Intelligent Workflows AI Planet - Complete Source Code Documentation

- **Project Name:** Intelligent Workflows - No-Code AI Workflow Builder
- **Document Version:** 1.0
- **Last Updated:** January 18, 2026
- **Document Type:** Source Code Reference
- **Status:** Production Ready

1. Project Overview

Purpose

A comprehensive no-code workflow automation platform that enables users to create visual AI workflows. The system integrates real-time Large Language Models (LLMs), handles document embeddings for semantic search, and executes complex logic chains through a drag-and-drop interface.

Key Technologies

- **Frontend:** React 18.2.0, ReactFlow 11.10.0, Tailwind CSS 3.x
- **State Management:** React Context API (Global Store)
- **Routing:** React Router v6
- **HTTP Client:** Axios (Interceptors & Error Handling)
- **Icons:** Lucide React
- **Build Tool:** Vite

Supported Workflows

1. **User Query Node:** Entry point for dynamic user questions.
2. **Knowledge Base Node:** Vector database retrieval and RAG (Retrieval-Augmented Generation).
3. **LLM Engine Node:** Live connection to OpenAI GPT-4, Google Gemini, and Anthropic Claude.
4. **Output Node:** Renders Markdown, JSON, or Text responses.

2. Directory Structure

The project follows a modular features-based architecture:

- `frontend/src/`
  - `App.jsx` -> Root component with routing configuration.
  - `main.jsx` -> Entry point.
  - `components/`
    - `WorkflowCanvas.jsx` -> The primary ReactFlow workspace.
    - `WorkflowNode.jsx` -> Smart component handling individual node logic.
    - `WorkflowExecutor.jsx` -> Handles the runtime execution loop.

        - NodeSettings.jsx -> Configuration panel for API keys and Model params.
    - context/
        - AuthContext.jsx -> JWT handling and user session management.
        - WorkflowContext.jsx -> Global state for nodes, edges, and execution logic.
    - api/
        - endpoints.js -> Centralized API service definitions.

3. Frontend Architecture

Component Hierarchy

The application follows a strict provider pattern to ensure state accessibility:

App (Root) -> BrowserRouter -> AuthProvider -> WorkflowProvider -> Routes -> Protected Pages

Layered Design Pattern:

1. **Presentation Layer:** React components (Canvas, Nodes, Panels).
2. **State Layer:** Context API (Authentication, Workflow Data).
3. **Service Layer:** Axios instances managing API communication.

4. Core Context Management

4.1 AuthContext.jsx

Purpose: Manages the entire user session lifecycle including JWT persistence and auto-renewal.

Key Features:

- **Session Persistence:** Automatically loads user data and tokens from localStorage on boot.
- **Secure Storage:** Manages access tokens and handles logout cleanup.
- **State Exposure:** Provides isAuthenticated, user, and token to the rest of the app.

4.2 WorkflowContext.jsx

Purpose: The "Brain" of the frontend application. It manages the graph state (nodes/edges) and handles the communication bridge to the execution engine.

Real-Time Execution Logic:

Instead of static responses, the executeWorkflow function now triggers a live backend job:

1. **State Locking:** Sets isExecuting to true to prevent concurrent modifications.
2. **Payload Construction:** Aggregates current node configurations and user input.

3. **API Dispatch:** Sends the payload to the execution endpoint.
4. **Response Handling:** Receives the streaming or final text response from the LLM and updates the global state.

State Structure:

- **Nodes:** Array of objects containing position, type, and live config data.
- **Edges:** Connection array defining the data flow path.
- **WorkflowData:** Stores the active `query` and the live `response` payload.

5. Component Documentation

5.1 WorkflowCanvas.jsx

Purpose: The interactive workspace wrapper around ReactFlow.

Key Functions:

- `onDrop`: Calculates real-time coordinates relative to the viewport to place new nodes.
- `onNodesChange`: Syncs drag-movements with the global Context state.
- `onConnect`: Validates and establishes logical connections between nodes (e.g., preventing loops or invalid type connections).

5.2 WorkflowNode.jsx

Purpose: A smart wrapper for individual nodes that handles specific logic based on `data.type`.

Execution Logic:

- **User Query Node:** captures input and triggers the `handleExecuteQuery` function.
- **LLM Engine Node:** Visualizes the active model (e.g., displaying "GPT-4" badge).
- **Output Node:** Subscribes to `workflowData.response` and auto-updates when data arrives from the backend.

5.3 ComponentPanel.jsx

Purpose: The library of available tools. It defines the default configuration for every new node.

Default Configurations:

- **LLM Engine:** Defaults to `GPT-4o`, `Temperature: 0.7`.
- **Knowledge Base:** Defaults to `text-embedding-3-large`.
- **Output:** Defaults to `Markdown` format.

6. Data Flow & Integration

6.1 Workflow Execution Flow

This describes the lifecycle of a user request from input to AI response.

`User Inputs Query` -> `Click Execute` -> `Context Aggregation` -> `API Request (POST /execute)` -> **[Backend Processing]** -> `Receive Response` -> `Update Output State` -> `Render Result`

Detailed Steps:

1. **Input Phase:** User types data into the "User Query" node; state is updated in `WorkflowContext`.
2. **Trigger:** The execute action bundles the node graph (JSON) and the user query.
3. **Processing:** The system awaits the asynchronous response from the AI Provider.
4. **Completion:** The returned data (text, JSON, or code) is injected into the "Output" node variable.

6.2 Node Connection Logic

The data flows strictly downstream from input to output.

`User Query (Input)` -> `Knowledge Base (Context Injection)` -> `LLM Engine (Processing)` -> `Output Node (Display)`

7. API Integration

Service Layer (endpoints.js)

The frontend utilizes a centralized API definition file to manage endpoints.

- `workflowsAPI.list()` -> `GET /api/workflows`
- `workflowsAPI.create(data)` -> `POST /api/workflows`
- `workflowsAPI.execute(id, data)` -> `POST /api/workflows/{id}/execute`

Execution Payload Example:

When execute is called, the frontend sends a structured JSON object containing the entire graph topology and current configuration variables to ensuring the backend executes exactly what is visualized.

8. State Management Pattern

Context Architecture

The application uses a **Three-Tier State Strategy**:

1. **Local State (Component Level):** Used for UI toggles like "Show Settings" or "Is Loading" spinners.
2. **Context State (Feature Level):** Used for the Node Graph, Authentication User Object, and Execution Results.
3. **Server State (API Level):** Managed via `useEffect` hooks that sync the dashboard list with the database.

9. Routing & Navigation

Route Structure

- **Public Routes:**
  - / -> Home (Landing Page)
  - /login -> Auth Interface
- **Protected Routes (Guarded by useAuth):**
  - /dashboard -> Workflow Management List
  - /builder/:workflowId -> The active IDE for building workflows

10. Performance & Best Practices

Optimization Techniques implemented:

1. **Memoization:** Heavy calculation functions and event handlers (like onNodesChange) are wrapped in useCallback to prevent render thrashing.
2. **State Separation:** The Canvas state is decoupled from the Execution state to ensure dragging a node doesn't trigger a re-execution logic check.
3. **Lazy Loading:** Heavy UI elements like the "Settings Panel" and "Executor Modal" are only rendered when requested by the user.

Error Handling Strategy:

- **Validation:** The WorkflowControls component runs a topology check before saving (e.g., ensuring an Output node exists).
- **Boundaries:** A Global ErrorBoundary wraps the main canvas to catch runtime crashes without breaking the entire app.
- **Feedback:** API errors (401/500) are caught in the Axios interceptor and displayed as user-friendly toast notifications.