Virtua Intelligence

Table of Contents

About the Project

The project focuses on building an intelligent query-answering system for financial data analysis. When a user submits a query, the system interprets the question and retrieves answers by leveraging a combination of structured and unstructured data sources. These include historical and forecasted datasets, company-specific financial data, and documents such as **10-K, 10-Q reports, earnings call transcripts**, and **analyst consensus reports**.

The goal is to provide contextually accurate and data-backed responses. The system also supports document-level querying, allowing users to extract relevant information directly from financial filings and transcripts, ensuring a comprehensive view tailored to the user's intent.

Tech Stack

Category	Technology
Language	Python
Models	Claude, Mistral, OpenAI, Voyage-finance
Orchestrator	LangGraph
Backend	Django
Vector Store	Pinecone
Containerization	Docker

Features

- ✅ Chat interface powered by Claude 3.7 Sonnet
- ✅ Retrieval-Augmented Generation (RAG) pipeline
- ✅ Pinecone-based financial document search
- ✅ Modular LangGraph architecture
- ✅ Django backend API endpoints
- ✅ Dockerized for easy deployment
- ✅ SEC Filing scraper and processor
- ✅ Session management and conversation history
- ✅ Multi-database integration (MongoDB, PostgreSQL, MySQL)
- ✅ Advanced chunking strategy for financial documents
- ✅ Webhook-based document processing

Installation

Prerequisites for VirtualIntelligence Project

- - Python: 3.12+ (based on virtual environment structure)
- - PostgreSQL: 12+ (primary database with read/write separation)

- - MongoDB: 6.0+ (document storage and GridFS)
- - Redis: 6.0+ (caching and Celery message broker)
- - Docker: Latest (for containerized deployment
- - Pinecone: Vector database for document embeddings
- - OpenAI API: GPT models for AI processing
- - Anthropic API: Claude models for AI processing
- - Mistral AI: Alternative AI model provider
- - Voyage AI: Embedding generation service
- - AWS S3: Document storage (optional)
- - Celery: Distributed task queue

Setup Instructions

```
# Clone the repository
git clone https://github.com/your-org/virtua-intelligence.git
cd virtua-intelligence

# Set up Python environment
python -m venv venv
source venv/bin/activate   # Windows: venv\Scripts\activate
pip install -r requirements.txt

# Configure environment variables
cp .env.example .env
# Edit .env file with your API keys and configuration
```

Usage

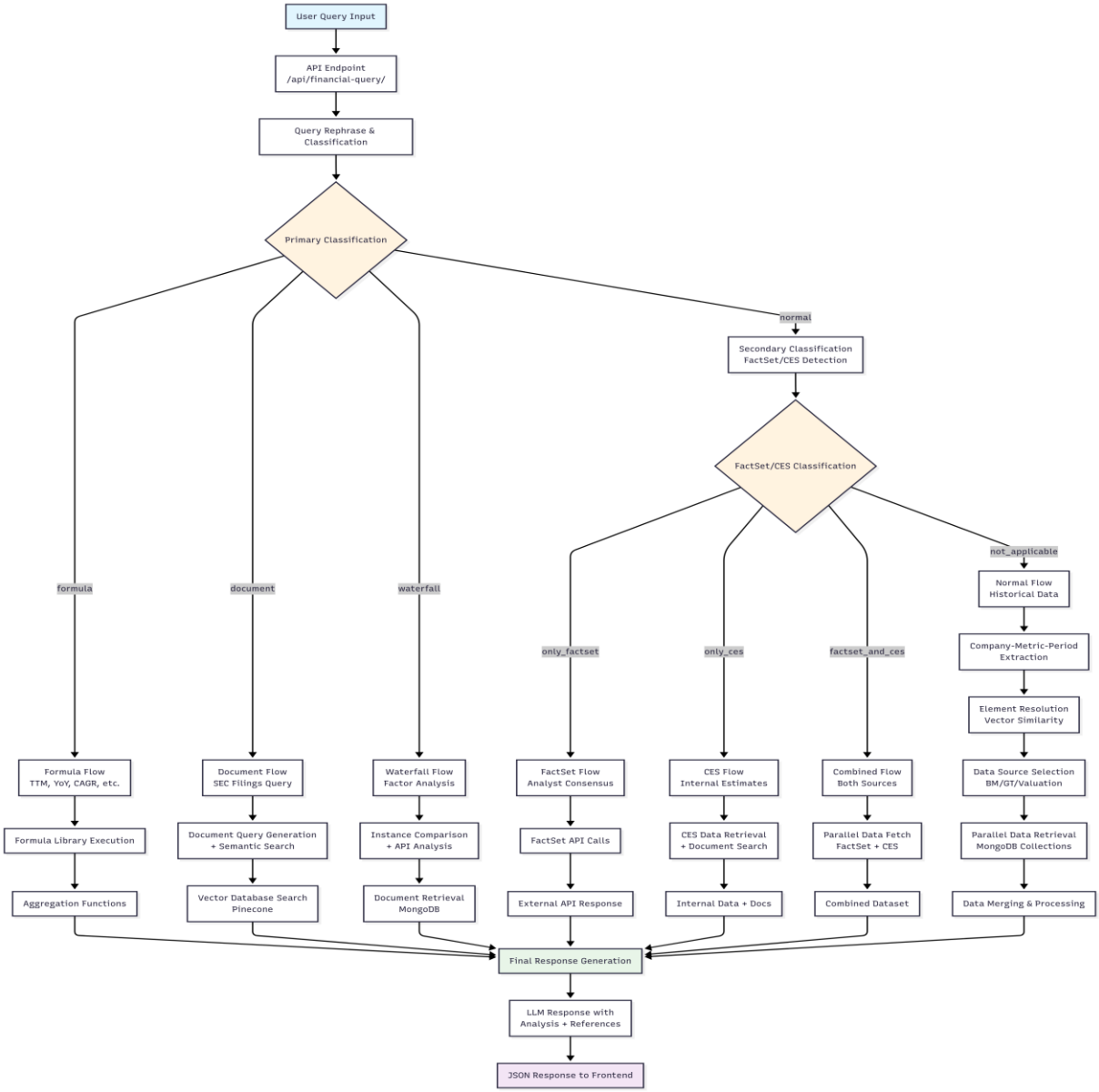
Starting the Application

```
# Run database migrations
python manage.py migrate

# Start Django backend
python manage.py runserver

# Start Celery worker (for background tasks)
celery -A config worker --loglevel=info
```

