Software Requirements Specification

for

File System Watcher

Version 1.0 approved

Prepared by Mansur Yassin and Tairan Zhang

University of Washington Tacoma

April 21st, 2025

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Mansur Yassin	05/05/25	Completed Document parts 3 and 4 for our group of two.	1.0

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) defines the functional and non-functional requirements for version 1.0 of the File Monitoring and Logging System. The system enables users to monitor local file system events (such as create, delete, modify, rename) for files with specific extensions. The application will display these events in real-time within a graphical user interface (GUI) and optionally store them in a SQLite database for later querying.
This document is intended for developers, testers, and future maintainers of the software. Unless otherwise noted, all requirements described here are considered high priority and are committed for release 1.0.

1.2 Document Conventions

This document uses standard IEEE 830-1998 formatting for requirement specifications.
All functional requirements are labeled using the format (FR-x.y) where x represents the feature section and y the specific requirement. Keywords such as shall, should, and may follow the standard definitions: Shall = mandatory requirement, Should = recommended but not required May = optional feature

1.3 Intended Audience and Reading Suggestions

This document is primarily intended for: Developers, to understand what to implement and how it integrates with system components. Testers, to verify that all requirements are properly implemented. Project managers, to track milestones and scope. Instructors or evaluators, to assess the completeness and scope of the project. Readers unfamiliar with software development may begin with Section 2 (Overall Description) for an overview, followed by Sections 3–5 for detailed specifications.

1.4 Project Scope

The File Monitoring and Logging System is a desktop application with a graphical user interface that: Allows users to select file types to monitor (e.g., .txt, .pdf) Detects file system changes using system APIs (e.g., Java WatchService, .NET FileSystemWatcher, or Python watchdog)
Displays changes in real time (including the filename, full path, event type, and timestamp)
Saves logged events to a local SQLite database provides a searchable interface for querying historical file events Offers a menu-driven UI with accessibility and tooltips

1.5 References

IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications
Java WatchService API Documentation -
https://docs.oracle.com/javase/8/docs/api/java/nio/file/WatchService.html

SQLite C/C++/Java documentation – https://www.sqlite.org/docs.html

Python watchdog module – https://pypi.org/project/watchdog/

Project description: file_watcher_project.pdf (documentation guide)

2. Overall Description

2.1 Product Perspective

This system is a **new, stand-alone product** with no external dependencies beyond the local file system and a SQLite database. It is not part of an existing system but may be extended in the future to support remote directory monitoring or cloud storage integration.

System Diagram:

$$\begin{tikzcd}
\text{[File System]} \arrow[r] & \text{[Monitoring Module]} \arrow[r] & \text{[GUI Display]} \arrow[r] & \text{[SQLite Database]} \\
 & & \text{[User Interaction]} \arrow[u] &
\end{tikzcd}$$

2.2 Product Features

Real-time monitoring of file events in specified directories, GUI-based control panel with Start/Stop monitoring buttons, Menu bar with Help/About and database export options, Support for customizable file extension filters, Logging to SQLite database (including filename, path, event, timestamp) Querying interface for searching past events by extension, type, or time.

2.3 User Classes and Characteristics

General Users: Basic computer knowledge; it's able to interact with a GUI Advanced Users / Developers: May use the system for testing file-based applications or logging behavior No programming knowledge is required for end users

2.4 Operating Environment

Desktop OS: Windows 10/11, macOS, or Linux Runtime: Java 8+ / Python 3.8+ / .NET Framework (depending on implementation) Database: Local file-based SQLite Display: Minimum 1024x768 resolution.

2.5 Design and Implementation Constraints

*Must use the standard file watching API based on the chosen platform: Java WatchService OR .NET FileSystemWatcher OR Python watchdog GUI must be implemented using native or cross-platform frameworks (e.g., JavaFX, Tkinter, WinForms) Database must be SQLite
Logging must occur asynchronously to avoid freezing the GUI.*

2.6 User Documentation

*The system will include: A built-in "About" page via the Help menu, Tooltips for all buttons
a README file explaining setup, usage, and query features, tutorial video (optional, extra credit)*

2.7 Assumptions and Dependencies

It is assumed that the user has permission to access and monitor the selected file directories. SQLite libraries or drivers will be pre-installed or bundled with the executable. The system assumes a single user environment and does not support concurrent multiuser access.

