preservation**: Source-specific metadata (sentiment, AI tags, etc.) is preserved in source tables

Source Tables

GNews Table (`gnews`)

- **Primary Key**: `id` (VARCHAR) – Article ID from GNews API
- **Core Fields**: title, description, content, url, image, published_at
- **Source Metadata**: source_id, source_name, source_url, source_country
- **Language**: `lang` (default: "en")
- **Keyword**: Tagged keyword for categorization
- **Timestamps**: created_at, updated_at

NewsAPI Table (`newsapi`)

- **Primary Key**: `url` (VARCHAR) – URL is unique identifier
- **Core Fields**: title, description, content, author, url_to_image, published_at
- **Source Metadata**: source_id, source_name
- **Keyword**: Tagged keyword for categorization
- **Timestamps**: created_at, updated_at

NewsData Table (`newsdata`)

- **Primary Key**: `article_id` (VARCHAR) – Article ID from NewsData.io
- **Core Fields**: title, link (URL), description, content, pub_date
- **Rich Metadata**:
  - keywords (comma-separated), creator, category, language
  - sentiment, sentiment_stats (JSON)
  - ai_tag, ai_region, ai_org, ai_summary
  - country, source_priority
- **Media**: video_url, image_url
- **Keyword**: Tagged keyword for categorization
- **Timestamps**: created_at, updated_at

Unified Article Schema

The Database Watcher normalizes all sources into this unified format:

Listing 1: Unified Article Schema

```
{
    "article_id": "gnews_123" | "newsapi_abc" | "newsdata_xyz",
    "source_type": "gnews" | "newsapi" | "newsdata",
    "title": str,
    "description": str,
    "content": str,
    "url": str,
    "author": str,
    "published_at": str (ISO format),
    "source_name": str,
    "category": str,
    "language": str (default: "en")
}
```

Multilingual Handling

- **Language Detection**: Articles tagged with `lang` or `language` field
- **Default**: English ("en") if not specified
- **Storage**: Language preserved in source tables and unified schema
- **Search**: Language-aware search supported via database queries

Normalization Strategy

1. **Article ID Generation**:
   - GNews: `gnews_{id}`
   - NewsAPI: `newsapi_{md5_hash(url)}`
   - NewsData: `newsdata_{article_id}`
2. **Field Mapping**:
   - `link` (NewsData) → `url`
   - `pub_date` (NewsData) → `published_at`
   - `creator` (NewsData) → `author`
   - `url_to_image` (NewsAPI) → preserved in source table
3. **Content Combination**: For embedding generation, combines `title + description + content`

Metadata Strategy

- **Region**: Stored in `source_country` (GNews) or `country` (NewsData)
- **Time**: `published_at` standardized to UTC
- **Source**: `source_name` normalized across all sources
- **Keywords**: Tagged during ingestion from predefined keyword list
- **AI-Enriched**: NewsData provides sentiment, AI tags, summaries

How the Model Enables Downstream Intelligence

1. **Keyword-Based Retrieval**: Fast SQL queries using keyword column
2. **Semantic Search**: Unified schema enables consistent embedding generation
3. **Source Attribution**: Preserves source information for citation
4. **Temporal Analysis**: Standardized timestamps enable time-based queries
5. **Multi-Source Aggregation**: UNION ALL queries across tables for comprehensive coverage

Real-Time Embedding Pipeline

Trigger Mechanisms

The embedding pipeline is triggered by:

1. **Database Polling**: Database Watcher polls PostgreSQL tables every 5 seconds (configurable)
2. **New Article Detection**: Compares `created_at` timestamps to find new articles
3. **Embedding Check**: Verifies if embedding already exists (by cache key)
4. **Kafka Publishing**: Sends articles to Kafka only if embedding is missing

Streaming vs Batch Behavior

- **Streaming**: Pathway processes Kafka messages in real-time as they arrive
- **Batch Processing**: Embedder service processes articles one-by-one from Kafka stream
- **No Batching**: Each article processed independently for low latency