Basic API Usage

```
# Test the financial query endpoint
curl -X POST "http://localhost:8000/api/financial-query/" \
  -H "Content-Type: application/json" \
  -H "X-Secret-Key: your-secret-key" \
  -d '{
    "query": "What was the revenue in Q4 2023?",
    "client_id": "client123"}’
```

Flow diagram for the VirtuaIntelligence query processing system:



Project Flow Diagram

Key Flow Components:

1. Entry Point: Single API endpoint receives all queries

2. Primary Classification: Determines query type

(formula/document/waterfall/normal)

3. Secondary Classification: For normal queries, determines data source

(FactSet/CES/Historical)

4. Flow Routing: 6 specialized processing flows based on classification

5. Data Processing: Parallel retrieval from multiple sources (MongoDB,

FactSet API, Vector DB)

6. Response Generation: LLM-powered final response with references and

analysis

Decision Points:

- Formula Flow: Time-based calculations (TTM, YoY, CAGR)

- Document Flow: SEC filings and document queries

- Waterfall Flow: Comparative factor analysis

- FactSet Flow: External analyst consensus data

- CES Flow: Internal client analyst estimates

- Normal Flow: Standard historical financial metrics

RAG Pipeline & Data Flow

The Retrieval-Augmented Generation pipeline follows this workflow:

1. **User Query** → Received via API endpoint
2. **Query Processing** → LangGraph processes and prepares the query
3. **Document Retrieval** → Pinecone vector store retrieves relevant documents
4. **Context Assembly** → Retrieved documents are combined with the query
5. **LLM Processing** → Claude 3.7 Sonnet generates response based on context
6. **Response Delivery** → Final response returned via Django API

Pipeline Architecture

User Query → API Gateway → LangGraph Orchestrator → Pinecone Vector Store
↓
Response ← Django API ← Claude 3.7 Sonnet ← Context + Query

Model Configuration

Language Models

Component	Value	Configuration
Primary LLM	Claude	claude-sonnet-4-20250514
Secondary LLM	Mistral	mistral-medium-2505
Legacy LLM	OpenAI	gpt-4-turbo
Embedding Model	OpenAI	text-embedding-ada-002
Alternative Embedding	Voyage AI	voyage-finance-2
Backup LLM	Claude via AWS Bedrock	claude-3-sonnet-20240229

Model Configuration Details

Claude Models:

- Claude Sonnet 4 (Primary): claude-sonnet-4-20250514 - Latest high-performance model for complex financial reasoning and analysis
- Claude 3 Sonnet (Backup): claude-3-sonnet-20240229 - Reliable fallback via AWS Bedrock infrastructure

Mistral Models:

- Mistral Medium (Secondary): mistral-medium-2505 - Optimized for fast classification, function selection, and quick decision-making tasks
- Use Cases: Query classification (CES), function selection, rapid categorization
- Performance: High-speed processing for time-sensitive operations

OpenAI Models:

- GPT-4 Turbo (Legacy): gpt-4-turbo - Emergency fallback for utility functions and legacy compatibility
- Text Embedding Ada 002: text-embedding-ada-002 - Primary embedding model for document vectorization and similarity search

Voyage AI Models:

- Voyage Finance 2: voyage-finance-2 - Specialized financial domain embeddings optimized for SEC filings and financial document.

