RIA25 Complete Documentation

Research Insights Assistant 2025

Research Insights Assistant Team

April 18, 2025

Table of Contents

# RIA25 Complete Documentation

This directory contains comprehensive documentation for the full end-to-end development of the RIA25 (Research Insights Assistant 2025) project. The documentation covers all aspects of the project from initial planning through implementation, testing, deployment, and maintenance.

## Directory Structure

### Core Documentation Files (Numbered Sequence)

1. **Documentation Overview** (00\_README.md)
   * This file - provides a map of all documentation
2. **Project Overview** (01\_project\_overview.md)
   * Executive summary, goals, timeline, stakeholders, and milestones
3. **Implementation Plan** (02\_implementation\_plan.md)
   * Detailed requirements and implementation planning documents
4. **Data Architecture**
   * **A. Data Processing Workflow** (03\_data\_processing\_workflow.md)
     + Core data processing pipeline and workflow
   * **B. Normalized Data Strategy** (04\_normalized\_data\_strategy.md)
     + Data normalization approach and implementation
   * **C. 2025 Survey Questions** (05\_survey\_questions\_and\_responses.md)
     + Reference of survey questions and response formats
5. **System Architecture** (06\_system\_architecture.md)
   * Overall system design, component diagrams, file access modes, and integration points
6. **Prompt Engineering** (07\_prompt\_evolution.md)
   * Evolution of the prompt system, design principles, and implementation details
7. **Vector Store Reference** (08\_vector\_store\_reference.md)
   * Vector database setup, data embedding strategies, and retrieval methods
8. **Development Timeline** (09\_development\_timeline.md)
   * Development process, timeline, technical challenges, and code organization
9. **Testing Methodology** (10\_testing\_methodology.md)
   * Testing methodologies, test cases, and known limitations
10. **Deployment Guide** (11\_vercel\_deployment\_guide.md)
    * Deployment procedures, infrastructure diagrams, and environment configurations
11. **Maintenance Procedures** (12\_maintenance\_procedures.md)
    * Update procedures, troubleshooting guides, and monitoring strategies
12. **Canonical Topic Reference** (13\_canonical\_topic\_reference.md)
    * Reference of canonical topics and their mappings to file IDs
13. **API Reference** (14\_api\_reference.md)
    * Comprehensive documentation of all API endpoints, request/response formats
14. **Thread Data Management** (15\_thread\_data\_management.md)
    * Thread and cache management system, incremental loading
15. **Glossary** (16\_glossary.md)
    * Terminology definitions, external references, and bibliographic information

### Supporting Files

* **generate\_documentation.sh**
  + Script to generate combined documentation in various formats
* **RIA25\_Complete\_Documentation.docx**
  + Most recent generated complete documentation file

## Target Audiences

This documentation serves multiple audiences:

1. **Developers**
   * Technical implementation details
   * API references
   * Code architecture
   * Development practices
2. **System Administrators**
   * Deployment procedures
   * Environment configuration
   * Maintenance tasks
   * Monitoring and alerting
3. **Data Analysts**
   * Data structure information
   * Query capabilities
   * Analysis methodologies
   * Data processing workflows
4. **Project Stakeholders**
   * Project overview
   * Implementation plans
   * Development timeline
   * System capabilities

## How to Use This Documentation

* Start with the Project Overview section for a high-level understanding
* If you’re looking for specific technical details, navigate to the relevant section
* Cross-references are provided throughout the documentation to help navigate between related topics
* All documentation is version-controlled to track changes over time

## Contributing to Documentation

When contributing to this documentation:

1. Follow the established structure and templates
2. Ensure all technical claims are accurate and verified
3. Include relevant code examples where appropriate
4. Add standard headers with last updated date, target audience, and related documents
5. Update this README if adding new sections or significant content

## Implementation Plans

For information on planned and implemented features, please see the implementation plans in:

* /RIA25\_Documentation/plans/complete/ - Implemented features
* /RIA25\_Documentation/plans/current/ - Planned/in-progress features

## Documentation Version

Current documentation version: 2.0  
Last updated: April 18, 2025

ewpage

# RIA25 Project Overview

## Project Summary

RIA25 (Research Insights Assistant 2025) is an AI-powered system designed to provide actionable insights from the 2025 Global Workforce Survey data. Building on the foundation of its predecessor, RIA25 offers enhanced capabilities for data comparison, visualization, and analysis while ensuring accuracy and preventing data fabrication. The system leverages advanced prompt engineering, vector store retrieval, and the OpenAI API to deliver comprehensive workforce insights based on survey data from both 2025 and 2024 (where comparable).

## Project Goals

1. **Data-Driven Insights**: Provide accurate, data-driven workforce insights from survey responses
2. **Nuanced Analysis**: Deliver analysis that respects the complexity of demographic segments without oversimplification
3. **Year-over-Year Comparison**: Enable valid comparisons between 2024 and 2025 data where methodologically sound
4. **Demographic Segmentation**: Support analysis across various demographic dimensions while preventing invalid cross-segmentation
5. **User-Friendly Interface**: Create an intuitive interface for accessing complex survey data insights
6. **Ethical AI Deployment**: Ensure responsible implementation with guardrails against misinformation and bias

## Development Phases

| Phase | Timeframe | Key Activities |
| --- | --- | --- |
| Planning | January-February 2024 | Requirements gathering, system design, data strategy development |
| Development | March 2024 | Data processing, prompt engineering, vector store implementation |
| Testing | Late March 2024 | System testing, prompt refinement, accuracy validation |
| Deployment | April 2024 | Production deployment, monitoring setup, documentation |
| Maintenance | Ongoing | Updates, enhancements, and support |

## Project Roles

* **Developers**: Responsible for system implementation and maintenance
* **Data Engineers**: Handling data processing and transformation
* **End Users**: Business stakeholders accessing workforce insights

## Core Components

1. **Data Processing Pipeline**: Transforms raw CSV survey data into structured JSON format, with advanced harmonization and canonical topic mapping (see process\_survey\_data.js, process\_2025\_data.js)
2. **Vector Store**: OpenAI vector database for efficient retrieval of relevant survey data, integrated via utility modules and API endpoints
3. **Prompt System**: Carefully designed prompt engineering to guide AI responses and enforce data integrity, implemented in prompt templates and utility modules
4. **API Layer**: Next.js API endpoints (chat-assistant, retrieve-data, etc.) orchestrate query handling, data retrieval, validation, and logging
5. **Utility Modules**: Core logic for retrieval, validation, and caching (utils/openai/retrieval.js, utils/validation/data-validation.js, utils/cache-utils.ts)
6. **Web Interface**: User-friendly front-end for querying the system
7. **Documentation**: Comprehensive documentation for usage and maintenance

### System Structure Update (2025)

* **Data Harmonization**: The pipeline now includes harmonization logic for 2024/2025 data and canonical topic mapping, enabling valid year-over-year comparison and robust metadata.
* **API Endpoints**: The backend is structured around Next.js API endpoints, with chat-assistant/route.ts as the main orchestrator for user queries and retrieve-data/route.js for direct data access.
* **Utility Modules**: Retrieval, validation, and caching are handled by dedicated modules, ensuring modularity and maintainability.
* **Vector Store Integration**: Data is embedded and retrieved via OpenAI’s vector store, with integration logic in utility modules and API endpoints.
* **Prompt System**: Prompt templates and logic are managed in both the prompts directory and utility modules, supporting anti-fabrication and canonical topic mapping.

## Key Innovations

1. **Two-Segment Rule**: Approach to prevent invalid cross-segmentation of demographic data
2. **Anti-Fabrication Measures**: Guidelines to prevent AI from generating fictional data
3. **Canonical Topic Mapping**: Dynamic mapping of user queries to canonical survey topics
4. **Selective Year-over-Year Comparison**: Rules-based approach to only compare methodologically compatible data points

## Project Objectives

RIA25 aims to achieve:

* Processing of 2025 Global Workforce Survey data 0- Comparison where possible with 2024 Global Workforce Survey data
* Implementation of data integrity safeguards
* Deployment with appropriate monitoring
* Comprehensive documentation and support resources

## Next Steps

1. User feedback collection and analysis
2. Potential enhancements based on usage patterns
3. Preparation for future survey data integration
4. Ongoing performance optimization

*This overview was last updated on April 13, 2025*

ewpage

# RIA25 Implementation Plan

## Overview

This document outlines the implemented architecture, development workflow, and technical decisions for the RIA25 system, reflecting the actual build rather than initial planning.

## System Components

### 1. Data Processing Pipeline

* **Implemented Solution**: Node.js scripts to process CSV data into structured JSON
* **Key Scripts**:
  + process\_survey\_data.js: Transforms raw CSV data into structured JSON files
* **Output Format**: JSON files split by question number with consistent metadata structure
* **Location**: scripts/ directory

### 2. Vector Store

* **Technology**: OpenAI-based vector database
* **Proposed Approach**: Use Assistants API with file uploads
* **Data Structure**: Individual question responses with associated metadata
* **Query Mechanism**: Semantic similarity search with context retrieval

### 3. Prompt Engineering

* **Approach**: Multi-layered system prompt with specific instructions for:
  + Segment detection and validation
  + Data accuracy confirmation
  + Anti-fabrication measures
  + Result formatting
* **Evolution**: Documented in RIA25\_Documentation/07\_prompt\_evolution.md

### 4. Web Interface

* **Framework**: Next.js
* **Components**:
  + User input interface with query submission
  + Response display with formatting
  + Session management
  + API integration
* **Deployment**: Vercel

### 5. API Integration

* **Implementation**: OpenAI Assistants API
* **Authentication**: API key management via environment variables
* **Error Handling**: Robust error handling with appropriate user feedback

## Implementation Workflow

### Phase 1: Data Processing

1. Parse raw CSV data using process\_survey\_data.js
2. Transform into structured JSON format with consistent metadata
3. Split data by question number for efficient retrieval
4. Validate accuracy of processed data

### Phase 2: Vector Store Setup

1. Configure vector store parameters
2. Develop tools to ingest processed JSON files into the vector store
3. Test retrieval accuracy and adjust as needed
4. Implement error handling for vector operations

### Phase 3: Prompt Engineering

1. Develop initial prompt structure
2. Integrate segment detection rules
3. Implement anti-fabrication measures
4. Test and refine prompt for accuracy
5. Document prompt evolution

### Phase 4: Web Interface Development

1. Create Next.js application structure
2. Implement API integration components
3. Design user interface
4. Add error handling and loading states
5. Test user experience

### Phase 5: Deployment

1. Configure Vercel deployment
2. Set up environment variables
3. Implement monitoring
4. Complete documentation
5. Perform final testing

## Technical Decisions

### Data Structure

The implemented system uses individual JSON files for each question, with:

* Consistent metadata structure
* Demographic information
* Response data
* Year information

This approach was chosen over alternatives (single large JSON file, database) for:

* Simplified vector store ingestion
* Reduced memory requirements
* Improved retrieval accuracy

### Segment Handling

The implementation enforces a two-segment maximum rule:

* Prevents invalid cross-segmentation
* Maintains statistical validity
* Ensures response accuracy

### API Integration

OpenAI Assistants API was selected over alternatives due to:

* Built-in vector search capabilities
* Simplified file management
* Reduced development complexity
* Strong performance characteristics

## Challenges and Solutions

| Challenge | Solution |
| --- | --- |
| CSV format inconsistencies | Robust column mapping implementation |
| Cross-segmentation validity | Two-segment rule enforcement |
| Response fabrication | Multi-layered verification prompts |
| Vector retrieval accuracy | Question-specific JSON files |
| Response formatting | Structured prompt instructions |

## Monitoring and Maintenance

* **Logging**: Error logging for API interactions

## Next Steps

1. User feedback collection
2. Performance optimization
3. Enhanced analytics features
4. Preparations for future survey data

*Last updated: April 5, 2024*

ewpage

# RIA25 Data Processing Workflow

**Last Updated:** April 30, 2024  
**Target Audience:** Developers, System Architects, Technical Stakeholders  
**Related Documents:**

* 06\_system\_architecture.md
* 15\_thread\_data\_management.md
* 07\_prompt\_evolution.md

## Overview

This document describes the data processing workflow implemented for RIA25, detailing the transformation of raw survey data into structured JSON files optimized for vector database ingestion, and the subsequent query processing and data retrieval systems.

## Workflow Diagram

Raw CSV Data → Data Validation → Column Mapping → JSON Transformation → Split by Question → Vector Store Ingestion

## Data Processing Steps

### 1. Input Data Format

* **Source**: Raw CSV files from the 2025 and 2024 Global Workforce Surveys
* **Format**: CSV with headers
* **Location**: /scripts/data/2025/2025\_global\_data.csv, /scripts/data/2024/2024\_global\_data.csv
* **Structure**:
  + Rows represent individual survey responses
  + Columns include question responses and demographic information

### 2. Data Validation & Harmonization

* **Purpose**: Ensure data integrity and harmonize structure across years before processing
* **Implementation**:
  + Initial data validation in process\_survey\_data.js
  + Advanced harmonization and validation in process\_2025\_data.js
  + Canonical topic mapping and normalization logic
* **Checks**:
  + CSV format verification
  + Required columns presence
  + Data type validation
  + Consistency and mapping to canonical topics
  + Harmonization of 2024/2025 data for valid comparison

### 3. Column Mapping

* **Purpose**: Create flexible mapping between CSV columns and structured JSON
* **Implementation**: Dynamic column mapping in process\_survey\_data.js and harmonization in process\_2025\_data.js
* **Benefits**:
  + Resilience to column order changes
  + Adaptation to CSV format variations
  + Simplified maintenance

### 4. JSON Transformation

* **Purpose**: Convert raw CSV data into structured JSON
* **Implementation**: Transformation logic in process\_survey\_data.js and process\_2025\_data.js
* **Structure**:
  + Consistent metadata format
  + Demographic information categorization
  + Response data formatting
  + Year information
  + Canonical topic mapping

### 5. Split by Question

* **Purpose**: Create individual JSON files per question for optimized retrieval
* **Implementation**: File creation in process\_survey\_data.js and process\_2025\_data.js
* **Output**:
  + Individual files named 2025\_[question\_number].json, 2024\_[question\_number].json
  + Located in scripts/output/split\_data/
  + Harmonized and global files in scripts/output/

### 6. Vector Store Integration

* **Purpose**: Prepare and upload data to vector database for semantic retrieval
* **Implementation**:
  + Data retrieval and embedding logic implemented in utils/openai/retrieval.js and API endpoints (e.g., app/api/retrieve-data/route.js)
  + Validation and coverage checks in utils/validation/data-validation.js
* **Process**:
  + Read processed JSON files
  + Format for vector embedding
  + Upload and retrieve via OpenAI Assistants API and utility modules
  + Verify successful ingestion and retrieval

## Example Data Transformation

### Input CSV Format

question\_1,question\_2,region,age\_group,gender,org\_size  
"Agree","Strongly Agree","North America","25-34","Male","1000-4999"

### Output JSON Format

{  
 "metadata": {  
 "survey\_year": 2025,  
 "question\_number": 1,  
 "question\_text": "I feel empowered to make decisions in my role."  
 },  
 "responses": {  
 "by\_region": {  
 "North America": {  
 "response\_count": 1500,  
 "response\_data": {  
 "Agree": 45,  
 "Strongly Agree": 30,  
 "Neutral": 15,  
 "Disagree": 7,  
 "Strongly Disagree": 3  
 }  
 }  
 // Additional regions...  
 },  
 "by\_age": {  
 // Age group breakdowns...  
 }  
 // Additional demographic breakdowns...  
 }  
}

## Key Scripts

### process\_survey\_data.js

* **Purpose**: Core data processing script for both 2024 and 2025 data
* **Location**: /scripts/process\_survey\_data.js
* **Functionality**:
  + CSV parsing
  + Data validation
  + Column mapping
  + JSON transformation
  + File generation

### process\_2025\_data.js

* **Purpose**: Advanced harmonization and canonical topic mapping for 2025 data
* **Location**: /scripts/process\_2025\_data.js
* **Functionality**:
  + Data harmonization across years
  + Canonical topic mapping
  + Output of harmonized and split files

### Vector Store Integration

* **Purpose**: Vector store integration for semantic retrieval
* **Implementation**:
  + Data retrieval and embedding logic in utils/openai/retrieval.js
  + API endpoints in app/api/retrieve-data/route.js, app/api/chat-assistant/route.ts
  + Validation in utils/validation/data-validation.js
* **Process**:
  + Read and process JSON files
  + Embed and retrieve via OpenAI Assistants API and utility modules
  + Validate and log results

## Query Processing System

The query processing system is designed to efficiently handle user queries, identify relevant data files, and retrieve only the necessary data segments. This system has been optimized to reduce latency and improve user experience.