3. System Features

This section organizes the system's functional requirements based on its primary features. Each feature outlines specific behavior, interactions, and associated software capabilities needed to meet the system goals of the File System Watcher project.

3.1 File Monitoring and Event Logging

3.1.1 Description and Priority

This feature enables the application to monitor a user selected directory for real time file system changes. The system captures four core event types: file creation, deletion, modification, and renaming. It logs these events and displays them within the main graphical interface. This feature is essential to the system's functionality and is assigned High priority. The benefit of this feature is rated 9, the penalty of failure is rated 9, the cost to implement is rated 4, and the risk is rated 3.

3.1.2 Stimulus/Response Sequences

The user launches the application and selects a directory to monitor. The user specifies one or more file extensions to watch. The user clicks the "Start Monitoring" button. The system begins listening for file system events in the selected directory. When an event occurs, the system logs the file name, full path, type of event, and timestamp. The log appears in the on-screen event table. The user clicks "Stop Monitoring" to end the session. The system halts event capture and retains the session log in memory.

3.1.3 Functional Requirements

REQ-1: The system shall allow the user to select a valid directory path for monitoring. REQ-2: The system shall detect and log file creation, deletion, modification, and renaming events. REQ-3: The system shall log each event with the file name, full absolute path, event type, and ISO 8601 timestamp. REQ-4: The system shall display each event in a scrollable GUI table in real time. REQ-5: The system shall support filtering monitored events by file extension specified before monitoring begins. REQ-6: The system shall terminate monitoring only upon user command or application exit. REQ-7: The system shall ignore subdirectory events in the initial release. REQ-8: If the selected directory is invalid or inaccessible, the system shall display an error message.

3.2 Save Events to Database

3.2.1 Description and Priority

This feature allows the user to persist all logged events to a local SQLite database. It guarantees that monitoring data can be stored, reviewed, and queried in future sessions. The feature is categorized as High priority. The benefit is rated 8, the penalty of absence is rated 7, the implementation cost is rated 3, and the risk is rated 2.

3.2.2 Stimulus/Response Sequences

After monitoring events, the user clicks the "Save to Database" button. If no database file exists, the system creates a new file in a predefined writable location. If the file exists, the system appends the new log data. Upon successful save, the system provides a confirmation message. If the operation fails, the system notifies the user with a dialog.

3.2.3 Functional Requirements

REQ-9: The system shall create a local SQLite database if none exists. REQ-10: The system shall write each logged event into a persistent table with fields for name, path, event type, and timestamp. REQ-11: The system shall append new data if the database already exists. REQ-12: The system shall inform the user of success or failure following a save attempt. REQ-13: The system shall store the database in a location that supports user-level write access.

3.3 Query Logged Events

3.1.3 Description and Priority

This feature allows users to retrieve historical file events from the local database using filters such as file extension, event type, and time range. It provides meaningful insights from past data and is considered medium priority. The benefit is rated 7, penalty is rated 4, cost is rated 5, and risk is rated 4.

3.2.3 Stimulus/Response Sequences

The user clicks the "Query Database" button. A new window appears with input fields for file extension, event type, start time, and end time. The user enters filter criteria and submits the query. The system returns matching results from the database and displays them in a tabular view. While in query mode, monitoring controls are disabled.

3.3.3 Functional Requirements

REQ-14: The system shall provide an interface for users to apply filters and submit queries. REQ-15: The system shall return and display events that match all provided filter criteria. REQ-16: The system shall prevent monitoring while the query interface is active. REQ-17: The system shall limit query result sets to 10,000 records to maintain performance. REQ-18: The system shall allow users to exit the query mode and return to monitoring or idle states.