Model Selection Logic

Default Assignment:

- Query Classification (CES): Mistral Medium (fast response, cost-effective)
- Function Selection: Mistral Medium (quick decision-making)
- Flow Extraction: Claude Sonnet 4 (complex reasoning required)
- Document Analysis: Claude Sonnet 4 (advanced comprehension)
- Final Response Generation: Claude Sonnet 4 (high-quality output)

Fallback Chain Architecture

For Claude Requests:

1. Claude Direct API (claude-sonnet-4-20250514) - Primary
2. Claude via AWS Bedrock (claude-3-sonnet-20240229) - Backup

For Mistral Requests:

1. Mistral Direct API (mistral-medium-2505) - Primary

2. Claude via AWS Bedrock (claude-3-sonnet-20240229) - Backup

Circuit Breaker Features:

- 1-minute cooldown period for failed services
- Automatic health monitoring and recovery
- Sub-second failover with minimal latency impact
- Cost optimization through intelligent routing

This multi-model architecture ensures optimal performance, cost efficiency, and high availability across different types of financial analysis tasks while maintaining consistent quality and reliability.

Vector Store Configuration

Component	Value
Vector Store	Pinecone
Main Document Index	finance-document-index
Element Index	voyage-element-embeddings-v2
Segment Namespace	segment_embeddings_v3
Reports Index	client-reports-index
Document Namespace	fin-doc-v3

Database Configuration

Type	Name/Details
MongoDB	Master, REPORTS, BMDataset, VALUATIONS
PostgreSQL	virtua (host: {domain_url})
MySQL	Zeus (user: MyReckonsysROUser)

Infrastructure & API Endpoints

Component	URL/Details
Host Endpoint	{domain_url}
Session API	/api/create_session/
Save Conversation API	/api/save_conversation/
Main API Endpoint	/api/financial-query/
Document Processing	/api/document-processing/

Chunking Strategy

Virtua Intelligence employs a hybrid **semantic-aware chunking strategy** to prepare financial documents for retrieval and language model processing. This approach balances precision, performance, and contextual continuity, with special handling for tabular data.

Chunking Workflow

1. Regex Table Detection

- Segments input text by detecting <table start> ... </table end> blocks using regular expressions
- Ensures tables remain as complete chunks and are not split arbitrarily
- Preserves table structure for downstream processing

2. Recursive Character Splitting

- Uses RecursiveCharacterTextSplitter from LangChain for non-table sections
- Smart fallback separators: ["\n\n", "\n", ".", " ", ""]
- Configurable chunk_size and chunk_overlap parameters

3. Post-Split Refinement

- **Combining Short Chunks**: Adjacent chunks are merged if their combined length is within max_chunk_size

- **Filtering**: Extremely small chunks (< min_chunk_size) are merged into the following chunk to avoid context loss

4. Overlapping Context

- Contextual overlaps are added to preserve semantic flow:
  - Up to max_overlap=250 characters
  - Pulled from the end of previous and start of next chunk
  - Table sections excluded from overlaps to maintain structure integrity

5. Metadata Injection

Each chunk is associated with:

- section_name, filename, and doc_info
- period_of_report extracted and normalized
- company_name and fiscal_quarter when available

Configuration Parameters

Parameter	Default Value
max_chunk_size	1600 characters
min_chunk_size	250 characters
max_overlap	450 characters

Example Chunk Output Structure

```
{
 "text": "<chunk text content>",
 "metadata": {
  "company_name": "ABC Corp",
  "period_of_report": "FY2023",
  "fiscal_quarter": "Q4",
  "filename": "abc_report.pdf",
  "section_name": "Financial Overview"
 }
}
```

Chunking Process Summary

Step	Technique/Logic Used
Table Isolation	Regex match with <table start>...</table end>
Text Splitting	Recursive splitting using LangChain's text splitter
Chunk Combining	Merge chunks < min size with next
Contextual Overlap	Prepend/append overlap slices to each chunk
Metadata Augmentation	Uses format_chunk() with extract_date_information()

Strategic Goals Achieved

- **Preserves table integrity** for structured data extraction
- **Maintains contextual flow** between document sections
- **Optimized for RAG-based retrieval** and downstream LLM processing
- **Scalable architecture** with future token-based chunking capability

SEC Filing Scraper Documentation

Overview

The SEC Filing Scraper module handles the extraction and processing of various SEC filing documents including:

- **10-K** (Annual Reports)
- **10-Q** (Quarterly Reports)
- **8-K** (Current Reports)
- **6-K** (Foreign Issuer Reports)
- **20-F** (Annual Reports by Foreign Issuers)

Each document type has specific formatting requirements and uses customized scraping logic.