### Query Flow Diagram

User Query → Query Intent Parsing → File Identification → Smart Filtering → Segment-Based Caching → Response Generation

### Query Intent Parsing

* **Purpose**: Extract meaningful intent from natural language queries
* **Implementation**: utils/data/smart\_filtering.js
* **Components**:
  + parseQueryIntent(query, conversationHistory): Extracts topics, demographics, years, and specificity from the query
  + mapIntentToDataScope(queryIntent): Maps the parsed intent to a data scope (topics, demographics, years, file IDs)
* **Features**:
  + Detection of follow-up questions through heuristics
  + Identification of demographic segments mentioned in the query
  + Topic extraction and mapping to canonical topics
  + Year detection for historical comparisons

### Enhanced Follow-up Detection

The system implements multi-level heuristics to identify follow-up queries:

1. **Short Query Check**: Brief queries (< 15 characters) are likely follow-ups
2. **Pronoun Check**: Queries starting with pronouns (it, this, they) indicate follow-ups
3. **Reference Check**: Terms like “previous,” “above,” or “mentioned” suggest follow-ups
4. **Comparative Check**: Words like “more,” “less,” or “better” often indicate follow-ups

For ambiguous cases, an optional lightweight model classification can be used:

async function classifyQueryWithModel(query, previousQuery) {  
 // Use a lightweight model to classify if this is a follow-up  
 // Returns boolean: true if this is a follow-up question  
}

### Optimized File Identification

* **Purpose**: Identify only the relevant data files for a given query
* **Implementation**: utils/openai/retrieval.js
* **Process**:
  + For starter questions: Use precompiled file IDs from JSON configuration
  + For follow-up queries: Reuse cached file IDs from previous interactions
  + For new queries: Use OpenAI to identify relevant file IDs
* **Performance Optimization**:
  + Skip file identification for clear follow-up queries
  + Cache file IDs by thread ID for future queries
  + Implement thread-level data caching

### Smart Data Filtering

The system implements intelligent data filtering to reduce payload size and processing time:

1. **Base Data Filter**

* function getBaseData(retrievedData, queryIntent) {  
   // Return only essential data for general queries  
   return {  
   summary: generateSummary(retrievedData),  
   topStats: extractTopStats(retrievedData),  
   };  
  }

1. **Specific Data Filter**

* function getSpecificData(retrievedData, { demographics = [] }) {  
   // Return detailed data filtered by demographics and other criteria  
   return {  
   filteredData: filterByDemographic(retrievedData, demographics),  
   detailedStats: generateDetailedStats(filteredData),  
   };  
  }

### API Flow Optimization

The system has been optimized to reduce API calls and improve response time:

1. **Streamlined API Route**
   * /api/chat-assistant/route.ts handles the entire data flow
   * Implements follow-up detection and cache utilization
   * Manages thread creation and message handling
   * Streams responses back to the client
2. **Two-Stage Processing**
   * Stage 1: Data retrieval and preprocessing (handled by /api/query)
   * Stage 2: Response generation with OpenAI Assistant (handled by /api/chat-assistant)
3. **Dynamic Data Granularity**
   * Processing varies based on query specificity
   * General queries receive summarized data
   * Specific queries receive detailed, filtered data

## Smart Filtering and Segment-Aware Caching

After data is processed and ingested, the system applies a segment-aware smart filtering and caching pipeline to ensure that queries are answered with the most relevant demographic or categorical breakdowns.

**Segment Selection Logic:**

* **Starter Questions:** If the query is a recognized starter question (e.g., “SQ2”), the relevant segments are explicitly set in the corresponding starter JSON file (e.g., "segments": ["sector"]).
* **All Other Queries:** For general queries, the OpenAI model is prompted (see utils/openai/1\_data\_retrieval.md) to infer the most relevant segments (e.g., “sector”, “age”, “region”, “gender”) and return them in a "segments" array in its JSON response.
* **Default Fallback:** If neither the starter JSON nor OpenAI inference provides segments, the system defaults to ["region", "age", "gender"] (see utils/data/segment\_keys.js).

**Canonical Segment Keys and Centralized Config:**

* All valid segment keys are defined in a single config file: utils/data/segment\_keys.js.
  + DEFAULT\_SEGMENTS is the fallback: ["region", "age", "gender"]
  + CANONICAL\_SEGMENTS is the full set used for filtering: ["overall", "region", "age", "gender", "org\_size", "sector", "job\_level", "relationship\_status", "education", "generation", "employment\_status"]
* All filtering, matching, and fallback logic references this config, ensuring consistency and maintainability.

**How Segments Are Used:**

* The selected segments are injected into the query intent and used throughout the filtering and caching pipeline (see utils/openai/retrieval.js and utils/data/smart\_filtering.js).
* Only canonical segment keys are used for filtering, guaranteeing alignment with the data files.
* The cache tracks which segments have been retrieved for each file, enabling efficient follow-up queries and minimizing redundant data loading.

**Enhanced Logging and Traceability:**

* Every step logs the query, files, topics, and segments being retrieved.
* If no stats are found for the selected segments, the system logs both the requested segments and the available segments in the data file, making it easy to debug any mismatch or data hygiene issue.
* This provides full transparency and traceability from query to assistant response.

**Rationale:**

* This hybrid approach allows for per-question customization (for starters), dynamic adaptation (for general queries), and robust fallback behavior.
* Centralizing segment logic in a config file ensures that all code paths use the same canonical keys, eliminating inconsistencies.
* Enhanced logging guarantees that any segment alignment issues can be quickly diagnosed and resolved.

## Incremental Data Loading

To optimize performance, the system implements incremental data loading, which only retrieves segments that aren’t already in the thread cache:

1. **Determine Required Segments**
   * Parse query intent to identify needed segments
   * Check which segments are already available in thread cache
   * Calculate missing segments that need to be loaded
2. **Load Missing Segments**
   * Retrieve only the segments that are missing from the cache
   * Load these segments from the appropriate data files
   * Merge new segments with existing cached data
3. **Update Thread Cache**
   * Add newly loaded segments to the thread cache
   * Track both loaded and available segments for each file
   * Use thread ID for cache key to maintain conversation context

function loadMissingSegments(cachedFiles, neededSegments) {  
 const missingSegments = {};  
  
 cachedFiles.forEach((file) => {  
 // Determine which segments are missing for this file  
 const missing = neededSegments.filter(  
 (seg) => !file.loadedSegments.has(seg) && file.availableSegments.has(seg)  
 );  
  
 if (missing.length > 0) {  
 missingSegments[file.id] = missing;  
 }  
 });  
  
 // Load missing segments from disk and update cache  
 // Return only the newly loaded data  
}

## Thread-Level Caching and Data Payload Minimization

The system implements a thread-level cache to optimize data retrieval and minimize payloads sent to the assistant. This cache tracks which datafiles and segment slices have already been sent for each thread, ensuring that only new or required data is transmitted for follow-on queries.

### Caching Workflow Scenarios

**1. Follow-on question (same thread), no additional data needed**

* The follow-on question can be answered using datafiles and segments already cached in the thread.
* The system recognizes that all required data is present in the cache.
* No new datafiles or segments are loaded or sent.
* The assistant uses the cached data to answer the question.

*Result: Zero additional data payload; response is generated from the existing in-memory cache.*

**2. Follow-on question (same thread), additional data needed (e.g., user requests job\_level breakdown)**

* The follow-on question requires a segment (e.g., job\_level) or datafile not previously loaded.
* The system compares the required datafiles/segments for the new query against the thread’s cache.
* Only the missing datafiles or segment slices are loaded and sent to the assistant.
* The cache is updated to include the new data/segments for future queries in the thread.

*Result: Only the incremental data (e.g., job\_level segment) is sent, minimizing payload. Future queries can use the expanded cache.*

**3. Follow-on, but a completely different question (same thread) requiring different datafiles**

* The new question requires datafiles and/or segments not present in the thread’s cache.
* The system identifies which datafiles/segments are missing.
* Only the new, required datafiles/segments are loaded and sent.
* The cache is updated to include these new datafiles/segments.

*Result: Payload includes only the new data needed for the new question; previously cached data remains available for further follow-ons.*

## Expected Performance

* **Processing**: The system is designed to efficiently process CSV data
* **Response Time**: Optimized for sub-10-second responses for most queries
* **Follow-up Queries**: Near-instantaneous response for simple follow-ups
* **Payload Size**: Minimized through segment-aware caching
* **Token Usage**: Reduced by sending only relevant segments
* **Output**: Multiple JSON files split by question number
* **Organization**: Files organized for optimized retrieval

## Error Handling Strategy

* **CSV Format Validation**: Detection of format inconsistencies
* **Data Validation**: Verification of required fields and data types
* **Segment Validation**: Checks if requested segments exist in data files
* **Cache Integrity**: Verification of cache structure and content
* **Logging**: Capturing processing information for troubleshooting

## Maintenance Procedures

### Adding New Survey Data

1. Place new CSV file in the appropriate directory
2. Update column mappings if format has changed
3. Run process\_survey\_data.js with appropriate parameters
4. Verify output JSON files
5. Process files for vector database integration when available

### Modifying Data Structure

1. Update transformation logic in process\_survey\_data.js
2. Test with sample data
3. Regenerate all JSON files
4. Update vector database with new format

### Monitoring Performance

1. Review performance logs to identify slow queries
2. Check cache hit/miss ratios for optimization opportunities
3. Monitor segment usage patterns to improve default segments
4. Adjust follow-up detection heuristics based on usage patterns

*Last updated: April 30, 2024*

ewpage

d to # Normalized Data Strategy and Canonical Topic Mapping

## Overview

The normalized data strategy represents a significant evolution from the 2024 approach, allowing for more intuitive querying and accurate cross-year comparisons. This document outlines how RIA25 organizes survey data around canonical topics rather than raw question numbers, enabling more effective data retrieval and analysis.

## Data Strategy Evolution

### 2024 Approach Limitations

In the 2024 implementation, several challenges were identified:

* **Inconsistent Structures**: The 2024 data used different keys and structures (e.g., “question”, “response”, “data” with subkeys like “region”), making direct year-over-year comparisons difficult.
* **Raw Question Mapping**: The system relied on raw question IDs (e.g., Q4\_9, Q4\_10), requiring constant reconciliation when questions were modified or reordered.
* **Formatting Inconsistencies**: Data was represented inconsistently across years (e.g., percentages as 0.64 vs “64%”).
* **User Experience Limitations**: End users had to understand internal question numbering rather than focusing on the actual survey topics.

### 2025 Normalized Approach

The 2025 implementation fundamentally changes the data organization:

* **Topic-Centric Structure**: Data is organized around canonical topics (e.g., “AI\_Attitudes”, “Work\_Life\_Flexibility”) rather than raw question numbers.
* **Standardized Schema**: A consistent data structure is used across years, with normalized market names, demographic keys, and percentage formats.
* **Human-Friendly Questions**: Each canonical topic is associated with a human-friendly “canonicalQuestion” that describes the essence of what’s being measured.
* **Comparable Markets**: The system explicitly defines which markets can be compared across years, preventing invalid comparisons.

## Canonical Topic Mapping Implementation

### canonical\_topic\_mapping.json

The core of this approach is the canonical\_topic\_mapping.json file located in scripts/reference files/. This file:

1. Defines metadata about the canonical topic structure
2. Establishes data access paths and rules
3. Lists all valid JSON files for each survey year
4. Organizes topics by themes (e.g., “Talent Attraction & Retention”)
5. Maps canonical topics to specific question files across years

### Structure of the Mapping File

{  
 "metadata": {  
 "version": "1.0",  
 "description": "Canonical topic mapping for survey data organized by themes and topics",  
 "lastUpdated": "2025-03-15"  
 },  
 "dataAccess": {  
 "basePath": "scripts/output/split\_data/",  
 "comparableMarkets": [  
 "United Kingdom",  
 "United States",  
 "Australia",  
 "India",  
 "Brazil"  
 ],  
 "allMarkets2025": [...]  
 },  
 "retrievalRules": {  
 "comparable": "...",  
 "nonComparable": "...",  
 "singleYear": "..."  
 },  
 "allValidFiles": {  
 "2024": [...],  
 "2025": [...]  
 },  
 "themes": [  
 {  
 "name": "Theme Name",  
 "topics": [  
 {  
 "id": "Canonical\_Topic\_ID",  
 "canonicalQuestion": "Human-readable question?",  
 "rationale": "Explanation of why this topic matters",  
 "mapping": {  
 "2024": [{"id": "Q1", "file": "2024\_1.json"}],  
 "2025": [{"id": "Q1", "file": "2025\_1.json"}]  
 },  
 "comparable": true/false,  
 "availableMarkets": [...],  
 "userMessage": "Message explaining comparison limitations",  
 "alternatePhrasings": [...]  
 }  
 ]  
 }  
 ]  
}

### Key Components

1. **Canonical Topics**: Each topic (e.g., “AI\_Attitudes”) represents a key area of inquiry that may span multiple raw questions and years.
2. **Theme Organization**: Topics are grouped into thematic categories (e.g., “Skills & Development”) for logical organization.
3. **Mapping to Raw Data**: Each canonical topic maps to specific question files in each survey year, making connections explicit.
4. **Comparability Flags**: The comparable property indicates whether direct year-over-year comparisons are valid.
5. **User Messages**: Each topic includes custom messages to explain any limitations in data comparability.
6. **Alternate Phrasings**: To improve retrieval, topics include alternative ways users might refer to the same concept.

## How RIA25 Uses Canonical Topic Mapping

### Query Processing

1. When a user submits a question, the system identifies relevant canonical topics by matching keywords and concepts.
2. The system uses the mapping file to locate the appropriate data files for each topic.
3. Based on the comparable flag, the system retrieves data from one or both years.
4. If limitations exist, the appropriate userMessage is included in the response.

### Data Retrieval Rules

The system follows specific rules based on the mapping:

* **Comparable Topics**: For topics marked as comparable, data from both 2024 and 2025 is retrieved, but only for the designated comparable markets.
* **Non-Comparable Topics**: For non-comparable topics, data from each year is presented separately with appropriate disclaimers.
* **Single-Year Topics**: For topics available in only one year, the system clearly indicates the limitation.

### Market Comparability

Only five markets are designated as comparable across years:

* United Kingdom
* United States
* Australia
* India
* Brazil

The 2025 survey includes additional markets that have no counterpart in 2024.

## Benefits of the Approach

1. **Intuitive Querying**: Users can focus on topics rather than question numbers.
2. **Cross-Year Integrity**: The system prevents invalid comparisons between years.
3. **Semantic Understanding**: The alternatePhrasings support better natural language understanding.
4. **Maintenance Efficiency**: New survey years can be added by updating the mapping file rather than modifying core code.
5. **Standardized Formats**: Consistent data representation improves reliability.

## Implementation Details

The canonical topic mapping is used throughout the RIA25 system:

1. **Data Processing**: The ETL pipeline normalizes raw CSV data into the structured format.
2. **Vector Store Organization**: Vector embeddings are organized to align with canonical topics.
3. **Query Response**: The assistant uses the mapping to retrieve and present data appropriately.
4. **Response Templates**: Standard response formats leverage the canonical questions and user messages.

## Conclusion

The normalized data strategy and canonical topic mapping represent a significant advancement in how RIA25 organizes and accesses survey data. By focusing on the meaning of questions rather than their numerical identifiers, the system delivers more intuitive, accurate, and maintainable insights.

*Last updated: April 5, 2024*

ewpage

# 2025 Global Workforce Survey: Questions and Response Options

This document lists all questions from the 2025 Global Workforce Survey along with their possible response options.

