# RishFlow Application Technical Documentation

## 1. Overview

RishFlow is a smart file organizer application that helps users declutter their directories by automatically sorting files into logical categories. It features a modern, responsive user interface built with React and a robust backend built with Python. The application supports various sorting modes, including a sophisticated AI-powered mode that uses computer vision and optical character recognition (OCR) to classify files based on their content.

## 2. Architecture

The application follows a hybrid architecture:

- **Frontend**: A Single Page Application (SPA) built with React, Vite, and Tailwind CSS. It communic
- **Backend**: A Python application that handles file system operations, heavy processing (AI/Image
- **Communication**: The frontend calls Python functions exposed by `pywebview`, and the backend

## 3. Libraries & Dependencies

### 3.1 Backend (Python)

- **GUI & Integration**:
- `pywebview`: Hosts the web-based UI in a native window.
- `PyQt5`: Used as the underlying GUI framework for `pywebview` on Windows.
- **Image Processing & AI**:
- `opencv-python` (`cv2`): Computer vision library used for face detection, edge detection, and color a
- `Pillow` (`PIL`): Image processing library used for handling image formats.
- `pytesseract`: Python wrapper for Google's Tesseract-OCR engine, used to extract text from images
- `numpy`: Fundamental package for scientific computing, used for array manipulations in image proc
- **Utilities**:
- `sqlite3`: (Standard Lib) Lightweight database for logging user activity and operations.
- `threading`: (Standard Lib) For running long-running tasks (organizing, scanning) without freezing th
- `hashlib`: (Standard Lib) For calculating file hashes to find duplicates.
- `shutil`: (Standard Lib) For high-level file operations (moving files).

### 3.2 Frontend (React/TypeScript)

- **Core**:
- `React`: UI library.
- `Vite`: Build tool and development server.
- `TypeScript`: Static typing for better code quality.
- **Styling & UI**:
- `Tailwind CSS`: Utility-first CSS framework.
- `Framer Motion`: For animations and transitions.
- `Lucide React`: Icon set.
- `@radix-ui/*`: Unstyled, accessible UI primitives (used for Dialogs, Menus, Tooltips, etc).
- `Sonner`: Toast notifications.
- **Data Visualization**:
- `Recharts`: Composable charting library for React (used for storage usage visualization).

## 4. Core Logic & Algorithms

### 4.1 File Organization Modes

The `RishFlowAPI.start_organizing` method in `app.py` orchestrates the sorting process. It supports several modes:

1. **File Extension (Simple)**:
   - Logic: Splits filename to get extension (e.g., `.jpg`).
   - Action: Moves file to a folder named after the extension (e.g., `JPG`).

2. **Date Modified**:
   - Logic: Retrieves modification timestamp (`os.path.getmtime`).
   - Action: Converts timestamp to `YYYY-MM-DD` format and moves file to corresponding folder.

3. **Size Category**:
   - Logic: Checks file size (`os.path.getsize`).
   - Categories:
   - **Small**: < 1 MB
   - **Medium**: 1 MB - 100 MB
   - **Large**: > 100 MB

4. **File Name (Alphabetical)**:

- Logic: Checks the first character of the filename.
- Categories: `A-Z` (for letters), `0-9` (for digits), `Symbols` (for others).

5. **AI-Based Content (Advanced)**:
    - The `AISmartSorter` class in `ai_sorter.py` determines the category based on content analysis.

### 4.2 AI Classification Logic (`ai_sorter.py`)

This is the most complex part of the application, utilizing several heuristics and algorithms:

- **Image Classification**:

1. **Family/People**: Uses Haar Cascades (`haarcascade_frontalface_default.xml`, `haarcascade_profileface.xml`) to detect faces. If faces are found -> `Images/Family`.
2. **Screenshots**: Uses **Canny Edge Detection** to find edges. Calculates **Edge Density** (Ratio of edge pixels to total pixels). If > 8% -> `Images/Screenshots`.
    - *Math*: $Density = \frac{\sum pixels_{edge}}{Height \times Width}$
3. **Receipts**: Estimates text density using OCR confidence scores. If heavily textual (> 15% text density) -> `Images/Receipts`.
4. **Memes**: Checks for "colorfulness" by converting image to HSV color space and calculating mean saturation. High saturation + text -> `Images/Memes`.

- **Document Classification**:
- Uses Tesseract OCR to extract text from the first few pages.
- Searches for keywords:
- "invoice", "receipt", "bill" -> `Documents/Receipts` (Extracts date from text!).
- "resume", "cv" -> `Documents/Resume`.
- "report", "proposal" -> `Documents/Reports`.

- **Code Classification**:
- Reads the first 2KB of the file.
- Counts keywords specific to languages (e.g., `def`, `import` for Python; `function`, `const` for JS).
- Assigns the language with the highest keyword count.

### 4.3 Duplicate Finding Logic (`duplicate_finder.py`)

The `DuplicateFinder` uses **hashing** to identify identical files.

- **Algorithm**: MD5 (Message Digest Algorithm 5).

- **Process**:

1. Reads each file in 64KB chunks to efficiently compute a 128-bit hash.
2. Stores hashes in a dictionary: `{ hash: [list_of_filepaths] }`.
3. Any entry with more than 1 file path is a duplicate.
    - *Note*: While MD5 is not cryptographically secure against collision attacks, it is fast and sufficient f

## 5. Mathematical Concepts Used

1. **Hashing (MD5 & SHA256)**: Maps data of arbitrary size to fixed-size string. Used for identifying duplicates and caching AI results.
2. **Edge Detection (Calculus/Gradient)**: The Canny algorithm uses gradients (derivatives) of image intensity to find edges. The app calculates the density of these edges.
3. **Statistical Analysis**:
    - **Mean**: Used to calculate average color saturation.
    - **Density/Ratio**: Used to determine if an image is a screenshot or text-heavy.
4. **Coordinate Geometry**: Used implicitly in image processing (pixel coordinates) and defining regions of interest for OCR.

## 6. Execution Flow

1. **Startup**: `app.py` initializes the `RishFlowAPI` class and starts the `pywebview` window, loading the React app from `dist/index.html`.
2. **User Interaction**: User clicks "Select Folder". Frontend calls `backend.browse_folder()`.
3. **Scanning**: Backend scans the folder, categorizes files by type (image, video, etc.) for the UI stats, and returns a JSON object.
4. **Organization**: User selects a mode (e.g., AI) and clicks "Organize".
    - Frontend calls `backend.start_organizing()`.
    - Backend spawns a **Background Thread** to prevent UI freezing.
    - Files are processed one by one, moved to new paths, and logged in `activity_log`.
    - `last_ops.json` is updated to allow for "Undo" functionality.
5. **Completion**: Backend executes JavaScript (`window.onOrganizeComplete`) to notify the frontend to refresh the view.

## 7. File Structure & Verification

- `app.py`: Main entry point and backend API.

- `ai_sorter.py`: Contains `AISmartSorter` class with all vision/OCR logic.
- `duplicate_finder.py`: Contains `DuplicateFinder` class.
- `rishflow_activity.db`: SQLite database storing history of all moves.
- `last_ops.json`: JSON file storing the specific paths of the last operation for the "Revert" feature.
- `requirements.txt`: Python package dependencies.
- `RishFlow UI Design/`: Source code for the Frontend.