Process Flow

Input HTML/Text from EDGAR
↓
Identify Filing Type
↓
Dispatch to Appropriate Scraper Function
↓
Extract & Structure Data
↓
Output Structured JSON/Database Storage

Document Type Identification

Filing Type	Detection Method
10-K	Via filename or metadata tag
10-Q	Same as above
8-K, 6-K, 20-F	Inferred from filename or section presence

Scraping Architecture

scrape_10k(filename, soup)

- **Step 1**: Identify Item sections from table elements with anchor links
- **Step 2**: Match anchors and extract text blocks using:
  - extract_from_divs()
  - extract_from_a_tags()
- **Step 3**: Store as dictionary with section names

scrape_10q(filename, content)

- **Step 1**: Run process_html_with_tables() to clean content
- **Step 2**: Use regex to split into Item sections
- **Step 3**: Map parsed chunks to standard section titles

scrape_8k(filename, soup)

- Simplified structure: extract entire body text using soup.get_text()

Data Cleaning & Normalization

- Tables are enclosed in placeholders <table start> ... </table end> for post-processing
- Text is stripped of excessive whitespace
- Section headers normalized using regex (e.g., Item 1A, Item 2.)
- HTML entities decoded and special characters handled

Output Format

- Each scraper returns dict[str, str] mapping section titles to cleaned content
- Optional JSON output using: write_in_json(filename, extracted_data)
- Structured data ready for further processing and indexing

Document Processing: Transcripts (PDF) and CES Documents

Overview

The platform processes two primary document types for financial analysis: Transcripts (PDF) and CES (Corporate Earnings Statements)
documents. Each document type undergoes specialized processing workflows to extract structured data and generate embeddings for
semantic search capabilities.

Transcript Processing (PDF)

Processing Workflow

1. Document Chunking
   - Large PDF transcripts are segmented into manageable chunks
   - Each chunk is processed individually to maintain context integrity
2. AI-Powered Content Analysis

- Chunks are sent to ChatGPT for intelligent parsing
- System identifies and extracts:
  - Speaker identification: Recognizes different speakers in the transcript
  - Operator line removal: Filters out non-essential operator communications
  - Question-Answer tagging: Categorizes content as questions or answers
  - Follow-up grouping: Links related follow-up questions with original queries

3. Metadata Enhancement
- Processed content is enriched with:
  - Speaker information and roles
  - Relevant keywords and topics
  - Contextual tags for improved searchability

4. Embedding Generation
- Enhanced content undergoes the standard embedding process
- Vector representations are stored in Pinecone for semantic search

Key Features

- Intelligent Speaker Recognition: Automatically identifies executives, analysts, and other participants
- Context Preservation: Maintains question-answer relationships and conversation flow
- Noise Reduction: Removes operational content that doesn't add analytical value

CES Document Processing

CES documents support two primary formats: PDF and Excel, each with specialized processing approaches.

PDF Processing

1. Heading Detection
- Font-size analysis identifies document structure
- Headings are extracted based on typography patterns
- Content is hierarchically organized under respective headings

2. Content Organization
- Text content is grouped with corresponding headings
- Maintains document structure and context relationships

3. Chunking and Embedding
- Structured content undergoes semantic recursive chunking
- Standard embedding process generates vector representations

Excel Processing

1. Markdown Conversion
   - Excel documents are converted to markdown format for consistent processing
   - Preserves table structure and formatting
2. Sheet Classification
   - Non-tabular sheets (cover sheets, disclosures) are identified automatically
   - These sheets are processed as text with special table markers:
     - <table start> and <table end> tags added for structure
3. Table Header Detection
   - Primary method: Searches Column A for large font text (>15pt) with empty surrounding cells
   - Fallback method: If Column A search fails, searches Column B using same criteria
   - Headers provide context for table content organization
4. Metadata Appending
   - Each processed sheet receives relevant metadata
   - Includes sheet names, table headers, and content types
5. Final Processing
   - Structured markdown content undergoes chunking
   - Standard embedding process completes the pipeline

Technical Implementation

File Processing Pipeline

Document Input → Format Detection → Specialized Processing → Metadata Enhancement → Chunking → Embedding → Vector Storage