## Q1 - If you were to look for a new job, what would be the most important factors for you?

Response options:

* Salary and compensation
* Flexible work hours
* Remote work options
* Workplace location/commute
* Career advancement opportunities
* Learning and development opportunities
* Job security
* Organizational culture and values
* Employee benefits (healthcare, wellness, etc.)
* Work-life balance
* Manager/leadership quality
* Colleagues and team dynamics
* Organizational reputation
* Meaningful/purposeful work
* Industry/field of work
* Social responsibility (e.g., societal justice, community outreach) policies and practices
* Environmental sustainability policies and practices
* Use of advanced technologies (e.g., AI, automation)
* Visa sponsorship / immigration assistance from an employer
* Other

## Q2 - When considering the below options which are would be most likely to make you stay at your current company?

Response options:

* Employee benefits (e.g., healthcare, mental health and wellness services, Paid Time Off, complimentary food and services)
* Manager I trust
* Flexible work hours
* Employer’s respect for personal priorities outside of work
* Colleagues I work with
* Career advancement
* Commute to office
* Learning and development opportunities
* Reputation of the organization
* Workplace policy (i.e. in office, from home, hybrid)
* Organizational culture and values
* Organizational purpose and mission
* Social responsibility (e.g., societal justice, community outreach) policies and practices
* Inclusive policies and practices
* Environmental sustainability policies and practices
* Use of advanced technologies (e.g., AI, automation)
* Visa sponsorship / immigration assistance from an employer
* Other

## Q3 - When considering the below options, which are most likely to make you leave your current company?

Response options:

* Inadequate employee benefits (e.g., healthcare, mental health and wellness services, Paid Time Off, complimentary food and services)
* Employer’s respect for personal priorities outside of work
* Manager I don’t trust
* Colleagues I work with
* Lack of flexible work hours
* Commute to office
* Poor organizational culture and values
* Inadequate learning and development opportunities
* Workplace policy (i.e. in office, from home, hybrid)
* Reputation of the organization
* Organization not fulfilling purpose and mission
* Poor social responsibility (e.g., societal justice, community outreach) policies and practices
* Non-inclusive policies and practices
* Poor environmental sustainability policies and practices
* Use of advanced technologies (e.g., AI, automation)
* Visa sponsorship / immigration assistance from an employer
* Other

## Q4 - Please confirm the location of your current place of work, and also where this would ideally be for you?

Current place of work (response options):

* Full-time in office
* Full-time remote
* Hybrid work (blend of working from home and office)
* Unsure

Ideal place of work (response options):

* Full-time in office
* Full-time remote
* Hybrid work (blend of working from home and office)
* Unsure

## Q5 - To what extent do you agree with the following statements about AI in the workplace and skills.

Statements to rate:

* My organization offers a broad enough range of learning approaches [e.g. self-directed video/e-learning/podcasts/short modules/practical workshops] to suit my needs
* My organization encourages experimentation with new technologies
* I feel excited and positive about how emerging technologies like AI will change the way I work
* I think using AI in my role will bolster my value in the next three years
* When I’ve been asked to use AI to help with my job, I found it produced better results and/or improved efficiency
* I feel leaders in my organization understand AI
* I feel adequately trained to use AI tools
* I feel certain that my role will be replaced by AI/tech in the next three years

## Q6 - To what extent do you agree with the following statements with regards to your workplace.

Statements to rate:

* I feel my company provides a variety of ways to communicate and connect with colleagues
* I feel leaders in my company respect the needs of employees across generations
* I feel my company provides adequate support for employee mental well-being
* I have strong connections with my remote colleagues
* I would stay at a job if it paid me the salary I want, even if I hated the role
* I would stay at a job if it allowed me flexibility, even if I hated the role
* My organization has taken steps to reduce cultural gaps between different generations of workers
* I feel I work better with people my own age

## Q7 - To what extent do you agree with the following statements.

Statements to rate:

* The organization handles decisions about people with sensitivity and care
* I am worried about how to make my resume/ CV stand out in the competitive job market
* My organization’s leaders value people over profits
* I am concerned about a lack of jobs for my skillset
* I am open to relocating to another country to secure a job
* I am concerned that protectionism policies around immigration are affecting my ability to find work
* I expect a significant reduction in DEI initiatives at my company since new governments were elected
* I feel my company is using Return to Office policies and other rules to push me out

## Q8 - To what extent do you agree with the following statements.

Statements to rate:

* Leaders at my organization are often not aligned
* I feel recent changes in senior leadership has negatively impacted the operations at our firm
* I am stretched too far beyond my capabilities at work
* I feel that I am overlooked for leadership roles because of my age
* I frequently doubt my ability to fulfill my work responsibilities
* I feel that I am overlooked for training because of my age
* I feel that I am overlooked for promotions, pay rises, or leadership training because of my class
* I feel that I am overlooked for promotions, pay rises, or leadership training because of my sex
* I feel that I am overlooked for promotions, pay rises, or leadership training because of my race
* I struggle with imposter syndrome
* I share negative reviews about my employer on social media or review sites (such as Glassdoor)

## Q9 - To what extent do you agree with the following statements.

Statements to rate:

* My manager empowers me
* I feel that my ideas and opinions will be welcomed by company leaders
* If a problem in my personal life is affecting my work/performance, I feel comfortable telling my manager about it
* Our external reputation matches our internal culture
* I’m comfortable discussing my personal life with my colleagues
* My manager appears overwhelmed at work
* My organization has cut back on the number of managers
* The lack of managers at my organization leaves me feeling directionless

## Q10 - What would your ideal role be?

Response options:

* Permanent Full Time
* Part time
* Contract / Interim Part Time
* Contract / Interim Full Time
* Freelance Part Time
* Freelance Full Time

## Q11 - How well do you think you are compensated for your skillset?

Response options:

* My company gives me a salary and benefits that are below the value of my skills
* My company gives me a salary and benefits that match the value of my skills
* My company gives me a salary and benefits that exceed the value of my skills

## Q12 - Are you planning on leaving your current role in the next three months?

Response options:

* No, not interested in moving
* I’m considering it and looking at options
* I’m currently applying and/or interviewing
* I’m preparing to retire

## Q13 - What would your ideal role be?

Response options:

* Permanent Full Time
* Part time
* Contract / Interim Part Time
* Contract / Interim Full Time
* Freelance Part Time
* Freelance Full Time

## Q14 - What are the biggest challenges you face when working with colleagues from different generations?

Response options:

* Different communication styles
* Gaps in technology skills
* Conflicting values or priorities
* Leadership bias towards specific age groups
* I don’t experience any challenges

## Q15 - Which of the following would help you collaborate better with colleagues from different generations?

Response options:

* Training on communication and teamwork
* A stronger focus on shared values and goals
* Technology training for bridging skill gaps
* Reverse mentorship programmes
* Training on how to work remotely with colleagues
* None of these

## Q16 - Do you feel your company allows you to switch off at the end of the day without an expectation to answer emails or calls?

Response options:

* Yes, there is no expectation to answer emails or calls after I log off
* Somewhat, I still have to answer urgent emails or calls on a case-by-case basis
* Not at all, my company expects me to always be available and responsive

## Q17 - To what extent do you agree with the following statements.

Statements to rate:

* My job makes good use of my skills and abilities
* My job provides opportunities to do challenging and interesting work
* I have good opportunities for learning and development at the company
* The organization motivates me to do my best work
* The organization is strategically adapting to changes in the business environment
* I feel motivated to do more than is required of me
* I have trust and confidence in the company’s senior leadership team
* The organization shows care and concern for its employees

For rating scales (Q5-Q9, Q17), respondents appear to indicate their level of agreement, with responses categorized as “Agree” or “Disagree” in the data.

ewpage

# RIA25 System Architecture

**Last Updated:** April 18, 2024  
**Target Audience:** Developers, System Architects, Technical Stakeholders  
**Related Documents:**

* 03\_data\_processing\_workflow.md
* 15\_thread\_data\_management.md
* core\_files.md

## 1. System Overview

RIA25 (Research Insights Assistant 2025) is an AI-powered analytics assistant designed to provide insights from workforce survey data. The system maintains conversation context across interactions, efficiently caches relevant data, and intelligently handles both follow-up and new queries within ongoing conversations.

## 2. Architecture Diagram

**Note:** The following diagram is written in [Mermaid](https://mermaid-js.github.io/mermaid/#/) syntax.

* On GitHub, GitLab, Obsidian, Notion, and many documentation tools, this will render as a visual diagram.
* In VSCode, install the “Markdown Preview Mermaid Support” extension to see the diagram in the Markdown preview.
* If you need a static image (PNG/SVG) version of this diagram embedded in the .md, let the development team know and it can be generated and inserted.

graph TD  
 UI[Web Interface<br/>(Next.js, app/, components/)]  
 API[API Layer<br/>(app/api/chat-assistant, retrieve-data, etc.)]  
 UTILS[Utility Modules<br/>(utils/openai/retrieval, validation, cache)]  
 SCRIPTS[Data Processing<br/>(scripts/process\_survey\_data, process\_2025\_data)]  
 DATA[Data Files<br/>(scripts/output/, split\_data/)]  
 PROMPTS[Prompts<br/>(prompts/, config/)]  
 CACHE[Thread Cache<br/>(cache/)]  
  
 UI --> API  
 API --> UTILS  
 API --> DATA  
 UTILS --> DATA  
 SCRIPTS --> DATA  
 API --> PROMPTS  
 UI --> PROMPTS  
 API --> CACHE  
 UTILS --> CACHE

## 3. Core Components

### 3.1 Next.js Web Interface

* **Purpose**: Provides user-facing application for querying and displaying survey insights
* **Key Features**:
  + User query input
  + Response rendering with Markdown support
  + Thread and session management
  + Error handling and loading states
  + Persistent thread tracking via localStorage
* **Key Files**:
  + app/layout.tsx - Root layout
  + app/page.tsx - Main landing page
  + components/MainComponent.js - Central UI component
  + components/PromptInput.tsx - User prompt input component

### 3.2 API Layer (app/api/)

* **Purpose**: Handles user queries, data retrieval, and orchestration of analysis and validation
* **Key Endpoints**:
  + chat-assistant/route.ts: Main endpoint for handling user queries, integrating with OpenAI, and orchestrating retrieval/analysis
  + retrieve-data/route.js: Endpoint for direct data retrieval and analysis
  + Other endpoints: For logging, configuration, and auxiliary functions
* **Implementation Details**:
  + Integrates with utility modules for retrieval, validation, and caching
  + Handles logging and performance metrics
  + Maintains thread continuity and context

### 3.3 Utility Modules

* **Purpose**: Provide core logic for data retrieval, validation, caching, and prompt handling
* **Key Files**:
  + utils/openai/retrieval.js: File identification, analysis generation, prompt handling
  + utils/validation/data-validation.js: Data validation and coverage checks
  + utils/cache-utils.ts: Thread and file cache management
  + utils/helpers.tsx: Performance metrics and common helper functions
  + utils/data/smart\_filtering.js: Filtering loaded data based on segments

### 3.4 File Access Modes

RIA25 supports two different file access modes:

#### 3.4.1 Standard Mode (Default)

* **Process**:
  1. System identifies relevant files with identifyRelevantFiles
  2. System retrieves and processes the actual data files
  3. System generates analysis of the data
  4. Pre-processed analysis is sent to the assistant
  5. Assistant responds based on the analysis
* **Advantages**:
  + Full control over data processing
  + Consistent data presentation to the assistant
  + Optimized for specific data formats

#### 3.4.2 Direct Access Mode

* **Process**:
  1. System identifies relevant file IDs using identifyRelevantFiles
  2. System sends only the prompt and file IDs to the assistant
  3. Assistant retrieves and processes data directly
  4. Assistant generates its own analysis and response
* **Advantages**:
  + Simplified backend processing
  + More flexibility for the assistant
  + Reduced processing overhead

#### 3.4.3 Mode Selection

* **Configuration Options**:
  + Environment variable: FILE\_ACCESS\_MODE=direct|standard
  + Query parameter support: ?accessMode=direct|standard
  + Assistant-specific configuration in mapping file

### 3.5 Data Processing Pipeline

* **Purpose**: Transforms raw survey data into structured format for analysis
* **Key Features**:
  + CSV parsing and validation
  + Data normalization and harmonization (2024/2025)
  + Canonical topic mapping
  + JSON transformation
  + File generation (split, harmonized, global)
* **Implementation**:
  + Node.js scripts: process\_survey\_data.js, process\_2025\_data.js
  + Output to scripts/output/ and scripts/output/split\_data/

### 3.6 Prompt System

* **Purpose**: Provides instructions and context for various system operations
* **Prompt Types**:
  + **Data Identification Prompt**: Maps user query to relevant data files
  + **Analysis Prompt**: Analyzes retrieved data for insights
  + **Response Generation Prompt**: Creates final user-facing response
* **Implementation**:
  + External files in prompts/ and utils/openai/
  + Version-controlled templates
  + Configurable instructions

## 4. Thread Management System

### 4.1 Thread Persistence

* **Thread Creation**:
  + New thread created for first interaction
  + Thread ID persisted in localStorage
  + Thread state retrievable after page refresh/session end
* **Thread State**:
  + Maintains full conversation history
  + Tracks which data files have been accessed
  + Records user queries and system responses

### 4.2 Thread-Specific Caching

* **Cache Structure**:
  + Thread-specific file system cache in cache/ directory
  + JSON files named by thread ID
  + Tracks loaded files and segments
* **Cache Operations**:
  + File-based caching for thread continuity
  + De-duplication of file IDs
  + Progressive enhancement as conversation evolves

### 4.3 Thread Intelligence

* **Context Awareness**:
  + System understands relationships between queries in same thread
  + Detects topic shifts requiring new data vs. follow-ups
  + Maintains conceptual links between related questions
* **Query Classification**:
  + New topic queries - Full data retrieval with cache initialization
  + Follow-up queries - Re-use cached data, focus on continuity
  + Topic shift queries - Partial cache use, add new files as needed
  + Content transformation queries - Re-use cached data with transformation instructions

## 5. Data Flow

### 5.1 Query Processing Pipeline

1. **Query Reception**
   * System receives natural language query from user interface
   * Query is associated with current thread ID or creates new thread
   * System timestamps and logs query for audit/improvement
2. **Data Relevance Analysis**
   * Query is analyzed to identify relevant data files and topics
   * Analysis uses pre-defined canonical mapping of topics to files
   * Out-of-scope detection prevents responses to irrelevant queries
3. **Efficient Data Retrieval**
   * System loads only data files relevant to the current query
   * Caches data by thread ID to enable efficient follow-up handling
   * Incremental data loading adds new files as conversation expands in scope
4. **Response Generation**
   * Raw data and analysis are injected into assistant prompt
   * Response is generated using conversation history and data context
   * Response is streamed back to user for immediate feedback

### 5.2 Core Functionality Flow

1. **Query Parsing**
   * Extracts user intent, keywords, and context markers
   * Checks for out-of-scope conditions
   * Identifies follow-up queries
2. **File Retrieval Based on Query**
   * Determines needed data files using canonical mapping
   * Accesses files from local storage or via API
   * Returns relevant file IDs and data
3. **Prompt Construction and Data Injection**
   * Combines user query, data, and instructions
   * Creates prompt for assistant processing
   * Injects relevant context from thread history
4. **Response Processing**
   * Assistant generates response from prompt and data
   * Response is validated and formatted
   * Result is streamed to user interface

## 6. Deployment Architecture

### 6.1 Production Environment

* **Web Hosting**: Vercel platform
* **API Access**: Direct connection to OpenAI services
* **Environment Variables**: Securely stored in Vercel
* **Domain**: Custom domain with SSL
* **Cache Strategy**: Local file-based cache with future Vercel KV integration

### 6.2 Development Environment

* **Local Development**: Next.js development server
* **API Integration**: Same OpenAI services with development keys
* **Environment Variables**: Local .env file
* **Cache Strategy**: Local file-based cache

## 7. Performance Optimization

### 7.1 OpenAI API Optimization

* Reduced polling interval from 1000ms to 250ms
* Improved error handling and timeout management
* Enhanced performance tracking and timing metrics
* Optimized prompt structure and token usage

### 7.2 Thread and Data Caching

* Thread-specific caching for efficient follow-up handling
* Segment-aware data retrieval to minimize data loading
* Incremental retrieval of only missing data for follow-ups
* In-memory caching with file system persistence

### 7.3 Starter Question Optimization

* Pre-compiled starter question data in JSON format
* Fast-path processing for recognized starter questions
* Direct prompt construction from pre-compiled data
* Segment-specific data loading

## 8. Security Considerations

* **API Keys**: Secure storage in environment variables
* **User Data**: No user data persistence beyond session requirements
* **Rate Limiting**: Implementation for stability and cost management
* **Error Handling**: Secure error messaging without system details exposure

## 9. Monitoring and Logging

* **API Interactions**: Logging of API requests and responses
* **Error Tracking**: Error logging in API endpoints and utility modules
* **Performance Metrics**: Performance timing and logging
* **Usage Statistics**: Query volume and patterns

## 10. Future Enhancements

### 10.1 Cache Storage Enhancement

* Planned migration from file-based caching to Vercel KV for production
* Redis-compatible key-value store integration
* Environment-aware cache implementation
* Improved cache persistence and thread continuity

### 10.2 Advanced Query Classification

* ML-based classification of query types
* Enhanced follow-up detection
* Better topic shift handling
* Multi-topic query optimization

*Last updated: April 18, 2024*

ewpage

# RIA25 Prompt System Evolution

## Introduction

This document outlines the development journey of the RIA25 prompt system, tracking its evolution from initial test questions to the final sophisticated system prompt that powers the workforce insights assistant. The development process took place primarily in March 2024, with refinements continuing into April.