4. External Interface Requirements

4.1 User Interfaces

The system includes a graphical user interface that may be implemented using Python-based frameworks such as Tkinter or PyQt. It provides a toolbar with buttons labeled Start Monitoring, Stop Monitoring, Save to Database, Query Database, and Clear Database. A status bar indicates the current operational state. A main display area presents event logs in a scrollable table with headers for file name, path, event type, and timestamp. Dialogs are used for folder selection, error handling, and save confirmation. The GUI disables unavailable controls based on state. Menu options labeled Help, About, and Exit are included in the main menu strip.

4.2 Hardware Interfaces

The software does not interact with or depend on any specialized hardware. It is designed to operate on standard desktop computers with x64 architecture. It supports storage devices formatted with NTFS, APFS, FAT32, or HFS+. Input is provided using a mouse and keyboard. No touchscreen or external peripheral support is required.

4.3 Software Interfaces

The system may run using a Python interpreter and may use the watchdog library for file monitoring. SQLite access is performed using the sqlite3 module or SQLAlchemy. GUI rendering and user interactions are handled using Python frameworks such as Tkinter or PyQt. Data is passed internally between UI components and logic handlers. The system uses structured SQL queries to retrieve and save data. No external APIs or services are integrated.

4.4 Communications Interfaces

The application does not perform or require any network communications. It does not send data over HTTP, FTP, or any other protocol. It does not access remote servers or cloud services. All operations are performed locally on the user's machine. No encryption, synchronization, or message formatting is necessary for this version.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

The File Watcher system is expected to respond to file events almost instantaneously. Events such as creation, deletion, renaming, or modification of files should be detected and logged within a 2-second window from the moment the system receives the notification. The application must maintain a responsive user interface at all times, even when processing high-frequency file system *events. The software should also be capable of handling bursts of up to 100 file events per second without freezing or failing. All databases write operations (such as saving logged events) should be completed within 1 second of initiation. When querying the database, results for standard filters (e.g., by file extension or date) must return within 3 seconds for datasets up to 10,000 records. The software must remain operational for extended monitoring sessions (at least 8 hours continuously) without memory leaks or degradation in performance.*

5.2 Safety Requirements

The File Watcher software does not perform any write or delete operations on user files; it operates purely in a read-only monitoring mode. However, there are safety concerns associated with potentially unexpected crashes or misinterpreted file paths. Therefore, the application must include safeguards such as validating all input paths before monitoring and catching all unhandled exceptions to prevent crashes. If the database has not yet been written at the time of application closure, the user must be prompted with a dialog asking whether to save the currently logged information. To prevent user errors, the "Clear Database" feature must include a confirmation dialog. The system will never attempt to open, modify, or interact with any files beyond registering their events. External dependencies like SQLite.NET must be verified secure and kept up to date to prevent vulnerabilities related to unsafe data access.

5.3 Security Requirements

While the application is strictly local and does not interface with any network components, it still requires attention to file and data security. The SQLite database file, which logs monitored events, must be saved in a location that honors system-level user access controls, such as the user's Documents or AppData directory. The application must never transmit data externally. Only the user who initiated monitoring and wrote the data to the database should have access to that database file by default. The application should not run with administrative privileges unless explicitly required. Since this software does not include user authentication, its access model is entirely dependent on the operating system's user session. The application must prevent command injection, script execution, or dynamic loading of files beyond its scope. Optional read-only database access may be provided in a future release to support audit use cases.

5.4 Software Quality Attributes

The File Watcher software is designed with reliability and usability as top priorities. It must operate consistently across supported environments, including Windows 10+ and macOS Monterey and later using .NET 6+. The user interface must clearly indicate system status (monitoring/not monitoring) and visually disable unavailable controls to guide user behavior and avoid mis-operations. Error messages must be user-friendly and specific. From a maintainability perspective, the application code must be structured using a modular design, separating concerns between the UI, file monitoring logic, and database handling. This separation allows for future enhancements (e.g., CSV export, cloud sync) to be added without refactoring the entire system. Testability is built-in via the use of logging to both UI and database, allowing easy verification of output correctness. Portability is supported on macOS via frameworks such as .NET MAUI or Avalonia UI. Overall, this software favors extensibility and reliability over feature complexity, which is in line with its primary purpose, dependable file system change tracking.