Embedding Lifecycle

1. **Article Ingestion**: News service stores article in PostgreSQL
2. **Detection**: Database Watcher detects new article (polling)
3. **Normalization**: Article normalized to unified schema
4. **Cache Check**: Generate cache key, check if embedding exists
5. **Kafka Publish**: If missing, publish to `news_articles_topic`
6. **Embedding Generation**: Embedder service consumes from Kafka
7. **Model Encoding**: Sentence-transformer model generates 384-dim vector
8. **Cache Storage**: Save embedding as pickle file in `.embeddings/` directory
9. **Index Update**: Document Store automatically picks up new embedding via Pathway

Cache Key Generation

Cache key is generated from article content hash:

Listing 2: Cache Key Generation

```
cache_key = f"{title}_{hash(title + description + content)}"
```

This ensures:

- Same article content = same embedding (cache hit)
- Different articles = different embeddings (cache miss)
- Prevents redundant computation

Vector Storage Strategy

- **Format**: Pickle files (`.pkl`) containing:
  - Embedding vector (numpy array, 384 dimensions)
  - Article metadata (title, content, timestamps, etc.)
  - Cache key
  - Model information
- **Location**: Shared volume (`.embeddings/`) accessible by Embedder and Document Store
- **Naming**: Files named by cache key for fast lookup
- **Indexing**: Pathway KNNIndex loads all embeddings into memory for fast search

Latency & Consistency Guarantees

End-to-End Latency:

- Polling interval: 5 seconds (configurable)
- Embedding generation: ∼100–500ms per article
- Index update: Real-time via Pathway
- **Total**: ∼5–10 seconds from article storage to searchable

Consistency:

- At-least-once delivery (Kafka)
- Idempotent embedding generation (cache prevents duplicates)
- Eventual consistency for index updates (Pathway handles this)

Retrieval-Augmented Generation (RAG)

Why RAG is Used

RAG enables the system to:

1. **Ground Responses**: Provide answers based on actual news articles, not just model knowledge
2. **Real-Time Information**: Access latest news that may not be in model training data
3. **Source Attribution**: Cite specific articles for transparency
4. **Reduce Hallucination**: Constrain model to retrieved context

How Embeddings are Queried

1. **Query Embedding**: User query is embedded using same sentence-transformer model
2. **Vector Similarity**: Pathway KNNIndex finds k nearest neighbors using cosine similarity
3. **Ranking**: Results ranked by similarity score (higher = more relevant)
4. **Metadata Return**: Returns article text, metadata, and similarity scores

How Pathway Enables Real-Time Updates

- **Incremental Index**: Pathway KNNIndex supports incremental updates
- **Automatic Detection**: New embeddings in `.embeddings/` directory are automatically detected
- **Stream Processing**: Pathway processes new embeddings as they arrive
- **No Manual Reload**: Index updates without service restart

Context Assembly Logic

1. **Retrieval**: RAG tool fetches top k articles (default: 10)
2. **Formatting**: Articles formatted as context strings:

```
--- Document 1 ---
Title: ...
Source: ...
Published: ...
Description: ...
Content: ...
URL: ...
```

3. **Context Injection**: Formatted context added to LLM system prompt
4. **Token Limits**: Context truncated if exceeds model limits

Response Synthesis Pipeline

1. **Agent Receives Query**: User query enters agent system
2. **Tool Call**: Agent calls `query_rag` tool with query
3. **RAG Search**: Document store performs semantic search
4. **Context Assembly**: Top k results formatted as context
5. **LLM Reasoning**: Agent uses context + query to generate response
6. **Source Extraction**: URLs and titles extracted for citation
7. **Streaming Response**: Response streamed to user via SSE

Reranking (Optional)

- **Purpose**: Improve relevance by re-ranking RAG results
- **Model**: Qwen3-Reranker (hosted on Modal)
- **Process**:
  1. Fetch k documents from RAG (e.g., k=20)
  2. Send to reranker with query
  3. Receive re-ranked top N (e.g., top 5)
- **Benefit**: Better precision at the cost of additional latency