## Timeline of Development

### Initial Testing Phase (March 22, 2024)

The development process began with test\_questions.md, which established a foundation for testing the system’s capabilities with various query types:

* **Individual file testing questions**: Focused on detailed demographics, cross-segment analysis, outlier identification, and conditional analysis
* **Cross-file comparative questions**: Testing relationships between attraction and retention factors, consistency checks, and generational analysis
* **Complex analytical questions**: Exploring contradictory trends, sector-specific deep dives, and integrated gender analysis

These test questions were designed to verify if the system could correctly:

* Access the canonical mapping
* Apply default 2025 behavior
* Handle comparable vs. non-comparable topics
* Process basic theme/topic questions
* Manage demographic detail questions
* Respond to ambiguous queries

### Query Parsing Development (March 24, 2024)

The development then focused on establishing a robust query parsing system:

1. **query\_parser.md (01:59 PM)**: The first version of the query parser defined:
   * Critical requirements (single message responses, no fabrication)
   * A structured workflow (parse query, validate against canonical mapping, retrieve data, format response)
   * Data structure examples and verification steps
   * Clear guidelines on correct vs. incorrect responses
2. **new\_parser.md (03:51 PM)**: Expanded the parser with:
   * More detailed stages (connection to canonical file, parse and map query, answer delivery)
   * Explicit error handling
   * Required reporting sections
   * Clear display format for parsing reports
3. **query\_parser\_config.json (03:57 PM)**: Formalized the parser configuration in JSON format
4. **parser\_spec.md (04:02 PM)**: Created a comprehensive specification document that outlined:
   * Purpose and goals (process natural language queries, map to canonical topics)
   * Core principles (zero hard-coding, transparency, data integrity)
   * System architecture (component structure, data flow, dependencies)
   * Functional requirements
   * Technical specifications for each processing stage
   * Output formats and critical rules
5. **new\_parser.json (09:24 PM)**: Finalized the parser configuration in a structured JSON format

### Segment Detection Refinement (March 24, 2024)

A critical development was the implementation of segment detection in test\_segment\_detection.json (09:52 PM), which addressed:

* Detecting combinations of demographic segments (countries, job levels, age groups, genders, sectors)
* Identifying semantic patterns suggesting cross-segment analysis
* Implementing a strict response protocol for segment violations
* Ensuring segments are analyzed separately with appropriate disclaimers

This represented a significant enhancement to prevent invalid cross-segment analysis that wasn’t supported by the underlying data structure.

### Anti-Fabrication Testing (March 24, 2024)

Multiple iterations of anti-fabrication tests were developed at 08:49 PM:

* **prompt\_final\_no\_fabrication\_test1.md**: Established comprehensive guidelines for:
  + Vector store data retrieval
  + Topic identification
  + Query management
  + File retrieval and verification
  + Segment handling rules (global data only)
  + Data processing
  + Narrative-driven response structure
  + Strict anti-fabrication rules
  + Presentation guidelines
  + Error handling
  + Response quality checklist
  + Tone of voice guidelines
* **prompt\_final\_no\_fabrication\_test2.md through test4.md**: Successive refinements to the anti-fabrication approach, each building on the previous version

### System Prompt Finalization (March 24 - April 4, 2024)

1. **system\_prompt.md (05:16 PM, March 24)**: Initial comprehensive system prompt
2. **system\_prompt.json (04:16 PM, April 4)**: Final refined system prompt in JSON format, featuring:
   * Critical data access instructions
   * Mandatory reporting structures
   * Enhanced query processing with automatic segment detection
   * Comprehensive data verification processes
   * The Two Segment Rule enforcement
   * Narrative guidelines
   * DEI handling protocols
   * Clear scope boundaries
   * Response quality standards
   * Critical rules
   * Data fabrication prevention measures
   * Two segment verification protocols

## Key Evolutionary Themes

Throughout the development process, several key themes emerged:

1. **Increasing Formalization**: From markdown documents to structured JSON configurations
2. **Enhanced Data Integrity**: Growing focus on preventing fabrication and ensuring data retrieval
3. **Segmentation Enforcement**: Development of robust mechanisms to prevent invalid cross-segment analysis
4. **Query Processing Refinement**: Evolution from basic parsing to sophisticated intent and topic mapping
5. **Response Quality Standards**: Increasingly detailed guidelines for comprehensive, narrative-driven responses
6. **Transparency**: Implementation of detailed reporting while maintaining a seamless user experience

## Conclusion

The RIA25 prompt system underwent significant evolution from initial testing concepts to a sophisticated, rule-based system with strict data integrity enforcement. The final system prompt (system\_prompt.json) represents the culmination of this development process, incorporating lessons learned through multiple testing iterations and refinements.

The progression shows a deliberate path from initial concept testing to increasingly formalized structures, with particular emphasis on data integrity, segment handling, and high-quality response generation. The final system encapsulates these developments into a comprehensive JSON configuration that powers the workforce insights assistant.

## Current Implementation (2025)

* **Prompt Templates**: The system uses prompt templates and configurations stored in the prompts/ directory (e.g., system\_prompt.md, system\_prompt.json, starter\_prompt\_template.md).
* **Integration in Codebase**: Prompt selection, anti-fabrication logic, and canonical topic mapping are programmatically enforced in utils/openai/retrieval.js and orchestrated by API endpoints (app/api/chat-assistant/route.ts).
* **Canonical Topic Mapping**: The prompt system leverages canonical topic mapping to ensure queries are mapped to valid survey topics, with harmonization logic for year-over-year comparison.
* **Anti-Fabrication Enforcement**: The anti-fabrication and two-segment rules are enforced both in prompt templates and in the logic of utility modules, ensuring that responses are always grounded in real data and system constraints.
* **Dynamic Prompting**: The system dynamically selects and customizes prompts based on the query, topic, and data context, supporting robust, high-integrity responses.

This implementation ensures that the prompt system is not only a set of static templates but an integrated, programmatically enforced part of the RIA25 architecture.

## Starter Question System

### Overview

To optimize performance and user experience, RIA25 implements a sophisticated starter question system that allows for predefined, high-performing queries with optimized data retrieval paths. This system was implemented in April 2024 to address latency concerns and provide consistent, high-quality responses for common workforce queries.

### Precompiled Starter Questions

The system supports predefined starter questions (e.g., SQ1, SQ2) through a structured approach:

* **File Structure**:
  + Precompiled data files stored in /utils/openai/precompiled\_starters/ directory
  + Each starter question has a corresponding JSON file (e.g., SQ1.json, SQ2.json)
  + Optional specialized prompt template in /prompts/starter\_prompt\_template.md
* **JSON Format Example**:
* {  
   "starterQuestionCode": "SQ1",  
   "question": "What factors influence employee retention?",  
   "data\_files": ["2025\_1\_2", "2025\_3\_4"],  
   "segments": ["region", "age"],  
   "matched\_topics": ["Remote\_Work", "Work\_Flexibility"],  
   "summary": "In 2025, the key factors influencing employee retention are..."  
  }

### Two-Stage Query Processing

When a starter question is detected:

1. **Stage 1: Optimized Data Retrieval**
   * The system checks if the query matches a known starter question pattern
   * If matched, it loads the precompiled data directly from the corresponding JSON file
   * The precompiled data includes the exact file IDs, segments to use, and a natural language question
   * This stage completely bypasses the expensive LLM-based file identification process
2. **Stage 2: Response Generation**
   * The natural language question and prefiltered data are sent directly to the OpenAI Assistant
   * The assistant uses this optimized context to generate a response
   * This approach ensures consistency and reduces token usage

### Implementation Details

The starter question system is implemented across several key files:

1. **Frontend (app/embed/[assistantId]/page.tsx)**:
   * Detects ?starterQuestion=SQ2 URL parameter
   * Implements a two-stage fetch process:
     + First fetch to /api/query for data retrieval and optimization
     + Second fetch to /api/chat-assistant for response generation
   * Provides accurate loading feedback during each stage
2. **Backend (utils/openai/retrieval.js)**:
   * Contains isStarterQuestion(query) function to detect starter question patterns
   * Implements getPrecompiledStarterData(code) to load the appropriate JSON file
   * Completely bypasses the LLM file identification step for starter questions
   * Returns consistent data structure for both starter and standard questions
3. **API Endpoints**:
   * /api/query/route.js: Handles data retrieval with special logic for starter questions
   * /api/chat-assistant/route.ts: Processes the optimized data and generates responses

### Performance Benefits

The starter question system provides significant performance improvements:

* **Reduced Latency**: Bypassing the LLM file identification step saves 5-10 seconds
* **Consistent Responses**: Precompiled data ensures high-quality, consistent answers
* **Optimized Segment Selection**: Each starter question includes explicitly defined segments most relevant to the query
* **Lower Token Usage**: Precompiled summaries and filtered data reduce token consumption
* **Better User Experience**: Faster responses and consistent quality for common queries

### Extensibility

The starter question system is designed to be easily extended:

* New starter questions can be added by creating additional JSON files
* Existing starter questions can be updated to reflect new data or insights
* The system supports both complete precompiled responses and data-driven responses
* An optional script /scripts/generate\_precompiled\_starters.js can be used to batch-generate or update starter data files

This approach provides an optimal balance between performance, consistency, and flexibility, particularly for frequently asked questions about workforce trends.

*Last updated: April 30, 2024*

ewpage

# RIA25 Vector Store Reference

This document lists all files uploaded to the vector store for the Research Insights Assistant 2025.

## Vector Store Details

* **ID:** vs\_lMoDck4HDODRImvJIz1jVJ2A
* **Expiration Policy:** Never

## Uploaded Files

The vector store contains two primary categories of files:

### 1. Split Data Files

All JSON files from the scripts/output/split\_data directory are uploaded to the vector store. These include:

* Individual question files for 2024 (e.g., 2024\_1.json, 2024\_2.json, etc.)
* Individual question files for 2025 (e.g., 2025\_1.json, 2025\_2.json, etc.)
* File index information (2024\_file\_index.json, 2025\_file\_index.json)
* Consolidated data file (global\_2025\_data.json)

In total, there are approximately:

* 40 files for 2024 data
* 65 files for 2025 data

### 2. Reference Files

All configuration files from the scripts/reference files/2025 directory:

| File | Purpose |
| --- | --- |
| canonical\_topic\_mapping.json | Maps questions to canonical topics and themes |
| topics\_to\_avoid.json | Topics outside the scope of the survey |
| narrative\_guidelines.json | Guidelines for response structure |
| DEI\_Response\_Guidelines.json | Special handling for DEI-related topics |
| supported\_topics.json | Topics directly addressed by the survey |
| Radically\_Human\_Tone\_of\_Voice.json | Guidelines for response style and tone |

## File Organization

### Split Data Structure

The split data files follow a consistent naming convention:

* YYYY\_N.json for top-level questions
* YYYY\_N\_M.json for sub-questions

Each file contains:

* Question text
* Response options
* Data broken down by all demographic segments
* Percentage values

### Canonical Topic Structure

The canonical\_topic\_mapping.json file serves as the backbone of the system, mapping raw question files to meaningful topics organized by themes. It defines:

* Which topics exist in the survey data
* Which files contain data for each topic
* Which topics are comparable across years
* Which markets are valid for comparisons
* What user messages to display for limitations

## Vector Store Usage

When querying the vector store:

1. The assistant identifies relevant canonical topics based on the user query
2. It retrieves the corresponding split data files
3. For year-over-year comparisons, it follows the rules in the canonical mapping
4. Reference files provide guidelines for tone, structure, and boundaries

## System Integration

* **Integration Logic**: Vector store access, embedding, and retrieval are orchestrated by utility modules (utils/openai/retrieval.js) and API endpoints (app/api/chat-assistant/route.ts, app/api/retrieve-data/route.js).
* **Validation**: Data validation and coverage checks are performed by utils/validation/data-validation.js to ensure integrity and completeness of responses.
* **Process**:
  1. User query is received by the API endpoint.
  2. Utility modules identify relevant canonical topics and data files.
  3. Data is retrieved from the vector store and validated.
  4. Results are formatted and returned to the user.
* **File Structure**: The system expects split data files and reference files to follow the conventions described above for seamless integration.

## Maintenance

When updating the vector store:

1. New split data files should be generated through the data processing pipeline
2. The canonical topic mapping should be updated to include new questions
3. All files should be uploaded maintaining the same structure
4. The vector store ID should be updated in the system configuration

*Last updated: April 13, 2025*

ewpage

# RIA25 Development Timeline

## Overview

This document chronicles the development of RIA25, capturing key milestones, challenges, and decisions throughout the project lifecycle from planning to completion.

## Development Approach

### Planning Phase

#### Requirements Gathering

* Initial concept development
* Requirements analysis
* Survey data structure evaluation

#### System Design

* Architecture planning
* Technology selection
* Data flow mapping

### Development Phase

#### Data Processing Infrastructure

* Data processing scripts implementation (process\_survey\_data.js)
* CSV parser implementation
* Data transformation logic development
* **Challenge**: CSV format inconsistencies
* **Solution**: Dynamic column mapping system

#### Data Structure Refinement

* JSON structure finalization
* Question splitting implementation
* Metadata consistency enforcement
* **Decision**: Per-question JSON files approach

#### Week 3: Vector Store Implementation

* Vector store configuration
* Data embedding process implementation
* Retrieval testing
* **Challenge**: Large file embedding inefficiencies
* **Solution**: Split files by question for optimized embedding and retrieval

#### Prompt Engineering

* System prompt development
* Segment detection rules implementation
* Anti-fabrication measures testing
* Response formatting guidelines
* **Decision**: Implemented two-segment rule for demographic cross-sectioning

### Testing Considerations

#### System Testing

* End-to-end system testing approach
* Edge case identification
* **Challenge**: Preventing response fabrication
* **Approach**: Multi-layered verification in prompts

#### Prompt Refinement

* Iterative prompt improvements process
* Accuracy validation methods
* Format consistency testing approaches

### Deployment Considerations

#### Production Deployment

* Vercel configuration
* Environment variable setup
* Monitoring implementation
* **Consideration**: API rate limiting
* **Approach**: Client-side caching strategy

#### Documentation

* System documentation
* Maintenance procedures
* User guides

## Key Technical Decisions

### Data Structure Design

| Decision | Rationale |
| --- | --- |
| Split JSON by question | Improved vector retrieval performance |
| Standardized metadata format | Enhanced consistency in data representation |
| Dynamic column mapping | Provided resilience to CSV format changes |

### Architecture Design

| Decision | Rationale |
| --- | --- |
| Next.js framework | Developer familiarity and deployment simplicity |
| OpenAI Assistants API | Integrated vector storage and retrieval capabilities |
| Direct API integration | Simplified architecture and reduced complexity |

### Prompt Engineering Approach

| Decision | Rationale |
| --- | --- |
| Two-segment rule | Prevent invalid demographic cross-sectioning |
| Anti-fabrication measures | Ensure response accuracy and data fidelity |
| Response formatting guidelines | Create consistent and readable outputs |

## Challenges and Solutions

### Technical Challenges

| Challenge | Solution | Impact |
| --- | --- | --- |
| CSV format inconsistencies | Dynamic column mapping | Resilient data processing regardless of format changes |
| Cross-segmentation validity | Two-segment rule enforcement | Prevented statistically invalid demographic combinations |
| Response fabrication | Multi-layered verification prompts | Significant reduction in AI hallucinations |
| Vector retrieval accuracy | Question-specific JSON files | Improved relevance of retrieved context |

### Process Considerations

| Challenge | Solution | Impact |
| --- | --- | --- |
| Changing requirements | Agile development approach | Flexibility to adapt to evolving needs |
| Integration complexity | Simplified architecture | Reduced development and maintenance overhead |
| Testing thoroughness | Scenario-based testing | Comprehensive validation of system capabilities |

## Lessons Learned

1. **Data Structure Planning**: Early investment in robust data structure design pays dividends throughout development
2. **Prompt Engineering Iteration**: Prompt development requires multiple iterations with rigorous testing
3. **Vector Store Optimization**: File structure significantly impacts vector search performance
4. **System Simplification**: Keeping the architecture streamlined improves development velocity and reduces maintenance
5. **Documentation Discipline**: Ongoing documentation throughout development creates more accurate and useful references

## Current Implementation Status

This section provides a comprehensive assessment of the current implementation against the RIA system specification, identifying strengths and gaps.