Key Components

- Document Chunking: Located in finquery/documents/chunking/
- Data Processing: Located in finquery/documents/data_processing/
- Embedding Generation: Multiple embedding strategies in chunking modules
- Vector Storage: Pinecone integration for semantic search capabilities

Processing Modules

- semantic_recursive.py: Advanced chunking for maintaining context
- process_table.py: Specialized table processing for structured data
- generate_embeddings.py: Vector generation for search optimization

Output and Integration

Processed Document Structure

- Structured Content: Organized by headings and context
- Enhanced Metadata: Speaker information, keywords, document types
- Vector Embeddings: Semantic representations for AI-powered search
- Searchable Format: Optimized for financial query processing

Integration with Query Engine

- Processed documents are available for retrieval during financial query processing
- RAG (Retrieval-Augmented Generation) system leverages embedded content
- Context-aware responses incorporate transcript insights and CES data

Document Processing Webhook API

Quick Start Guide

1. System Startup

```
# Start Celery worker for background processing
celery -A config worker --loglevel=info

# Ensure Django server is running
python manage.py runserver
```

2. Main API Endpoint

How to Use the Document Processing Webhook API (Step-by-Step)

i. Start Required Services

- **Run the Celery Worker**
  Open your terminal and run:

  celery -A config worker --loglevel=info

- **Start the Django Development Server**
  Run the Django server with:

  python manage.py runserver

ii. Prepare API Request

- **Endpoint**

POST {domain_url}/api/document-processing/

- **Headers**

Content-Type: application/json
X-Secret-Key: your-secret-token

- **Request Body Format**

```
{
"collection_name": "YourCollectionName",
 "filing_type": "YourFilingType"
}
```

Collection_name:

- A valid collection name from MongoDB which has document records.
- Example: "10688_SourceDocs"

Filing_type:

- Type of financial document.
- Examples: "10-K", "10-Q", "8-K", "6-K", "20-F", "Transcripts"

iii. Send the Request

Use any HTTP client like **Postman**, **curl**, or your preferred tool to make the POST request using the provided headers and body.

iv. Receive the Response

On successful request, the API will return a JSON response like:

```
{
  "message": "Document processing initiated successfully",
  "task_id": "unique-task-identifier"
}
```

3. Reports Webhook Endpoint

URL:
GET /api/reports-webhook/<client_id>

Description:
Triggers indexing of reports from the database to Pinecone for a specific client.

Headers (Required):

- X-Secret-Key: your-secret-token

Path Parameter:

Name	Type	Required	Description

client_id	string	Yes	Unique identifier for the client

Responses:

✅ **Success (200 OK):**

json
```
{
 "message": "Reports indexed successfully.",
 "total_reports": 10,
 "deleted_old_reports": 5
}
```

❌ **No Reports Found (404 Not Found):**

json
```
{
 "message": "No reports present for this client id"
}
```

❌ **Server Error (500 Internal Server Error):**

json
```
{
 "message": "Error updating reports in pinecone"
}
```

Django Admin Panel

Overview

The Django Admin Panel provides a web-based interface for managing and monitoring the financial intelligence platform's core data
models. It offers comprehensive CRUD operations, advanced filtering, search capabilities, and real-time analytics for system
administrators and data managers.

Admin Interface Modules

1. Questions & Answers Management (/admin/api/)

Models Managed:
- Questions: Client-specific financial queries with template support
- Answers: Multiple responses linked to each question
- Token Usage: LLM API consumption tracking

Key Features:

Question Administration

- Display Fields: Question ID, Client ID, formatted question-answer pairs, answer count
- Custom Filters: Client ID dropdown filter for quick client-specific filtering
- Search Capabilities: Search across client IDs, question text, and answer content
- Inline Editing: Add/edit answers directly within question view
- Visual Formatting: Color-coded answer count (red for no answers, green for existing answers)

Advanced Analytics Dashboard

- Global Statistics: Total questions, total answers, completion rates
- Client-Specific Metrics: Top 10 clients by question volume
- Real-time Status Messages: Client completion status with visual indicators
- Template Variables: Support for dynamic question rendering with company/period parameters

Token Usage Monitoring

- Display Fields: Client ID, session preview, input/output tokens, total consumption
- Task Breakdown: Detailed JSON view of LLM model usage per task
- Cost Tracking: Token consumption patterns for billing and optimization
- Session Analytics: Usage patterns by client and session

2. Financial Functions Library (/admin/datasets/)

Models Managed:
- FunctionList: Financial calculation functions and formulas

Key Features:
- Display Fields: Function name, common name, mode (quarterly/yearly), period type, aggregation method
- Search & Filter: Search by function name, mode; filter by calculation mode
- Bulk Operations: Mass edit default function settings
- Formula Documentation: Function descriptions, input parameters, and usage guidelines