Agentic AI Layer

Role of AI Agents

The Deep Research Agent serves as an **autonomous research assistant** that:

- Understands natural language queries

- Orchestrates multiple tools to gather information
- Synthesizes findings into coherent responses
- Provides source citations for transparency

How Agents Use RAG

1. **Primary Tool**: RAG search is a core tool in agent's toolkit
2. **Workflow Integration**: Agent calls RAG after database search for semantic coverage
3. **Context Enrichment**: RAG results provide semantic context beyond keywords
4. **Multi-Source**: Combines RAG results with database and web search results

Task Decomposition

The agent follows a structured workflow:

1. **Query Analysis**: Understands user intent and topic
2. **Tool Selection**: Chooses appropriate tools (query_db, query_rag, query_search)
3. **Sequential Execution**: Executes tools in recommended order:
   - Step 1: query_db (keyword-based, fast)
   - Step 2: query_rag (semantic, comprehensive)
   - Step 3: query_search (web, latest info)
4. **Result Analysis**: Evaluates tool results
5. **Iteration Decision**: Decides if more tools needed or ready to synthesize
6. **Synthesis**: Combines all findings into final answer

Reasoning Flow

1. **Initial State**: Agent receives user query and system prompt
2. **Tool Call Decision**: LLM decides which tool to call (or if ready to answer)
3. **Tool Execution**: Tool executes and returns results
4. **State Update**: Results added to agent state
5. **Reasoning Loop**: Agent analyzes results, decides next action
6. **Termination**: Max iterations reached OR agent decides sufficient information
7. **Final Response**: Agent synthesizes all findings

Multi-Agent Coordination

Currently **single-agent architecture**:

- One Deep Research Agent per query
- No inter-agent communication
- Each agent instance is independent

Future: Could support:

- Specialized agents (e.g., financial news agent, tech news agent)
- Agent orchestration layer
- Collaborative agent workflows

Tool System

Available Tools:

1. **query_db**: SQL queries on news database
   - Keyword-based filtering
   - Fast, structured queries
   - Returns article metadata
2. **query_rag**: Semantic search via document store
   - Meaning-based retrieval
   - Finds related articles by concept
   - Returns article text and metadata
3. **query_search**: Web search via Tavily API
   - Latest information from web
   - Fills gaps in database coverage
   - Returns web search results

Tool Execution:

- Tools are async functions
- Results formatted for LLM consumption
- Errors handled gracefully (logged, agent continues)

Query Lifecycle (End-to-End)

Step-by-Step Walkthrough

Step 1: User Query Enters System

- **Frontend**: User types query in chatbot interface
- **API Request**: POST to `/agents/deep-research` endpoint
- **Authentication**: JWT token validated
- **Request Body**: `{ "query": "...", "max_iterations": 10 }`

Step 2: Agent Invocation

- **Agent Initialization**: DeepResearchAgent created with LangGraph
- **State Initialization**: AgentState populated with:
  - `user_query`: User's question
  - `messages`: Initial system prompt + user message
  - `documents`: Empty (unless provided)
  - `tool_calls_made`: Empty list
  - `sources`: Empty list
  - `iteration_count`: 0

Step 3: First Tool Call (query_db)

- **Agent Reasoning**: LLM analyzes query, identifies keywords
- **Tool Selection**: Agent decides to call `query_db` first
- **SQL Generation**: Agent constructs SQL query:

Listing 3: Example SQL Query

```sql
SELECT id, title, description, url, published_at, source_name
FROM gnews
WHERE keyword IN ('Technology', 'Artificial Intelligence')
ORDER BY published_at DESC LIMIT 15
```

- **Tool Execution**: DatabaseTool executes SQL
- **Results**: Returns list of articles
- **State Update**: Results added to agent state, sources extracted

Step 4: Second Tool Call (query_rag)

- **Agent Reasoning**: LLM decides to use RAG for semantic search
- **Tool Selection**: Agent calls `query_rag` tool
- **RAG Search**:
  - Query embedded using sentence-transformer
  - Pathway KNNIndex finds top 10 similar articles
  - Results ranked by similarity score
- **State Update**: RAG results added to agent state