### 1. Query Processing Pipeline

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| Query Reception | ✅ Fully Implemented | The system properly receives queries and associates them with threads |
| Data Relevance Analysis | ✅ Fully Implemented | Uses identifyRelevantFiles() with canonical mapping |
| Efficient Data Retrieval | ⚠️ Partially Implemented | Basic caching works but lacks some optimizations |
| Response Generation | ✅ Fully Implemented | Follows the proper prompt construction and streaming |

**Key Gaps:**

* The current system lacks advanced incremental data loading strategies
* Cache invalidation for data updates isn’t implemented

### 2. Prompt Engineering Architecture

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| Data Identification Prompt | ✅ Fully Implemented | 1\_data\_retrieval.md serves this purpose well |
| Analysis Prompt | ⚠️ Partially Implemented | Analysis is performed but not via a dedicated prompt file |
| Response Generation Prompt | ⚠️ Partially Implemented | Construction happens inline in code rather than from template |
| Prompt Versioning | ❌ Not Implemented | No versioning or A/B testing infrastructure |

**Key Gaps:**

* Prompts aren’t fully externalized into template files
* No version control system for prompts
* Missing A/B testing infrastructure

### 3. Thread Management System

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| Thread Creation | ✅ Fully Implemented | Proper creation and localStorage persistence |
| Thread State | ⚠️ Partially Implemented | Caches file IDs but not full conversation context |
| Context Awareness | ⚠️ Partially Implemented | Basic follow-up detection without advanced topic shift handling |
| Thread Lifecycle | ❌ Not Implemented | No automatic archiving or data export |

**Key Gaps:**

* Limited thread intelligence for context awareness
* No thread lifecycle management beyond manual reset
* Missing thread data exportability

### 4. Data File Management

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| File Registry | ⚠️ Partially Implemented | Maps exist in canonical mapping but no central registry with metadata |
| File Identification Logic | ✅ Fully Implemented | Good mapping from topics to file IDs |
| Thread-Specific Cache | ✅ Fully Implemented | Strong structure with localStorage persistence |
| Intelligent Cache Management | ⚠️ Partially Implemented | Handles duplication but not invalidation or size limits |
| Cache Access Patterns | ⚠️ Partially Implemented | Simple classification without partial cache usage |

**Key Gaps:**

* No central file registry with comprehensive metadata
* Missing cache size limits and LRU eviction
* No cache invalidation for data updates

### 5. Query Classification and Handling

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| New Topic Queries | ✅ Fully Implemented | Properly handles new topics |
| Follow-up Queries | ⚠️ Partially Implemented | Simple heuristics without advanced classification |
| Topic Shift Queries | ❌ Not Implemented | No specific detection for topic shifts requiring partial cache |
| Content Transformation | ⚠️ Partially Implemented | Basic detection added to data-validation.js |
| Heuristic Rules | ⚠️ Partially Implemented | Limited to basic string matching and length checks |
| Machine Learning Model | ❌ Not Implemented | No ML-based classification |

**Key Gaps:**

* Limited follow-up detection heuristics
* No dedicated handling for topic shift queries
* Missing machine learning classification model

### 6. API Implementation

| Specification Requirement | Current Implementation Status | Notes |
| --- | --- | --- |
| Consistent API Structure | ⚠️ Partially Implemented | Core endpoints exist but with code duplication |
| Modular API Functions | ⚠️ Partially Implemented | Some shared utilities but duplicated logic across routes |
| Clear API Documentation | ✅ Fully Implemented | API documentation exists but needs updating for duplicate functionality |
| API Error Handling | ✅ Fully Implemented | Consistent error format across endpoints |

**Key Gaps:**

* Code duplication between chat-assistant/route.ts and openai/route.ts
* Lack of centralized utilities for OpenAI client operations
* No clear designation of primary vs. deprecated endpoints

### 7. Overall Assessment

**Strengths:**

1. Solid architecture for basic query flow
2. Good thread ID persistence mechanism
3. Proper caching of file IDs by thread
4. Functional data retrieval system

**Major Gaps:**

1. Prompt engineering infrastructure is incomplete
   * Need to externalize more prompts into template files
   * Missing versioning and A/B testing
2. Limited query classification intelligence
   * Basic heuristics without ML classification
   * No topic shift detection
3. Thread management limitations
   * No thread lifecycle management
   * Limited conversation context awareness
4. Cache optimization opportunities
   * No size limits or LRU eviction
   * Missing cache invalidation strategies
   * Limited partial cache usage
5. API implementation inefficiencies
   * Significant code duplication between API routes
   * Lack of shared utilities for common OpenAI operations
   * No clear API consolidation strategy

## Next Development Cycle Recommendations

Based on the current implementation status, the following priorities are recommended for the next development cycle:

1. Complete prompt externalization and templates
2. Enhance query classification with better heuristics
3. Add cache size management and invalidation
4. Implement topic shift detection
5. Add thread lifecycle management
6. Refactor duplicated API logic between API routes
7. Implement automated testing for prompt effectiveness
8. Explore incremental data update capabilities
9. Develop enhanced analytics features
10. Prepare system for multi-year data comparisons
11. Consider performance optimizations for larger datasets

*Last updated: April 30, 2024*

ewpage

# RIA25 Testing Methodology

## Overview

This document outlines the testing methodology implemented during the development of RIA25, detailing the approaches, tools, and criteria used to ensure system quality and accuracy.

## Testing Approach

The RIA25 testing approach focuses on:

1. **Data Accuracy First**: Ensuring the system accurately represents survey data was the primary concern
2. **Preventing Fabrication**: Testing focused heavily on preventing AI-generated fabrications
3. **Edge Case Identification**: Systematic exploration of boundary conditions and special scenarios
4. **Iterative Improvement**: Test-driven refinement of prompts and system behavior
5. **Real-world Usage Simulation**: Testing scenarios based on anticipated user interactions

## Testing Categories

### 1. Data Processing Testing

| Test Type | Description | Success Criteria |
| --- | --- | --- |
| CSV Parsing | Verification of correct CSV data interpretation | 100% accurate parsing of all fields |
| Column Mapping | Testing the dynamic column mapping functionality | Correct mapping despite column order changes |
| Data Transformation | Validation of CSV to JSON transformation | JSON output matches expected structure |
| File Generation | Testing file creation for all questions | All expected files generated with correct names |

**Testing Tools**:

* Custom Node.js validation scripts
* Manual data verification

### 2. Vector Store Testing

| Test Type | Description | Success Criteria |
| --- | --- | --- |
| Embedding Verification | Testing successful embedding of survey data | All files successfully embedded |
| Retrieval Accuracy | Testing relevance of retrieved content | Relevant context returned for test queries |
| Metadata Filtering | Validating filtering by demographic segments | Correct filtering based on metadata |

**Testing Tools**:

* OpenAI API test scripts
* Retrieval validation utilities

### 3. Prompt Engineering Testing

| Test Type | Description | Success Criteria |
| --- | --- | --- |
| Segment Detection | Testing identification of demographic segments | Correct segment detection in 95%+ of test cases |
| Fabrication Prevention | Testing resistance to generating false data | Zero fabrication in test scenarios |
| Response Formatting | Validating consistent output format | Adherence to defined format guidelines |
| Edge Case Handling | Testing behavior with unusual queries | Appropriate handling of edge cases |

**Testing Tools**:

* Systematic prompt testing scripts
* Manual evaluation

### 4. Integration Testing

| Test Type | Description | Success Criteria |
| --- | --- | --- |
| End-to-End Flow | Testing complete system workflow | Successful query-to-response cycle |
| API Integration | Validating API interaction points | Correct API calls and response handling |
| Error Handling | Testing system behavior during errors | Graceful error management and appropriate messaging |

**Testing Tools**:

* End-to-end test scripts
* Manual system testing

### 5. Performance Testing

| Test Type | Description | Success Criteria |
| --- | --- | --- |
| Response Time | Measuring time from query to response | Average response time < 10 seconds |
| Concurrent Users | Testing system under multiple simultaneous users | Stable performance with target user count |
| Resource Utilization | Monitoring system resource usage | Within defined resource parameters |

**Testing Tools**:

* Performance monitoring scripts
* Load testing utilities

## Test Scenarios

### Data Accuracy Scenarios

1. **Base Demographic Queries**: Testing responses to queries about single demographic segments
   * Example: “What percentage of respondents in North America agreed with Question 1?”
   * Success Criteria: Response matches actual survey data
2. **Cross-Demographic Queries**: Testing handling of queries across multiple demographics
   * Example: “How did responses to Question 5 differ between men and women in APAC?”
   * Success Criteria: Correct data or appropriate limiting of cross-segmentation
3. **Trend Analysis Queries**: Testing year-over-year comparison capabilities
   * Example: “How did responses to Question 10 change from 2024 to 2025?”
   * Success Criteria: Accurate comparison where methodologically sound

### Fabrication Prevention Scenarios

1. **Non-existent Data Queries**: Testing handling of queries about non-existent data
   * Example: “What percentage of left-handed respondents in Antarctica agreed with Question 20?”
   * Success Criteria: Clear indication that data is not available, no fabrication
2. **Ambiguous Queries**: Testing handling of unclear or ambiguous questions
   * Example: “What do people think about work?”
   * Success Criteria: Request for clarification, no unsupported generalizations
3. **Leading Questions**: Testing resistance to questions with false premises
   * Example: “Why did satisfaction decrease across all regions in 2025?”
   * Success Criteria: Verification of premise before answering

## Test Execution Process

### Phase 1: Unit Testing

1. Individual component testing
2. Automated test execution
3. Bug identification and fixing
4. Regression testing

### Phase 2: Integration Testing

1. Component interaction testing
2. API integration verification
3. Data flow validation
4. Error handling verification

### Phase 3: System Testing

1. End-to-end workflow testing
2. Performance measurement
3. Edge case exploration
4. User experience evaluation

### Phase 4: User Acceptance Testing

1. Stakeholder testing sessions
2. Feedback collection
3. Refinement based on feedback
4. Final validation

## Automated Testing Framework

RIA25 includes an automated testing framework located in the /tests directory that enables testing of the assistant’s responses across various question categories.

### Framework Structure

tests/  
├── data/ # Test data files in CSV format  
│ ├── Workforce Analysis Questions.csv  
│ ├── Survey Evaluation Questions.csv  
│ ├── Bias Evaluation Questions.csv  
├── test-results/ # Generated test results with timestamps  
├── index.js # Main test runner and CLI interface  
├── index-tests.js # Core test implementation logic  
└── README.md # Framework documentation

### Test Data Format

The test data files use a standardized CSV format with the following columns:

* Question No - Unique identifier for each test question
* Category - The category or type of question being tested
* Question - The actual test query sent to the assistant

Example test data:

Question No,Category,Question  
1,Data Accuracy,What percentage of respondents in North America agreed with Question 1?  
2,Segmentation,How did responses to Question 5 differ between men and women?  
3,Fabrication,What percentage of left-handed respondents in Antarctica agreed with Question 20?

### Running Automated Tests

Tests are executed via npm commands:

# Run tests with default assistant  
npm test  
  
# Run tests with a specific assistant ID  
npm test -- --assistant=asst\_your\_assistant\_id  
  
# Run a specific range of questions  
npm test -- --start=10 --end=20

The test runner:

1. Displays available test types based on CSV files in the data directory
2. Prompts the user to select which test to run
3. Processes each question through the OpenAI Assistant API
4. Records both the question and the assistant’s response
5. Saves results to a timestamped file in the test-results/ directory

### Test Results and Analysis

Test results are saved as CSV files in the test-results/ directory with the format:

[Original Filename]\_YYYYMMDD\_HHMMSS.csv

Each results file contains:

* The original question number
* The question category
* The test question
* The assistant’s complete response

These results can be manually reviewed to:

* Evaluate response accuracy
* Detect potential fabricated information
* Measure segment detection capabilities
* Assess formatting consistency
* Assess overall response quality
* Identify areas for prompt enhancement

### Adding New Test Cases

To test new scenarios:

1. Create a new CSV file in the tests/data/ directory
2. Follow the naming convention: [Test Type] Questions.csv
3. Include the required columns: Question No, Category, Question
4. Run the tests with the new file

This extensible approach allows for continuous expansion of test coverage as new requirements or edge cases are identified.

### Current Test Categories

The framework includes specialized test files for:

1. **Workforce Analysis Questions** - Testing the assistant’s ability to analyze and present workforce data correctly
2. **Survey Evaluation Questions** - Assessing the assistant’s understanding of survey methodology and limitations
3. **Bias Evaluation Questions** - Testing the assistant’s resistance to leading questions and unfounded assumptions

### Manual Testing

In addition to the automated tests, manual testing is performed to:

1. Verify data accuracy in responses
2. Check for appropriate handling of complex queries
3. Ensure proper formatting of responses
4. Confirm resistance to generating fabricated data

#### Data Retrieval Tester Interface (/test-retrieval)

A dedicated manual testing interface is available at the /test-retrieval route. This tool allows developers and QA to submit queries directly to the backend data retrieval system and visualize the results, errors, and processing time.

* **How to Access:**  
  Navigate to http://localhost:3000/test-retrieval in your browser while the app is running locally.
* **Purpose:**  
  Provides a direct way to test the two-step data retrieval and analysis pipeline, independent of the main user-facing application.
* **Usage:**
  1. Enter a query in the provided input box.
  2. Submit the query to trigger the backend retrieval and analysis.
  3. Review the returned results, matched topics, files used, analysis, and validation status.
* **When to Use:**
  + During development to verify backend changes.
  + For QA to manually test edge cases and data accuracy.
  + To debug issues with the data retrieval or analysis logic.

This interface is essential for rapid iteration, debugging, and validation of backend functionality.

## Test Documentation

* **Test Plans**: Detailed plans for each testing phase
* **Test Cases**: Specific scenarios with expected outcomes
* **Test Results**: Documentation of test execution and outcomes
* **Bug Reports**: Detailed descriptions of identified issues
* **Resolution Documentation**: Record of fix implementations

## Future Testing Considerations

Future testing enhancements could include:

* Extended test scenarios for edge cases
* Performance testing under various load conditions
* Automated accuracy verification against source data
* Systematic testing of cross-segmentation rules

## Testing Limitations

1. **Multiple Segment Limitation**: System intentionally limits cross-segmentation to two demographic dimensions
2. **Ambiguity in Certain Questions**: Some survey questions contain inherent ambiguity
3. **Year Comparison Limitations**: Not all questions allow for direct year-over-year comparison
4. **Response Time Variability**: Response times can vary based on query complexity

## Continuous Improvement Approach

1. Ongoing monitoring of system performance
2. Regular prompt refinement based on usage patterns
3. Periodic testing of key scenarios
4. Feedback-driven enhancements
5. Documentation updates as system evolves

*Last updated: April 5, 2024*

ewpage

# Deploying to Vercel

This guide outlines the steps to deploy the 2025 Global Workforce Survey AI Assistant to Vercel.