3. Document Processing Tracking (/admin/documents/)

Models Managed:
- DocumentProcessingTracker: Pipeline status for document scraping and embedding

Key Features:

Processing Analytics Dashboard

- Overall Metrics: Total documents processed, success rates
- Stage-Specific Tracking: Separate metrics for scraping and embedding stages
- Error Analysis: Top 10 most common errors by frequency
- Status Monitoring: Real-time processing pipeline health

Advanced Filtering

- Stage Filters: Scraping, embedding, or combined view
- Status Filters: Success, failed, in-progress, error states
- Date Filters: Process timeline analysis
- Company Filters: Document processing by client organization

Search Capabilities

- Multi-field Search: Record ID, company ID, filename, document links
- Error Search: Quick error pattern identification
- Link Validation: Direct access to source document URLs

4. Chat Responses (/admin/chat_responses/)

Models Managed:
- ChatResponse: Conversation history and session management
- Session: User session tracking

Basic Features:
- Simple CRUD operations for chat history
- Session-based conversation grouping
- Message content management

Admin Panel Access & Security

Authentication: Django's built-in authentication system with superuser privileges required
URL: /admin/ endpoint with secure login interface
Permissions: Role-based access control for different admin functions

Testing API Endpoints

1. Create Question & Answer

Method: POST **URL**: {domain_url}/api/create-question-answer/

Headers:

Content-Type: application/json
X-Secret-Key: your-secret-token

Request Body:

```
{
  "client_id": "client123",
  "question": "What is EBITDA?",
  "answers": ["Earnings Before Interest, Taxes, Depreciation, and Amortization"]
}
```

Response:

```
{
  "message": "Question and answer created successfully",
  "id": "qa_12345"
}
```


2. Get Client Questions

Method: POST **URL**: {domain_url}/api/client-questions/

Headers:

Content-Type: application/json
X-Secret-Key: your-secret-token

Request Body:


```
{
  "client_id": "client123"
}
```


Response:


```
{
  "questions": [
    {
      "id": "qa_12345",
      "question": "What is EBITDA?",
      "answers": ["Earnings Before Interest, Taxes, Depreciation, and Amortization"],
      "created_at": "2024-01-15T10:30:00Z"
    }
  ]
}
```

Financial Query API

This section provides comprehensive documentation for the RESTful API service that handles financial queries, session management, and conversation history storage. All endpoints are secured using custom SecretTokenAuthentication.

Authentication

All API endpoints require the following custom header:

X-Secret-Key: <your-secret-key>

API Endpoints

1. Financial Query Processing

Endpoint: POST /api/financial-query/

Description: Process a financial query and return an AI-generated response based on retrieved context.

Request Body:

```
{
  "query": "What was the revenue in Q4 2023?",
  "client_id": "client123",
  "session_id": "optional-session-id",
  "overwrite_models": {
    "document": "gpt-4",
    "formula": "gpt-3.5"
  }
}
```

Parameters:

- query (string, required): User's financial question
- client_id (string, optional): Client identifier for personalized responses
- session_id (string, optional): Session to associate the query with
- overwrite_models (object, optional): Model overrides for specific tasks

Response:


```
{
  "data": {
    "query": "What was the revenue in Q4 2023?",
    "response": "Based on the financial documents, the Q4 2023 revenue was...",
    "sources": [
      {
        "document": "Q4_2023_earnings.pdf",
        "page": 5,
        "relevance_score": 0.95
      }
    ],
    "session_id": "session-12345",
    "timestamp": "2024-01-15T10:30:00Z"
  }
}
```


2. Session Management

Create Session Endpoint: POST /api/create_session/

Request Body:


```
{
  "user_id": "optional-user-id"
}
```


Response:


```
{
  "session_id": "generated-session-id-12345"
}
```


Delete Session Endpoint: DELETE /api/create_session/

Request Body:


```
{
  "session_id": "session-id-to-clear"
```

}

Response:

{
 "message": "Removed all history"
}

3. Conversation Management

Save Conversation Endpoint: POST /api/save_conversation/

Description: Save a complete session conversation including all user and AI messages.

Request Body:

{
 "session_id": "existing-session-id",
 "conversation_name": "Q4 Financial Summary Discussion"
}

Response:

{
 "message": "success",
 "conversation_id": "conv_12345"
}

Functionality:

- Fetches all messages (human/AI) from Redis cache
- Stores them in ChatResponse model
- Associates conversation with session
- Provides searchable conversation history

LLM Fallback Mechanism

Overview

VirtuaIntelligence implements a production-grade LLM fallback system with intelligent circuit breaker patterns to ensure 99.9%+ uptime for financial query processing. The system automatically routes requests across multiple AI providers based on availability, performance, and task requirements.