Step 5: Optional Third Tool Call (query_search)

- **Condition**: If agent needs latest information or gaps exist
- **Tool Selection**: Agent calls `query_search` (Tavily API)
- **Web Search**: Searches web for latest information
- **State Update**: Web results added to agent state

Step 6: Context Assembly

- **All Results Combined**: Database + RAG + Web results
- **Formatting**: Results formatted as context strings
- **Context Injection**: Added to LLM system prompt

Step 7: Model Reasoning

- **LLM Call**: Arli AI LLM called with:
  - System prompt (with context)
  - User query
  - Tool results
- **Model Rotation**: If rate limit/error, rotate to next model
- **Reasoning**: LLM analyzes all information
- **Decision**: LLM decides:
  - Call more tools? (if iteration_count < max_iterations)
  - Synthesize final answer? (if sufficient information)

Step 8: Final Response Generation

- **Synthesis**: LLM generates comprehensive answer
- **Source Extraction**: URLs and titles extracted from tool results
- **Formatting**: Response formatted for user consumption
- **Streaming**: Response streamed via SSE to frontend

Step 9: Response Delivery

- **SSE Events**: Multiple events sent:
  - `tool_call`: Tool execution updates
  - `content`: Partial response chunks
  - `sources`: Source citations
  - `end`: Final event
- **Frontend Rendering**: UI updates in real-time
- **User Receives**: Complete answer with sources

Latency Breakdown

- **Agent Initialization**: ∼50ms
- **Tool Call (query_db)**: ∼100–300ms
- **Tool Call (query_rag)**: ∼200–500ms
- **Tool Call (query_search)**: ∼500–2000ms (external API)

- **LLM Reasoning**: ∼1000–3000ms (depends on model)
- **Streaming**: Real-time as chunks generated
- **Total**: ∼2–6 seconds for typical query

Error Handling

- **Tool Failures**: Logged, agent continues with available tools
- **LLM Errors**: Model rotation, retry with next model
- **Timeouts**: Return partial results
- **Max Iterations**: Stop and return current findings

Operational Considerations

Observability

Logging

- **Format**: Structured logging with timestamps, service names, log levels
- **Levels**: INFO (operations), WARNING (recoverable errors), ERROR (failures)
- **Key Events**:
  - Article ingestion (stored/updated counts)
  - Embedding generation (cache hits/misses)
  - Agent tool calls
  - API requests/responses
  - Error conditions

Metrics (Recommended)

- **Article Throughput**: Articles processed per minute
- **Embedding Generation Rate**: Embeddings generated per minute
- **Cache Hit Rate**: Percentage of cache hits vs misses
- **API Latency**: P50, P95, P99 latencies for endpoints
- **Tool Execution Times**: Average time per tool call
- **Error Rates**: Error percentage per service

Traces (Future)

- **Distributed Tracing**: Track requests across services
- **Tool Call Traces**: Full agent execution traces
- **Embedding Pipeline Traces**: End-to-end article processing

Error Handling

Service-Level Error Handling

- **Database Errors**: Transaction rollback, error logging, graceful degradation
- **API Errors**: Retry logic, model rotation, timeout handling
- **Kafka Errors**: Retry with backoff, message acknowledgment
- **LLM Errors**: Model rotation, fallback to alternative models

User-Facing Error Handling

- **API Responses**: HTTP status codes (400, 401, 404, 500, 503)
- **Error Messages**: User-friendly error messages (no internal details)
- **Partial Results**: Return partial results on partial failures
- **Timeout Handling**: Return current findings if timeout occurs

Scaling Strategy

Horizontal Scaling

- **Stateless Services**: News Service, Agent Service, Provider Service can scale horizontally
- **Stateful Services**: Document Store (single instance, index in memory)
- **Database**: PostgreSQL with read replicas for query scaling
- **Kafka**: Partition-based scaling for consumers