## Prerequisites

Before deploying to Vercel, make sure you have:

1. A Vercel account (create one at [vercel.com](https://vercel.com) if needed)
2. The Vercel CLI installed (optional, for local testing)
3. An OpenAI API key with access to the Assistants API
4. Created and configured your OpenAI Assistant

## Setup Steps

### 1. Prepare Secrets in Vercel

You’ll need to add the following secrets to your Vercel project:

1. Log in to your Vercel dashboard
2. Navigate to your project (or create a new one)
3. Go to “Settings” → “Environment Variables”
4. Add the following environment variables:
   * OPENAI\_API\_KEY: Your OpenAI API key
   * OPENAI\_ASSISTANT\_ID: The ID of your configured assistant

These secrets should be added to all environments (Production, Preview, and Development).

### 2. Configure Deployment

There are two ways to deploy to Vercel:

#### Option A: Deploy via Vercel Dashboard

1. Connect your GitHub, GitLab, or Bitbucket repository to Vercel
2. Select the project repository
3. Vercel will automatically detect Next.js and use the appropriate build settings
4. Click “Deploy”

#### Option B: Deploy via Vercel CLI

1. Install the Vercel CLI: npm i -g vercel
2. Navigate to your project directory in your terminal
3. Run: vercel
4. Follow the prompts to link your project and deploy

### 3. Verify Configuration

The deployment should automatically use the following configuration files:

* vercel.json: Vercel deployment configuration
* next.config.js: Next.js specific settings
* package.json: Project dependencies and scripts

### 4. Post-Deployment Steps

After successful deployment:

1. Run the data preparation scripts if you haven’t already:
   * This can be done locally before deployment
   * Alternatively, you can consider adding these steps to your build process
2. Verify that your environment variables are properly set and recognized by the application
3. Test the deployed application by sending a sample query to ensure the connection to your OpenAI Assistant is working

## Troubleshooting

### Common Issues

1. **API Timeout Errors**: If you encounter timeout errors, check the functions section in vercel.json. You may need to increase the maxDuration for API routes.
2. **Missing Environment Variables**: Verify that all required environment variables are set in the Vercel dashboard.
3. **Build Failures**: Check the build logs for specific errors. Most commonly, this is related to dependencies or Node.js version issues.
4. **API Routes Not Working**: Ensure that the API routes are correctly configured and that the api directory is properly structured.

### Logs and Debugging

* Access deployment logs from the Vercel dashboard under “Deployments” → select your deployment → “View Logs”
* For runtime logs, you can use Vercel’s built-in logging system or integrate with a third-party logging service

## Performance Considerations

* Consider using Vercel’s Edge Functions for faster response times
* Use Vercel’s serverless functions for API routes that require longer processing times
* Implement caching for frequently requested data to reduce API calls to OpenAI

## Scaling

As your application grows:

1. Consider implementing rate limiting to manage OpenAI API usage
2. Use Vercel’s analytics to monitor usage patterns and optimize accordingly
3. For high-traffic scenarios, implement appropriate caching strategies

## Support

For issues related to Vercel deployment, refer to:

* [Vercel Documentation](https://vercel.com/docs)
* [Next.js Deployment Documentation](https://nextjs.org/docs/deployment)

ewpage

# RIA25 Maintenance Procedures

**Last Updated:** April 30, 2024  
**Target Audience:** System Administrators, Developers, DevOps Engineers  
**Related Documents:**

* 06\_system\_architecture.md
* 15\_thread\_data\_management.md
* 11\_vercel\_deployment\_guide.md

## Overview

This document outlines the maintenance procedures, update processes, and troubleshooting guidelines for the RIA25 system. It serves as a reference for ongoing maintenance and support activities.

## Suggested Maintenance Tasks

### Regular System Check

Key tasks to maintain system health:

* Monitor system logs and error reports
* Review OpenAI API usage and costs
* Check response times periodically
* Verify cache integrity and performance

### Periodic Review

Recommended reviews on a regular basis:

* Analyze and categorize any system errors
* Review query patterns and system utilization
* Verify data accessibility and functionality
* Assess performance metrics against benchmarks

### Ongoing Improvements

Areas for continuous improvement:

* Review and refine system prompts based on usage
* Audit system security and access controls
* Update documentation as needed
* Verify backup procedures
* Optimize OpenAI API interaction patterns

## Update Procedures

### Survey Data Updates

When new survey data becomes available:

1. **Data Preparation**
   * Validate new CSV data format
   * Ensure consistent column structure with existing data
   * Place file in /2025\_DATA\_PROCESSING/ directory
2. **Processing Script Execution**

* cd scripts  
  node process\_survey\_data.js --input="../2025\_DATA\_PROCESSING/new\_data.csv" --output="./output/split\_data/"

1. **Output Verification**
   * Verify all expected JSON files are generated
   * Check JSON structure consistency
   * Validate metadata accuracy
2. **Data Integration**
   * Once JSON files are validated, they need to be integrated with the system
   * This would require updating any data references or configurations
3. **System Testing**
   * Test queries specific to new data
   * Verify retrieval accuracy
   * Check cross-referencing with existing data

### Prompt System Updates (Recommended Process)

When updating the system prompt:

1. **Prompt Development**
   * Draft updated prompt version
   * Document changes and rationale
2. **Test Environment Setup**
   * Create test assistant with new prompt
   * Maintain existing assistant during testing
3. **Testing**
   * Execute standard test scenarios
   * Test edge cases
   * Compare results with previous prompt version
4. **Deployment**
   * Update production assistant with new prompt
   * Document prompt version change
   * Monitor initial responses for quality
5. **Rollback Procedure**
   * If issues are detected, revert to previous prompt version
   * Document issues for further refinement

### Web Interface Updates

When updating the Next.js application:

1. **Development**
   * Implement changes in development environment
   * Test functionality thoroughly
2. **Staging Deployment**
   * Deploy to staging environment
   * Conduct user acceptance testing
3. **Production Deployment**

* git push vercel main

1. **Post-Deployment Verification**
   * Verify all features function correctly
   * Check mobile and desktop compatibility
   * Validate API integration

## OpenAI API Performance Optimization

The system has been optimized for OpenAI API performance based on the implementation of the OpenAI API Optimization Plan. The following procedures should be followed to maintain and further optimize API performance.

### Performance Monitoring

Regular monitoring should include:

1. **Response Time Metrics**
   * Average response time (target: <15 seconds)
   * Time to first token (target: <3 seconds)
   * Poll-to-completion delay
   * Thread creation time
2. **API Usage Statistics**
   * Number of API calls per query
   * Token consumption per query type
   * Polling frequency and patterns
   * Cache hit/miss ratios
3. **Automated Alerts**
   * Set up alerts for response times exceeding benchmarks
   * Monitor for abnormal polling patterns
   * Track API usage spikes or anomalies

### Key Optimization Techniques

The following optimizations have been implemented and should be maintained:

1. **Adaptive Polling with Exponential Backoff**

* // Example implementation (from app/api/chat-assistant/route.ts)  
  const initialInterval = 500;  
  let currentInterval = initialInterval;  
  const maxInterval = 3000;  
  const backoffFactor = 1.5;  
    
  // In polling loop  
  await new Promise((resolve) => setTimeout(resolve, currentInterval));  
  currentInterval = Math.min(currentInterval \* backoffFactor, maxInterval);  
    
  // Reset on state change  
  if (previousStatus !== runStatus) {  
   currentInterval = initialInterval;  
   previousStatus = runStatus;  
  }
  + Performance impact: 15-20% reduction in response time
  + Maintenance: Review and adjust parameters based on API response patterns

1. **Parallel Processing Operations**
   * Message prefetching while waiting for run completion
   * Concurrent tool call processing
   * Parallelized file retrieval
2. **Partial Content Processing**

* // Stream content as it becomes available (every 3 polls)  
  if (runStatus === "in\_progress" && pollCount % 3 === 0) {  
   const partialMessages = await openai.beta.threads.messages.list(threadId);  
   if (partialMessages.data && partialMessages.data.length > 0) {  
   // Stream new content to user  
   streamPartialContent(latestMessage);  
   }  
  }
  + Performance impact: Improved perceived response time
  + Maintenance: Ensure streaming remains stable across browsers

1. **Thread Caching and Reuse**
   * Thread-level caching for file IDs and segments
   * Efficient follow-up query handling
   * Cached files for thread persistence

### Performance Tuning Guidelines

When performance issues are detected:

1. **API Interaction Optimization**
   * Review polling intervals in app/api/chat-assistant/route.ts
   * Check for unnecessary API calls or redundant operations
   * Verify parallel operations are functioning correctly
2. **Cache Efficiency Review**
   * Check thread cache hit/miss ratios
   * Verify cache update procedures are working
   * Review cache size and clean-up procedures
3. **Cold Start Mitigation**
   * Analyze first-query performance
   * Consider implementing thread pooling if cold start issues persist
   * Review pre-warming strategies if deployed on serverless platforms

## Thread and Cache Management

### Cache Maintenance

1. **Cache Directory Management**
   * Location: /cache/ directory contains thread-specific cache files
   * Format: JSON files with naming pattern thread\_[threadId].json
   * Cleanup: Consider implementing a cache cleanup script to remove old cache files:

* # Example cleanup script (remove cache files older than 30 days)  
  find ./cache -name "thread\_\*.json" -type f -mtime +30 -delete

1. **Cache Structure Verification**
   * Each cache file should contain:
     + A files array containing file IDs and segment data
     + A scope object with metadata about the thread’s data scope
   * Periodic validation ensures cache integrity
2. **Cache Metrics Collection**
   * Consider implementing metrics for:
     + Cache hit/miss ratios
     + Average cache size
     + Cache entry lifespan
     + Performance impact of caching

### Thread Management

1. **Thread Cleanup**
   * OpenAI threads persist indefinitely
   * Consider implementing a thread cleanup process for inactive threads:

* // Example thread cleanup function  
  async function cleanupInactiveThreads(olderThanDays = 30) {  
   const threadList = await openai.beta.threads.list();  
   const cutoffDate = new Date();  
   cutoffDate.setDate(cutoffDate.getDate() - olderThanDays);  
    
   for (const thread of threadList.data) {  
   const lastActive = new Date(thread.last\_active\_at || thread.created\_at);  
   if (lastActive < cutoffDate) {  
   await openai.beta.threads.del(thread.id);  
   console.log(`Deleted inactive thread: ${thread.id}`);  
   }  
   }  
  }

1. **Thread Usage Monitoring**
   * Track thread creation rates
   * Monitor thread retention and reuse rates
   * Analyze thread lifecycle patterns

## Suggested Backup Procedures

### Data Backup

1. **Export Assistant Configuration**
   * Document assistant ID and configuration settings
   * Store configuration information securely
2. **JSON Data Backup**
   * Maintain copies of all processed JSON files
   * Store original CSV data securely
   * Document backup locations
3. **Recommended Backup Schedule**
   * Configuration documentation: When changed
   * JSON data backup: After processing
   * Code repository backup: Regular intervals
4. **Backup Storage Recommendations**
   * Primary: Secure cloud storage
   * Secondary: Local storage
   * Consider encryption for sensitive data

## Troubleshooting

### Suggested Diagnostic Approaches

#### API Connectivity Issues

1. Verify API key validity and environment variable configuration
2. Check API status at OpenAI status page
3. Test basic API connectivity with curl command
4. Review API response logs for error messages

#### Data Retrieval Issues

1. Verify assistant configuration
2. Check file accessibility and integrity
3. Validate data formats and structures
4. Review retrieval mechanisms

#### Response Quality Issues

1. Analyze prompt configuration
2. Test with simplified queries
3. Review context being provided to the model
4. Check for changes in underlying data

#### Performance Issues

1. Analyze response time logs
2. Check polling patterns
3. Verify thread and cache operations
4. Review concurrent request handling
5. Check for any API quota limitations

## Monitoring Recommendations

### Performance Monitoring

* **Response Time**: Track average and peak response times
* **API Usage**: Monitor token consumption and request volume
* **Error Rate**: Track frequency and types of errors
* **Time to First Token**: Monitor latency before initial response

### Usage Monitoring

* **Query Patterns**: Review common query types
* **User Sessions**: Monitor session activity
* **Feature Utilization**: Note which capabilities are most used
* **Cache Efficiency**: Track cache hit/miss rates

### Alert Thresholds

Consider setting up alerts for:

* Extended response times (e.g., >15 seconds average)
* Elevated error rates (e.g., >5% of requests)
* High API usage (e.g., >80% of quota)
* Any system downtime
* Abnormal polling patterns

## System Recovery Guidelines

### Critical Failure Recovery

1. **Assessment**
   * Identify failure point
   * Determine impact scope
   * Establish recovery priority
2. **Containment**
   * Limit system access if necessary
   * Implement temporary workarounds
   * Communicate status to stakeholders
3. **Recovery Steps**
   * Restore data if corruption occurred
   * Redeploy application if web interface failed
   * Reconfigure API connections if integration issues
4. **Verification**
   * Test system functionality
   * Verify data integrity
   * Confirm normal operation
5. **Documentation**
   * Record incident details
   * Document recovery process
   * Update procedures based on lessons learned

## Contact Roles

Key roles for system maintenance and support:

* System Administrator: Responsible for day-to-day maintenance
* Developer: Handles technical issues and code updates
* Prompt Engineer: Manages prompt system refinements
* Project Manager: Coordinates overall system management

*Last updated: April 30, 2024*

ewpage

# Canonical Topic Mapping Reference

## Overview

The canonical topic mapping system is a fundamental component of RIA25’s data architecture. This reference document details the structure, usage, and implementation of the canonical topic approach, which enables intuitive querying and accurate cross-year survey data comparisons.

## canonical\_topic\_mapping.json

Location: scripts/reference files/canonical\_topic\_mapping.json

### File Purpose

This configuration file serves as the master reference for mapping between:

* Human-friendly canonical topics (e.g., “AI\_Attitudes”)
* Raw question identifiers in each survey year (e.g., “Q4\_9” in 2024)
* Actual data files that contain the relevant information

The file also defines rules for data comparison, market availability, and appropriate messaging when presenting data with limitations.