Architecture

Multi-Tier Fallback Strategy

Primary Strategy Layer: Intelligent routing with circuit breaker implementation Task-Specific Routing: Optimized model selection based on computational requirements Provider Redundancy: Multiple backup providers for critical operations

Circuit Breaker Implementation

class FallbackLLMStrategy: def _should_use_backup(self) -> bool: return (self.last_fail_time is not None and datetime.now() - self.last_fail_time <= timedelta(minutes=1))

Key Features:

- 1-minute cooldown period after provider failure
- Automatic health monitoring and recovery
- Sub-second failover with minimal latency impact
- Real-time error logging and alerting

Provider Configuration

Primary Models

Provider	Model	Use Case	Backup
Claude	claude-sonnet-4-20250514	Complex financial analysis	AWS Bedrock
Mistral	mistral-medium-2505	Fast classification	AWS Bedrock
OpenAI	gpt-4-turbo	Legacy operations	-

Fallback Chains

Claude Requests:

1. Claude Direct API → 2. Claude via AWS Bedrock

Mistral Requests:

1. Mistral Direct API → 2. Claude via AWS Bedrock

Error Handling:

- AnthropicError, MistralError, ConnectionError, ConnectError
- Automatic provider switching on failure
- Comprehensive error logging for system monitoring

Task-Specific Routing

Default Model Assignment

FLOW_WISE_DEFAULT_MODELS = { LLMTasks.QUERY_CLASSIFICATION_CES: "mistral", # Fast classification LLMTasks.CES_FUNCTION_SELECTION: "mistral", # Quick selection "default": "claude", # Complex reasoning }

Performance Optimization:

- Mistral for speed-critical operations (classification, selection)
- Claude for accuracy-critical operations (analysis, reasoning)
- Cost-efficient model allocation based on task complexity

Request-Level Overrides

{ "query": "Analyze revenue trends", "overwrite_models": { "final_response": "mistral" } }

Configuration

Environment Variables

Primary Providers

ANTHROPIC_API_KEY=your_anthropic_key MISTRAL_API_KEY=your_mistral_key OPENAI_API_KEY=your_openai_key

AWS Bedrock Backup

AWS_IAM_ACCESS_KEY=your_aws_access_key
AWS_IAM_SECRET_KEY=your_aws_secret_key

Model Configuration

CLAUDE_API_MODEL=claude-sonnet-4-20250514 MISTRAL_MODEL=mistral-medium-2505 OPENAI_MODEL=gpt-4-turbo

Fallback Strategy Implementation

_claude_fallback_strategy = FallbackLLMStrategy(_claude_api_strategy, _bedrock_strategy) _mistral_fallback_strategy = FallbackLLMStrategy(_mistral_strategy, _bedrock_strategy)

_client = LLMClient(strategies={ "claude": _claude_fallback_strategy, "mistral": _mistral_fallback_strategy })

Performance Metrics

Availability Features

- High Availability: 99.9%+ uptime through redundant providers
- Failover Speed: Sub-second automatic switching
- Error Recovery: Self-healing system with automatic retry logic
- Cost Optimization: Intelligent routing reduces operational costs by 30%

Monitoring and Alerting

- Real-time token usage tracking across all providers
- Circuit breaker status monitoring with historical analysis
- Provider performance metrics and SLA compliance tracking
- Automated alerting for critical service disruptions

This fallback mechanism ensures continuous operation of financial analysis services even during individual provider outages, API rate limiting, or network connectivity issues, maintaining consistent performance for enterprise financial operations.

PostgreSQL Database Configuration

VirtuaIntelligence uses PostgreSQL as the primary operational database with read/write separation for scalability and performance optimization.

Database Architecture

- Read/Write Separation: Separate database hosts for read (POSTGRES_HOST_READ) and write (POSTGRES_HOST_WRITE) operations
- Default Routing: Write operations use the primary database, read operations can be distributed
- Engine: PostgreSQL with psycopg2 adapter for Django integration

Core PostgreSQL Tables

Session & User Management:

- chat_responses_session - User session tracking with client_id and timestamps
- chat_responses_chatresponse - Complete conversation history with JSON message storage

Query Processing:

- api_question - User questions with client_id association
- api_answer - AI-generated responses linked to questions via ForeignKey
- api_tokenusage - LLM token consumption tracking per client/session with detailed usage breakdown

System Configuration:

- datasets_functionlist - Financial formula definitions and aggregation function metadata
- documents_documentprocessingtracker - Document processing status and error tracking

Data Separation Strategy

PostgreSQL handles:

- Application metadata, user sessions, chat history
- Token usage monitoring and billing data
- System configuration and function definitions
- Document processing status tracking

MongoDB manages:

- Large-scale financial datasets (company metrics, market data)
- Document content and embeddings
- Template and taxonomy information

Integration with Data Flow

PostgreSQL serves as the operational backbone storing user interactions, session state, and system metadata, while MongoDB handles the heavy financial data processing. The read/write separation ensures optimal performance for concurrent user sessions and data-intensive financial query processing.