Vertical Scaling

- **Document Store**: More RAM for larger index
- **Embedder Service**: More CPU for faster embedding generation
- **Database**: More CPU/RAM for query performance

Bottlenecks

- **Document Store**: Single instance, memory-limited index size
- **Embedding Generation**: CPU-intensive, model loading time
- **Database**: Write contention during high ingestion
- **LLM API**: Rate limits from external providers

Data Freshness Guarantees

- **Article Ingestion**: Real-time (as articles fetched from APIs)
- **Embedding Generation**: $\sim$5–10 seconds after article storage
- **Search Availability**: Real-time (Pathway auto-updates index)
- **Agent Queries**: Always query latest data (no caching of search results)

Cost Considerations

Infrastructure Costs

- **PostgreSQL**: Database hosting costs
- **Kafka**: Message broker hosting
- **Redis**: Cache hosting (optional)
- **Compute**: Service hosting (CPU/memory)

External API Costs

- **News APIs**: GNews, NewsAPI, NewsData.io subscription costs
- **LLM APIs**: Arli AI API costs (per token)
- **Web Search**: Tavily API costs (per search)
- **Reranker**: Modal hosting costs

Optimization Strategies

- **Embedding Cache**: Reduces redundant computation
- **Model Rotation**: Handles rate limits, avoids premium tier costs
- **Query Optimization**: Efficient SQL queries reduce database load
- **Batch Operations**: Upsert operations reduce database writes

Security & Compliance

Data Access Boundaries

Authentication

- **JWT-Based**: All protected endpoints require valid JWT token
- **Token Expiration**: Configurable expiration (default: 24 hours)
- **Token Validation**: Signature verification, expiration check
- **User Isolation**: Users can only access their own data (agents, chatrooms)

Authorization

- **User-Scoped Resources**:
  - Ambient agents: User can only access their own agents
  - Chatrooms: User can only access their own threads
  - Execution history: User can only view their agent executions
- **Public Resources**:
  - Public chatrooms (if `is_public` flag set)
  - News articles (read-only, no user data)

API Security

Endpoint Protection

- **Authentication Required**: Most endpoints require JWT token
- **Public Endpoints**: Health checks, documentation
- **CORS**: Configured for frontend origin (configurable)
- **Rate Limiting**: Recommended for production (not currently implemented)

Input Validation

- **Pydantic Models**: Request/response validation
- **SQL Injection Prevention**: Parameterized queries, no raw SQL from users
- **XSS Prevention**: Input sanitization (handled by FastAPI)
- **Type Validation**: Strong typing prevents type-based attacks

Model Safety

LLM Safety

- **Prompt Engineering**: System prompts guide model behavior
- **Output Filtering**: No explicit content filtering (relies on model)
- **Tool Restrictions**: Agents can only call predefined tools
- **SQL Validation**: Generated SQL validated (SELECT only, no DDL/DML)

Error Message Safety

- **User-Facing**: Generic error messages (no internal details)
- **Logging**: Detailed errors logged server-side only
- **Stack Traces**: Not exposed to users

Data Privacy Assumptions

User Data

- **Password Storage**: SHA-256 hashed with salt (not plaintext)
- **Email Storage**: Stored in database (used for authentication)
- **Chat History**: Stored in database, user-scoped
- **Agent Queries**: Stored in database, user-scoped

News Data

- **Public Data**: News articles are public information
- **No PII**: Articles don't contain user personal information
- **Source Attribution**: Articles include source URLs (public)

Compliance Considerations

- **GDPR**: User data deletion capability (not currently implemented)
- **Data Retention**: No automatic data deletion policies
- **Audit Logging**: Basic logging (not comprehensive audit trail)
- **Data Export**: Users can access their data via API (not bulk export)

Recommendations for Production

1. **HTTPS**: Enforce HTTPS for all API communication
2. **Rate Limiting**: Implement rate limiting per user/IP
3. **API Keys**: Rotate API keys regularly
4. **Database Encryption**: Encrypt database at rest
5. **Secrets Management**: Use secret management service (not env vars)
6. **Audit Logging**: Comprehensive audit trail for sensitive operations
7. **Data Retention Policies**: Implement data retention and deletion
8. **GDPR Compliance**: User data export and deletion capabilities