### Top-Level Structure

{  
 "metadata": {...},  
 "dataAccess": {...},  
 "retrievalRules": {...},  
 "allValidFiles": {...},  
 "themes": [...]  
}

#### metadata

Contains version information and last update timestamp:

"metadata": {  
 "version": "1.0",  
 "description": "Canonical topic mapping for survey data organized by themes and topics",  
 "lastUpdated": "2025-03-15"  
}

#### dataAccess

Defines paths and market information:

"dataAccess": {  
 "basePath": "scripts/output/split\_data/",  
 "comparableMarkets": [  
 "United Kingdom",  
 "United States",  
 "Australia",  
 "India",  
 "Brazil"  
 ],  
 "allMarkets2025": [  
 "United Kingdom",  
 "United States",  
 "Australia",  
 "India",  
 "Brazil",  
 "France",  
 "Germany",  
 "Japan",  
 "United Arab Emirates",  
 "Saudi Arabia"  
 ]  
}

#### retrievalRules

Documents the rules for data retrieval:

"retrievalRules": {  
 "comparable": "For topics marked as comparable, retrieve data from both 2024 and 2025 surveys, but only include data from the five comparable markets.",  
 "nonComparable": "For topics marked as non-comparable, retrieve data from only the available survey year(s) and include the appropriate user message explaining why year-on-year comparison is not available.",  
 "singleYear": "For topics available in only one year, retrieve that data and include the user message noting the limitation."  
}

#### allValidFiles

Lists all valid JSON data files for each survey year:

"allValidFiles": {  
 "2024": [  
 "2024\_1.json",  
 "2024\_2.json",  
 // etc.  
 ],  
 "2025": [  
 "2025\_1.json",  
 "2025\_2.json",  
 // etc.  
 ]  
}

#### themes

Organizes topics into thematic categories:

"themes": [  
 {  
 "name": "Talent Attraction & Retention",  
 "topics": [...]  
 },  
 {  
 "name": "Employee Experience & Work‑Life",  
 "topics": [...]  
 },  
 // etc.  
]

### Topic Structure

Each topic within a theme follows this structure:

{  
 "id": "Attraction\_Factors",  
 "canonicalQuestion": "What are the most important factors when looking for a new job?",  
 "rationale": "Identify factors that drive candidates to consider a new job; essential for talent acquisition.",  
 "mapping": {  
 "2024": [  
 {  
 "id": "Q1",  
 "file": "2024\_1.json"  
 }  
 ],  
 "2025": [  
 {  
 "id": "Q1",  
 "file": "2025\_1.json"  
 }  
 ]  
 },  
 "comparable": false,  
 "availableMarkets": [],  
 "userMessage": "Year‑on‑year comparisons not available due to methodology changes.",  
 "alternatePhrasings": [  
 "job attraction factors",  
 "what attracts new hires",  
 "important job factors"  
 ]  
}

## Key Theme Categories

The canonical topic mapping organizes topics into the following themes:

1. **Talent Attraction & Retention**
   * Topics related to job attraction factors, retention factors, attrition drivers, and intentions to leave
2. **Employee Experience & Work‑Life**
   * Topics covering work-life flexibility, culture and values, wellbeing, communication, and fulfillment
3. **Skills & Development**
   * Topics on learning and development, skills utilization, and attitudes toward AI
4. **Leadership & Management**
   * Topics addressing leadership confidence, organizational adaptation, and manager capability
5. **Workplace Dynamics**
   * Topics concerning intergenerational collaboration, diversity and inclusion, and sustainability
6. **Compensation & Benefits**
   * Topics on pay fairness and reward adequacy
7. **Workplace**
   * Topics about current and preferred work locations
8. **Perceived Barriers**
   * Topics such as imposter syndrome

## Comparable vs. Non-Comparable Topics

### Comparable Topics

Topics marked as "comparable": true can be directly compared between 2024 and 2025 survey years. These include:

* Intention\_to\_Leave
* Ideal\_Role
* Skills\_Utilization
* AI\_Attitudes
* Leadership\_Confidence
* Organizational\_Adaptation
* DEI
* Sustainability
* Pay\_and\_Reward
* Imposter\_Syndrome

For these topics, data from both years can be presented together, but only for the five comparable markets: United Kingdom, United States, Australia, India, and Brazil.

### Non-Comparable Topics

Topics marked as "comparable": false should not be directly compared between years. These include:

* Attraction\_Factors
* Retention\_Factors
* Attrition\_Factors
* Work\_Life\_Flexibility
* Culture\_and\_Values
* Employee\_Wellbeing
* Communication\_and\_Connection
* Motivation\_and\_Fulfillment
* Learning\_and\_Development
* AI\_Readiness
* Manager\_Capability
* Intergenerational\_Collaboration
* Current\_and\_Preferred

For these topics, data from each year should be presented separately with the appropriate userMessage explaining why year-over-year comparison is not possible or valid.

## Single-Year Topics

Some topics are available only in one survey year (typically 2025), such as:

* AI\_Readiness
* Manager\_Capability
* Intergenerational\_Collaboration
* Current\_and\_Preferred

For these topics, the system provides data only from the available year, along with the appropriate userMessage explaining the limitation.

## Alternate Phrasings

Each topic includes a list of alternatePhrasings that represent different ways users might refer to the same concept. For example:

"alternatePhrasings": [  
 "AI attitudes",  
 "AI perception",  
 "excitement about AI",  
 "role impact of AI",  
 "emerging technology sentiment",  
 "optimism regarding AI",  
 "AI value",  
 "AI influence"  
]

These alternate phrasings improve the system’s ability to match user queries to the appropriate canonical topics, enhancing the natural language understanding capabilities.

## Usage in the System

### Query Processing

When processing a user query:

1. The system analyzes the query to identify relevant canonical topics, using both exact matches and semantic similarity against topic IDs, canonical questions, and alternate phrasings.
2. For each identified topic, the system retrieves the corresponding data file(s) based on the mapping.
3. The comparable flag determines whether data from both years should be retrieved and compared.
4. The appropriate userMessage is incorporated into the response to provide context about any limitations.

### Data Retrieval

The system follows these rules for data retrieval:

* For comparable topics, only data from the five comparable markets is included in year-over-year comparisons.
* For non-comparable topics, data from each year is presented separately with disclaimers.
* For single-year topics, only data from the available year is presented with a note about the limitation.

## Integration with Other Components

The canonical topic mapping integrates with several other RIA25 components:

* **Data Processing**: The ETL pipeline uses the mapping to organize and normalize survey data.
* **Vector Store**: Documents in the vector store are tagged with their corresponding canonical topics.
* **Prompt System**: The system prompt includes instructions for handling comparable and non-comparable topics.
* **Response Templates**: Response templates leverage canonical questions and user messages to provide consistent formatting.

## Maintenance and Updates

### Adding New Survey Years

To incorporate data from a new survey year:

1. Add the new year’s files to the allValidFiles section.
2. For each topic, update the mapping section to include the new year’s question IDs and files.
3. Determine which topics are comparable between years and update the comparable and availableMarkets properties accordingly.
4. Update the userMessage to reflect any new limitations or considerations.

### Adding New Topics

To add a new canonical topic:

1. Create a new topic entry in the appropriate theme section.
2. Define the id, canonicalQuestion, and rationale.
3. Set up the mapping to the relevant question files.
4. Determine if the topic is comparable between years and set the comparable flag accordingly.
5. Define availableMarkets and userMessage as appropriate.
6. Add relevant alternatePhrasings to improve query matching.

## Examples

### Example 1: Comparable Topic

{  
 "id": "AI\_Attitudes",  
 "canonicalQuestion": "What are your attitudes toward AI in the workplace?",  
 "rationale": "Capture employee attitudes toward AI—including excitement, perceived value, and fears of replacement—to gauge AI readiness.",  
 "mapping": {  
 "2024": [  
 { "id": "Q4\_9", "file": "2024\_4\_9.json" },  
 { "id": "Q4\_10", "file": "2024\_4\_10.json" },  
 { "id": "Q4\_11", "file": "2024\_4\_11.json" }  
 ],  
 "2025": [  
 { "id": "Q5\_2", "file": "2025\_5\_2.json" },  
 { "id": "Q5\_3", "file": "2025\_5\_3.json" },  
 { "id": "Q5\_8", "file": "2025\_5\_8.json" }  
 ]  
 },  
 "comparable": true,  
 "availableMarkets": [  
 "United Kingdom",  
 "United States",  
 "Australia",  
 "India",  
 "Brazil"  
 ],  
 "userMessage": "Data based on comparable markets only.",  
 "alternatePhrasings": [  
 "AI attitudes",  
 "AI perception",  
 "excitement about AI",  
 "role impact of AI"  
 ]  
}

### Example 2: Non-Comparable Topic

{  
 "id": "Attraction\_Factors",  
 "canonicalQuestion": "What are the most important factors when looking for a new job?",  
 "rationale": "Identify factors that drive candidates to consider a new job; essential for talent acquisition.",  
 "mapping": {  
 "2024": [{ "id": "Q1", "file": "2024\_1.json" }],  
 "2025": [{ "id": "Q1", "file": "2025\_1.json" }]  
 },  
 "comparable": false,  
 "availableMarkets": [],  
 "userMessage": "Year‑on‑year comparisons not available due to methodology changes.",  
 "alternatePhrasings": [  
 "job attraction factors",  
 "what attracts new hires",  
 "important job factors"  
 ]  
}

## Conclusion

The canonical topic mapping system represents a cornerstone of RIA25’s data architecture, enabling intuitive querying and accurate cross-year comparisons. By organizing data around meaningful topics rather than raw question numbers, the system provides a more user-friendly experience while maintaining data integrity.

*Last updated: April 5, 2024*

ewpage

# API Reference

**Last Updated:** April 18, 2024  
**Target Audience:** Developers, API Integrators  
**Related Documents:**

* 06\_system\_architecture.md
* 15\_thread\_data\_management.md

## 1. Overview

This document provides a comprehensive reference for the API endpoints available in the RIA25 system. These endpoints enable interaction with the core functionality of the assistant, including query processing, data retrieval, and thread management.

## 2. Core Endpoints

### 2.1 Chat Assistant API

#### /api/chat-assistant/route.ts

**Purpose**: Main endpoint for processing user queries through the OpenAI Assistants API.

**Method**: POST

**Request Body**:

{  
 "assistantId": "string", // ID of the assistant to use  
 "threadId": "string", // Optional: Thread ID for continuing conversation  
 "content": "string" // User's message/query  
}

**Response**: Stream of text chunks from the assistant’s response.

**Process Flow**:

1. Creates or retrieves OpenAI thread
2. Adds user message to thread
3. Creates a run with appropriate instructions
4. Polls for run completion
5. Handles any tool calls (data retrieval)
6. Streams response back to client

**Headers**:

* Content-Type: text/event-stream

**Example Request**:

POST /api/chat-assistant  
{  
 "assistantId": "asst\_abc123",  
 "threadId": "thread\_xyz789",  
 "content": "How do work preferences vary across age groups?"  
}

### 2.2 Data Query API

#### /api/query/route.js

**Purpose**: Processes user queries, identifies relevant data files, and prepares data for analysis.

**Method**: POST

**Request Body**:

{  
 "query": "string", // The user's question  
 "context": "string", // Optional: Data context (e.g., "all-sector")  
 "cachedFileIds": ["string"] // Optional: Previously cached file IDs  
}

**Response**:

{  
 "file\_ids": ["string"], // Identified relevant file IDs  
 "matched\_topics": ["string"], // Topics matched from the query  
 "analysis": "string", // Summary analysis of the data  
 "raw\_data": {}, // Retrieved data (may be large)  
 "status": "string", // Status of the query (success, follow\_up, etc.)  
 "processing\_time\_ms": number // Processing time in milliseconds  
}

**Example Request**:

POST /api/query  
{  
 "query": "What factors influence employee retention?",  
 "context": "all-sector",  
 "cachedFileIds": ["2025\_1\_2", "2025\_3\_4"]  
}

### 2.3 Data Retrieval API

#### /api/retrieve-data/route.js

**Purpose**: Retrieves specific data files by ID.

**Method**: POST

**Request Body**:

{  
 "file\_ids": ["string"] // Array of file IDs to retrieve  
}

**Response**:

{  
 "files": [  
 {  
 "id": "string", // File ID  
 "topic": "string", // Topic associated with the file  
 "data": {} // File content  
 }  
 ],  
 "topics": ["string"], // Unique topics from retrieved files  
 "totalDataPoints": number, // Total number of data points retrieved  
 "metadata": {  
 "succeeded": number, // Number of files successfully retrieved  
 "failed": number, // Number of files that failed retrieval  
 "processing\_time\_ms": number, // Processing time in milliseconds  
 "retrieved\_at": "string" // Timestamp of retrieval  
 }  
}

**Example Request**:

POST /api/retrieve-data  
{  
 "file\_ids": ["2025\_1\_2", "2025\_3\_4"]  
}

## 3. Thread Management

### 3.1 Thread Information

#### /api/thread/[threadId]/route.js

**Purpose**: Retrieves information about a specific thread.

**Method**: GET

**Parameters**:

* threadId: ID of the thread to retrieve information for

**Response**:

{  
 "threadId": "string",  
 "created\_at": "string",  
 "last\_active": "string",  
 "message\_count": number,  
 "cached\_files": ["string"] // File IDs cached for this thread  
}

### 3.2 Thread Messages

#### /api/thread/[threadId]/messages/route.js

**Purpose**: Retrieves message history for a thread.

**Method**: GET

**Parameters**:

* threadId: ID of the thread to retrieve messages for
* limit: Optional: Number of messages to retrieve (default: 10)
* order: Optional: Order of messages (“asc” or “desc”, default: “desc”)

**Response**:

{  
 "messages": [  
 {  
 "id": "string",  
 "role": "string", // "user" or "assistant"  
 "content": "string",  
 "created\_at": "string"  
 }  
 ],  
 "has\_more": boolean  
}

## 4. Configuration

### 4.1 Assistant Configuration

#### /api/assistant/[assistantId]/route.js

**Purpose**: Retrieves configuration for a specific assistant profile.

**Method**: GET

**Parameters**:

* assistantId: ID of the assistant to retrieve configuration for

**Response**:

{  
 "id": "string",  
 "name": "string",  
 "description": "string",  
 "instructions": "string",  
 "model": "string",  
 "tools": [  
 {  
 "type": "string",  
 "function": {  
 "name": "string",  
 "description": "string",  
 "parameters": {}  
 }  
 }  
 ],  
 "file\_ids": ["string"]  
}

## 5. Error Handling

All API endpoints follow a consistent error handling pattern:

### 5.1 Error Response Format

{  
 "error": {  
 "message": "string", // Human-readable error message  
 "code": "string", // Error code for programmatic handling  
 "details": {} // Optional: Additional error details  
 }  
}

### 5.2 Common Error Codes

| Code | Description |
| --- | --- |
| invalid\_request | Request format or parameters are invalid |
| file\_not\_found | Requested file ID could not be found |
| thread\_not\_found | Requested thread ID could not be found |
| assistant\_not\_found | Requested assistant ID could not be found |
| openai\_error | Error from the OpenAI API |
| internal\_error | Internal server error |

## 6. Rate Limiting

API endpoints implement rate limiting to ensure system stability:

* **Limit**: 60 requests per minute per IP address
* **Headers**:
  + X-RateLimit-Limit: Maximum requests per window
  + X-RateLimit-Remaining: Remaining requests in current window
  + X-RateLimit-Reset: Seconds until the rate limit resets

When rate limit is exceeded, returns 429 Too Many Requests with:

{  
 "error": {  
 "message": "Rate limit exceeded",  
 "code": "rate\_limit\_exceeded",  
 "details": {  
 "retry\_after": number // Seconds to wait before retrying  
 }  
 }  
}

## 7. Security

All API endpoints require authentication:

* Development environment uses API keys in environment variables
* Production environment uses secure environment variables in Vercel
* All endpoints enforce HTTPS in production
* CORS restrictions are implemented for cross-origin requests

## 8. Best Practices

1. **Thread Management**:
   * Store and reuse thread IDs to maintain conversation context
   * Clear thread IDs when starting new conversations
2. **Error Handling**:
   * Implement retry logic for transient errors
   * Parse error codes for conditional handling
3. **Rate Limiting**:
   * Implement exponential backoff for rate limit errors
   * Distribute requests evenly to avoid rate limit bursts
4. **Efficient Data Retrieval**:
   * Cache file IDs between related queries
   * Pass cached file IDs when making follow-up queries

*Last updated: April 18, 2024*

ewpage

# Thread and Data Cache Management

**Last Updated:** April 18, 2024  
**Target Audience:** Developers, System Architects  
**Related Documents:**

* 03\_data\_processing\_workflow.md
* 06\_system\_architecture.md

## 1. Overview

This document explains how the RIA25 system manages conversation threads and implements intelligent caching to provide efficient follow-up query handling. The system uses a combination of client-side thread persistence and server-side data caching to maintain conversation context while optimizing performance.