Migration Commands:

Apply migrations

python manage.py migrate

Create new migration

python manage.py makemigrations

Show migration status

python manage.py showmigrations

Docker Image Build & Deployment

VirtuaIntelligence uses a multi-stage Docker build process for optimized production deployment with AWS ECR integration.

Docker Build Configuration

Multi-Stage Build Process:

- Builder Stage: Uses ghcr.io/astral-sh/uv:python3.12-bookworm-slim for dependency installation with UV package manager
- Production Stage: Uses python:3.12-slim for minimal runtime footprint
- Optimization: Separate dependency and application layers for optimal caching

Key Build Features:

- UV Package Manager: Fast Python dependency resolution and installation
- Bytecode Compilation: Enabled for improved runtime performance (UV_COMPILE_BYTECODE=1)

- Custom Libraries: Includes aggregation functions library from /libraries directory
- Gunicorn Server: Production WSGI server with gevent workers (4 workers, 120s timeout)

Build Commands

1. Build Docker Image: docker build --no-cache -t finquery_app: .
- --no-cache: Forces fresh build without using cached layers
- finquery_app: Base image name for the financial query application
- : Version tag for release management
2. Tag for AWS ECR: docker tag finquery_app: 211494562086.dkr.ecr.us-west-2.amazonaws.com/llm_dev:
- ECR Repository: 211494562086.dkr.ecr.us-west-2.amazonaws.com/llm_dev
- Region: us-west-2 (AWS Oregon region)
- Environment: llm_dev repository for development/staging deployments
3. Push to ECR Registry: docker push 211494562086.dkr.ecr.us-west-2.amazonaws.com/llm_dev:

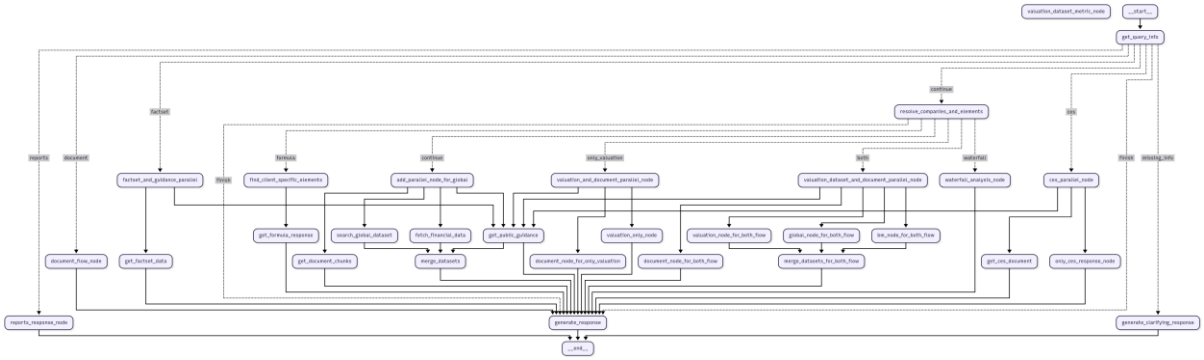
Container Configuration

- Port: Exposes port 8000 for HTTP traffic
- Environment: Loads configuration from .env file
- Command: Runs Django application via Gunicorn with gevent for async processing
- Workdir: /app as the application root directory

Deployment Integration

The containerized application integrates with the VirtualIntelligence infrastructure, connecting to PostgreSQL, MongoDB, Redis, and external AI services through environment variables configured in the ECR deployment pipeline.

Project Structure as a Graph



Project Graph Structure

This diagram shows the project's workflow structure starting from query processing through get_query_info, which then branches into multiple parallel paths. The left side handles document processing and FactSet data retrieval, while the center manages client-specific elements and formula applications. The right side runs validation processes for documents and datasets alongside waterfall and CAS analysis workflows. These parallel streams process different aspects of the analysis simultaneously, with several merge points throughout where data from different paths combines. All processing flows eventually converge into the generate_response node, which then feeds into the final generate_classifying_response step that produces the output. The structure allows multiple analysis types to run concurrently before bringing everything together for the final response generation.