Future Extensions

New Data Sources

Additional News APIs

- **RSS Feeds**: Direct RSS feed ingestion
- **Social Media**: Twitter, Reddit news aggregation
- **Specialized Sources**: Financial news (Bloomberg, Reuters), Tech news (TechCrunch, The Verge)
- **Regional Sources**: Non-English news sources with translation

Data Source Integration

- **Web Scraping**: Direct website scraping (with permission)
- **Newsletters**: Email newsletter parsing
- **Podcasts**: Audio transcription and processing
- **Video**: YouTube news channel transcripts

Additional Agents

Specialized Agents

- **Financial News Agent**: Specialized in market analysis, earnings reports
- **Tech News Agent**: Focused on technology, startups, innovation
- **Political News Agent**: Specialized in politics, policy, elections
- **Health News Agent**: Medical breakthroughs, public health

Agent Capabilities

- **Multi-Agent Collaboration**: Agents work together on complex queries
- **Agent Memory**: Long-term memory for user preferences
- **Agent Learning**: Learn from user feedback
- **Custom Agents**: User-defined agent behaviors

Model Upgrades

Embedding Models

- **Larger Models**: Higher-dimensional embeddings (768, 1024 dims)
- **Multilingual Models**: Better support for non-English content
- **Domain-Specific Models**: Fine-tuned models for news domain
- **Multimodal Models**: Support for images, audio, video

LLM Models

- **Larger Context Windows**: Support for more context in agent reasoning
- **Faster Models**: Lower latency for real-time responses
- **Specialized Models**: Models fine-tuned for news analysis
- **Open Source Models**: Self-hosted models for cost reduction

Cross-Modal Expansion

Image Processing

- **Image Embeddings**: Embed article images for visual search
- **OCR**: Extract text from images in articles
- **Image Analysis**: Identify objects, scenes in news images

Audio/Video Processing

- **Transcription**: Convert audio/video to text
- **Audio Embeddings**: Embed audio for search
- **Video Summarization**: Extract key frames and summaries

Global Scale Improvements

Multi-Region Deployment

- **Regional Data Centers**: Deploy services closer to users
- **Data Localization**: Store data in user's region
- **CDN**: Content delivery for static assets

Performance Optimizations

- **Query Optimization**: Advanced SQL query optimization
- **Index Optimization**: Better vector index structures (HNSW, IVF)
- **Caching Layers**: Multi-level caching (Redis, in-memory)
- **Async Processing**: More async operations for better throughput

Scalability Enhancements

- **Distributed Document Store**: Sharded vector index across nodes
- **Kafka Partitioning**: Better partition strategy for load distribution
- **Database Sharding**: Shard news tables by region/date
- **Microservices**: Further service decomposition

Advanced Features

Personalization

- **User Preferences**: Learn user interests and preferences
- **Personalized Feeds**: Custom news feeds per user
- **Recommendation Engine**: Recommend relevant articles
- **Interest Tracking**: Track user interests over time

Analytics

- **Trend Analysis**: Identify trending topics and keywords
- **Sentiment Tracking**: Track sentiment over time
- **Source Reliability**: Score news sources by reliability
- **Fact Checking**: Integration with fact-checking services

Collaboration

- **Shared Agents**: Users can share ambient agents
- **Team Workspaces**: Multi-user workspaces
- **Annotations**: Users can annotate and comment on articles
- **Export/Import**: Export research findings, import external data

Conclusion

6MChunks is a comprehensive real-time news intelligence platform that combines:

- **Multi-source aggregation** with unified data model
- **Real-time processing** using Kafka and Pathway
- **Semantic search** via vector embeddings
- **Intelligent agents** with multi-tool orchestration
- **Scalable architecture** with horizontal scaling capabilities

The system is designed for extensibility, allowing new data sources, agents, and capabilities to be added incrementally while maintaining system reliability and performance.