## 2. Thread Management System

### 2.1 Thread Lifecycle

1. **Thread Creation**
   * When a user first starts a conversation, the system creates a new thread through the OpenAI API
   * A unique threadId is generated and returned to the client
   * The thread ID is stored in localStorage for persistence across sessions
2. **Thread Persistence**
   * Thread IDs are saved to client-side localStorage
   * Upon page reload, the system retrieves the thread ID from localStorage
   * This ensures conversation continuity even if the browser is closed and reopened
3. **Thread Usage**
   * All subsequent queries include the thread ID
   * API endpoints associate queries with existing threads
   * Thread state (message history) is maintained on OpenAI’s servers
4. **Thread Termination**
   * Users can manually reset conversations
   * When reset, thread ID is removed from localStorage
   * Associated thread cache is also cleared

### 2.2 Implementation Details

// Thread creation in app/api/chat-assistant/route.ts  
const thread = await openai.beta.threads.create();  
finalThreadId = thread.id;  
  
// Thread persistence in client-side code  
useEffect(() => {  
 if (threadId) {  
 localStorage.setItem("chatThreadId", threadId);  
 }  
}, [threadId]);  
  
// Thread retrieval on page load  
const [threadId, setThreadId] = useState(() => {  
 if (typeof window !== "undefined") {  
 const savedThreadId = localStorage.getItem("chatThreadId");  
 return savedThreadId || null;  
 }  
 return null;  
});  
  
// Thread reset functionality  
const refreshChat = () => {  
 setThreadId(null);  
 localStorage.removeItem("chatThreadId");  
 // Clear thread cache...  
};

## 3. Data File Caching System

### 3.1 Cache Architecture

The system implements two separate but complementary caching mechanisms:

1. **Client-Side Cache**
   * Implemented in localStorage
   * Tracks file IDs and data per thread
   * Managed in React state with localStorage persistence
   * Used to optimize client-server communication
2. **Server-Side Cache**
   * Implemented as JSON files in the cache/ directory
   * One cache file per thread (e.g., thread\_ABC123.json)
   * Tracks loaded file IDs, segments, and data
   * Managed by utils/cache-utils.ts
   * Used to optimize data retrieval and processing

### 3.2 Cache Data Structure

#### Server-Side Cache Structure

Each thread cache file contains:

{  
 "files": [  
 {  
 "id": "2025\_1\_2",  
 "data": {  
 "region": { /\* segment data \*/ },  
 "age": { /\* segment data \*/ }  
 },  
 "loadedSegments": ["region", "age"],  
 "availableSegments": ["region", "age", "gender", "overall"]  
 },  
 // More files...  
 ],  
 "scope": {  
 // Additional metadata about the thread's data scope  
 }  
}

#### Client-Side Cache Structure

{  
 "thread\_123ABC": {  
 fileIds: ["2025\_1\_2", "2025\_3\_4"],  
 data: { /\* raw data from these files \*/ }  
 },  
 "thread\_456XYZ": {  
 fileIds: ["2025\_6\_7"],  
 data: { /\* raw data from these files \*/ }  
 }  
}

### 3.3 Server-Side Cache Implementation

The server-side cache is implemented in utils/cache-utils.ts with two main functions:

#### Loading Thread Cache

/\*\*  
 \* Loads the cache for a given thread.  
 \* @param threadId - The thread identifier.  
 \* @returns The cache entry containing cached files and metadata.  
 \*/  
export async function getCachedFilesForThread(  
 threadId: string  
): Promise<CachedFile[]> {  
 const cacheFilePath = path.join(CACHE\_DIR, `${threadId}.json`);  
 try {  
 if (!fs.existsSync(cacheFilePath)) {  
 return [];  
 }  
 const content = await fs.promises.readFile(cacheFilePath, "utf8");  
 const cacheEntry = JSON.parse(content);  
 // Convert loadedSegments and availableSegments to Set  
 const files = cacheEntry.files.map((file: any) => ({  
 ...file,  
 loadedSegments: new Set(file.loadedSegments),  
 availableSegments: new Set(file.availableSegments),  
 }));  
 return files;  
 } catch (error) {  
 logger.error(`Error reading cache for thread ${threadId}:`, error);  
 return [];  
 }  
}

#### Updating Thread Cache

/\*\*  
 \* Updates the cache for a given thread.  
 \* Merges new files or updates existing files with new segment data.  
 \* @param threadId - The thread identifier.  
 \* @param newFiles - Array of new or updated cached files.  
 \* @returns void  
 \*/  
export async function updateThreadCache(  
 threadId: string,  
 newFiles: CachedFile[]  
): Promise<void> {  
 // Implementation details...  
}

### 3.4 Client-Side Cache Implementation

The client-side cache is implemented using React state with localStorage persistence:

// Initialize cache from localStorage  
const [threadDataCache, setThreadDataCache] = useState(() => {  
 if (typeof window !== "undefined") {  
 const savedCache = localStorage.getItem("threadDataCache");  
 return savedCache ? JSON.parse(savedCache) : {};  
 }  
 return {};  
});  
  
// Update cache with new data  
const updateThreadCache = useCallback((threadId, newFileIds, newData) => {  
 setThreadDataCache((prevCache) => {  
 const updatedCache = { ...prevCache };  
  
 // Initialize if needed  
 if (!updatedCache[threadId]) {  
 updatedCache[threadId] = { fileIds: [], data: {} };  
 }  
  
 // Add new file IDs without duplicates  
 const existingIds = updatedCache[threadId].fileIds || [];  
 const allFileIds = [...existingIds];  
  
 newFileIds.forEach((id) => {  
 if (!allFileIds.includes(id)) {  
 allFileIds.push(id);  
 }  
 });  
  
 updatedCache[threadId].fileIds = allFileIds;  
  
 // Update data  
 updatedCache[threadId].data = {  
 ...(updatedCache[threadId].data || {}),  
 ...newData,  
 };  
  
 // Save to localStorage  
 if (typeof window !== "undefined") {  
 localStorage.setItem("threadDataCache", JSON.stringify(updatedCache));  
 }  
  
 return updatedCache;  
 });  
}, []);

## 4. Segment-Aware Incremental Loading

The cache system implements segment-aware incremental loading to optimize data retrieval for follow-up queries.

### 4.1 Core Concepts

1. **Data Segments**
   * Data files are organized into segments (e.g., “region”, “age”, “gender”)
   * Each segment contains a subset of the file’s data
   * Not all segments are needed for every query
2. **Incremental Loading**
   * System tracks which segments have already been loaded per file
   * For follow-up queries, only loads missing segments
   * Merges new segments with existing cached data
3. **Response Optimization**
   * Responses include only newly loaded (incremental) segments
   * Reduces payload size and processing time
   * Client merges incremental updates into local cache

### 4.2 Implementation

#### Segment Tracking

Each cached file tracks:

* loadedSegments: Which segments have already been loaded
* availableSegments: Which segments exist in the original file

#### Incremental Loading Process

1. Identify which segments are needed for the current query
2. Check which segments are already loaded in the cache
3. Load only the missing segments from disk
4. Merge the new segments into the cache
5. Return only the newly loaded segments in the response

#### Code Example: Loading Missing Segments

function loadMissingSegments(cachedFiles, neededSegments) {  
 const missingSegments = {};  
  
 cachedFiles.forEach((file) => {  
 // Determine which segments are missing for this file  
 const missing = neededSegments.filter(  
 (seg) => !file.loadedSegments.has(seg) && file.availableSegments.has(seg)  
 );  
  
 if (missing.length > 0) {  
 missingSegments[file.id] = missing;  
 }  
 });  
  
 // Load missing segments from disk  
 // Update cache with new segments  
 // Return only the newly loaded data  
}

## 5. Cache Usage for Follow-up Queries

### 5.1 Follow-up Detection

The system uses several methods to detect follow-up queries:

1. **Query Analysis**
   * Short queries often indicate follow-ups
   * Presence of pronouns (it, they, these) suggests follow-up
   * References to previous content
2. **Thread Context**
   * All queries in the same thread are evaluated in context
   * System checks if query is related to previous topics

### 5.2 Cache Utilization

When a follow-up query is detected:

1. The system retrieves the thread’s cached file IDs
2. Passes these IDs to the query processing pipeline
3. Uses the cache to avoid reloading already processed data
4. May load additional files if the follow-up expands the topic

// In API call  
const dataRetrievalResponse = await fetch("/api/query", {  
 method: "POST",  
 body: JSON.stringify({  
 query: questionText,  
 context: "all-sector",  
 cachedFileIds: cachedFiles.fileIds, // Pass cached IDs  
 }),  
});  
  
// In retrieval.js  
if (cachedFileIds && cachedFileIds.length > 0 && isLikelyFollowUp) {  
 // For follow-ups, use cached data  
 return {  
 file\_ids: cachedFileIds,  
 status: "follow\_up",  
 // ...  
 };  
}

## 6. Performance Benefits

The thread and cache management system provides significant performance benefits:

1. **Reduced API Calls**
   * Avoids redundant data retrieval for follow-up questions
   * Minimizes calls to data sources
2. **Lower Latency**
   * Faster responses for follow-up queries
   * Incremental loading reduces processing time
3. **Smaller Payloads**
   * Only missing segments are transferred
   * Prevents unbounded growth of response payloads
4. **Better User Experience**
   * Quicker responses maintain conversation flow
   * Consistent data across related queries

## 7. Future Enhancements

### 7.1 Vercel KV Migration

As described in the vercel\_kv\_cache\_migration\_plan.md, the system will be enhanced to:

* Replace file-based caching with Vercel KV in production
* Implement environment-aware cache selection (file vs. KV)
* Add cache expiration and size management
* Improve serialization/deserialization performance

### 7.2 Enhanced Follow-up Detection

Future versions will improve follow-up detection through:

* ML-based query classification
* More sophisticated context tracking
* Multi-topic query handling
* Intent-based cache utilization

*Last updated: April 18, 2024*

ewpage

# RIA25 Glossary

## Overview

This glossary defines key terms and concepts used throughout the RIA25 project documentation to ensure consistent understanding across team members and stakeholders.

## Terms and Definitions

### A

#### Anti-Fabrication Measures

Systems and prompt engineering techniques implemented to prevent the AI from generating fictional or inaccurate data that is not supported by the survey results.

#### API (Application Programming Interface)

A set of protocols and tools for building software applications that specifies how components should interact. In RIA25, primarily refers to the OpenAI API.

#### Assistants API

OpenAI’s API offering that provides capabilities for creating assistants with specific instructions, knowledge, and tools.

#### Alternate Phrasings

Different ways users might refer to the same canonical topic, included in the canonical topic mapping to improve query matching.

### C

#### Canonical Topic

A standardized, human-friendly topic identifier (e.g., “AI\_Attitudes”, “Work\_Life\_Flexibility”) that organizes survey data conceptually rather than by question numbers, enabling intuitive querying and accurate cross-year comparisons.

#### Canonical Question

The human-readable question associated with each canonical topic in the mapping system, representing the essence of what that topic measures.

#### Canonical Topic Mapping

The system that maps between human-friendly topics, raw question identifiers, and data files across survey years, stored in canonical\_topic\_mapping.json.

#### Comparable Topic

A canonical topic marked as directly comparable between 2024 and 2025 survey years, allowing for valid year-over-year comparisons but only for the five comparable markets.

#### Column Mapping

The technique implemented in the data processing pipeline to dynamically map columns from the CSV file to the structured JSON format, providing resilience to format changes.

#### Comparable Markets

The five markets that can be validly compared between 2024 and 2025: United Kingdom, United States, Australia, India, and Brazil.

#### Cross-Segmentation

The analysis of survey data across multiple demographic dimensions (e.g., analyzing responses by both gender and region simultaneously).

### D

#### Demographic Segment

A specific group defined by shared characteristics such as age range, gender, geographic region, organization size, etc., used to categorize survey respondents.

#### Data Normalization

The process of standardizing data representation across different survey years, including consistent market names, demographic keys, and percentage formats.

### E

#### Edge Case

An unusual or extreme scenario that tests the limits of the system’s design and handling capabilities.

#### ETL Pipeline

Extract, Transform, Load process that converts raw survey data into the normalized format used by RIA25.

### F

#### Fabrication

The generation of information or data that is not actually present in the underlying survey data.

### G

#### Global Workforce Survey

The annual survey that collects data about workforce trends, attitudes, and experiences worldwide, forming the basis of RIA25’s data.

### J

#### JID (JSON Identifier)

A unique identifier for each JSON file in the system, typically based on the question number and survey year.

#### JSON (JavaScript Object Notation)

A lightweight data-interchange format used for structuring the processed survey data in RIA25.

### M

#### Metadata

Supplementary information about the survey data, including question text, question number, survey year, and other contextual information.

### N

#### Next.js

The React framework used to build the web interface for RIA25.

#### Non-Comparable Topic

A canonical topic marked as not directly comparable between survey years due to methodology changes or question framing differences.

#### Normalized Data Strategy

RIA25’s approach to organizing survey data around canonical topics rather than raw question numbers, ensuring consistent data representation and valid comparisons.

### O

#### OpenAI

The company providing the AI models and API services that power RIA25’s intelligence capabilities.

### P

#### Prompt Engineering

The process of designing and refining instructions (prompts) given to the AI model to optimize its responses and behavior.

### Q

#### Query

A user request for information or analysis submitted to the RIA25 system.

### R

#### Response Data

The structured information about how survey respondents answered specific questions, typically broken down by demographic segments.

#### RIA25 (Research Insights Assistant 2025)

The AI-powered system designed to provide insights from the 2025 Global Workforce Survey data.

#### Retrieval Rules

Guidelines defined in the canonical topic mapping that specify how to retrieve and present data based on topic comparability.

### S

#### Segment Detection

The system’s ability to identify which demographic segments are relevant to a particular user query.

#### Single-Year Topic

A canonical topic available in only one survey year, typically 2025, with no counterpart in other years.

#### System Prompt

The core set of instructions that guide the AI’s behavior, including rules for data handling, segment detection, and response formatting.

### T

#### Themes

Categorical groupings of canonical topics in the mapping file, such as “Talent Attraction & Retention” or “Skills & Development”.

#### Two-Segment Rule

A design principle in RIA25 that limits cross-segmentation analysis to a maximum of two demographic dimensions to maintain statistical validity.

#### Topic-Centric Structure

The organization of data around meaningful topics rather than raw question numbers, a key feature of the normalized data strategy.

### U

#### User Message

A predefined message included in the canonical topic mapping that explains any limitations or considerations when presenting data for a topic.

### V

#### Vector Embedding

The process of converting text data into numerical vector representations that capture semantic meaning, enabling similarity-based retrieval.

#### Vector Store

A database optimized for storing and retrieving vector embeddings, used in RIA25 to enable efficient retrieval of relevant survey data.

#### Vercel

The cloud platform used for deploying the RIA25 web interface.

### Y

#### Year-over-Year Comparison

Analysis that examines how survey responses have changed between the 2024 and 2025 surveys for comparable questions.

## Acronyms

| Acronym | Definition |
| --- | --- |
| API | Application Programming Interface |
| CSV | Comma-Separated Values |
| DEI | Diversity, Equity, and Inclusion |
| ETL | Extract, Transform, Load |
| JSON | JavaScript Object Notation |
| RIA | Research Insights Assistant |
| YoY | Year-over-Year |
| QA | Quality Assurance |

## Survey Demographic Categories

| Category | Description | Example Values |
| --- | --- | --- |
| Region | Geographic region of respondent | North America, EMEA, APAC, LATAM |
| Age | Age range of respondent | 18-24, 25-34, 35-44, 45-54, 55+ |
| Gender | Gender identity of respondent | Male, Female, Non-binary, Prefer not to say |
| Org Size | Size of respondent’s organization | <100, 100-999, 1000-4999, 5000-9999, 10000+ |
| Sector | Industry sector of respondent | Technology, Finance, Healthcare, Manufacturing, etc. |
| Job Level | Hierarchical position in organization | Entry-level, Mid-level, Senior, Executive |

## Technical Environment

| Component | Technology | Purpose |
| --- | --- | --- |
| Frontend | Next.js, React | User interface |
| Backend Services | Node.js | Data processing |
| AI Integration | OpenAI API | Intelligence capabilities |
| Vector Database | OpenAI Embeddings | Semantic search |
| Deployment | Vercel | Hosting platform |

## Related Documentation

For more detailed information on specific topics, refer to:

* [Normalized Data Strategy](/RIA25_Documentation/3_Data_Architecture/normalized_data_strategy.md) - Complete overview of the normalized data approach
* [Canonical Topic Mapping Reference](/RIA25_Documentation/11_References/canonical_topic_reference.md) - Detailed reference for the canonical topic mapping system

*Last updated: April 5, 2024*