6. Other Requirements

The File Watcher system must support extensive logging to a lightweight, embedded SQLite database. Each entry in the database must include: the name of the file, the absolute path, the type of event (create, delete, modify, rename), and a timestamp in ISO 8601 format. While the software does not need to support internationalization for its interface in the current release, it must properly handle and display file paths and filenames containing UTF-8 characters, such as those in non-English languages (e.g., Japanese, Russian, Arabic). All time data must be consistent across time zones and saved using a 24-hour format. The application must include a Help to About menu that lists the program's usage instructions, the current version, and the developer's name. There are currently no legal or regulatory constraints on the software; however, the software must respect user permissions and never monitor system directories without explicit permission. The reuse objective is met through modular design: components such as the 'FileMonitorController' and 'DatabaseHandler' are implemented as reusable classes that could be embedded into other desktop monitoring utilities or server-side file logging tools. The only required third party library is SQLite.NET, which must be bundled with the application in a redistributable format for both macOS and Windows environments.

Appendix A: Glossary

*This glossary defines terms and concepts essential for understanding the File Watcher system. These definitions support clarity across stakeholders, especially for those without prior experience working with file system monitors or embedded databases. The **File Watcher** refers to the name of the desktop application developed as part of this project. It allows users to observe and log changes to the file system in real time, targeting specific file types based on user defined extensions. The application uses the **FileSystemWatcher** class in .NET (or equivalent wrappers in cross platform frameworks) to detect file events, including creation, deletion, renaming, and modification. These events are logged and, optionally, saved to a local **SQLite** database for later querying. An **event type** describes the nature of a file system change. For this system, the core event types are: Created, Deleted, Changed, and Renamed. A **monitoring path** refers to the folder selected by the user where these file system events will be watched. The monitoring scope can be either shallow (non-recursive) or deep (including all subdirectories). A **file extension** is the suffix at the end of a file name indicating its format or type (e.g., .txt, .md, .mov). Users may select from a preset list or define custom extensions to watch. The **database file** is a local .db file in SQLite format that stores event logs with fields for file name, path, event type, and timestamp. The **query window** is a GUI component that provides filters and search functionality, allowing users to find specific events from the database history. Supporting this are GUI elements like a **menu strip** (housing dropdowns for File, Help, and About) and a **tool strip** (with clickable buttons for Start, Stop, Save, and Query actions).While initially developed with Windows and .NET in mind, the File Watcher system also supports **macOS** through .NET 6+ compatibility and file system APIs available in macOS Monterey and later. For Mac users, the application may be run natively via Visual Studio for Mac or built using cross-platform frameworks like Avalonia UI or .NET MAUI, ensuring UI and file system access behaves consistently across platforms. Lastly, terms like **TBD** ("To Be Determined") and **read-only mode** refer to ongoing development elements and optional features that may be implemented in future releases (e.g., enabling users to view the database without making modifications).*

Appendix B: Analysis Models

*The File Watcher system follows a modular design that separates core responsibilities into three logical components: user interface, file monitoring, and data persistence. These are implemented via MainForm, FileMonitorController, and DatabaseHandler classes; This separation allows for more maintainable code and easier unit testing, particularly when adapting the application for different operating systems like macOS. From a use case perspective, the application supports the following core user interactions: starting the monitoring of a selected folder and file extension, stopping that monitoring process, saving the current event log to a database, querying past events from the database, clearing the database with confirmation, and exiting the application safely. Each of these actions transitions the system between defined internal states, typically from an idle state to an active one and then back. For instance, the system transitions from **Idle to Monitoring** when the user clicks "Start Monitoring," and from **Monitoring to Idle** when the user clicks "Stop." If the user chooses to save data, the state moves temporarily to **Writing to Database** and then returns to idle. During querying, the state becomes **Query Mode**, where the main interface is disabled until the user returns. These transitions are clearly controlled and reflected in the UI, so that the system cannot enter invalid or overlapping states. Diagrams such as class diagrams and state-transition models are currently under development. The class diagram will illustrate the relationship between the core logic classes and event-handling mechanisms, while the state-transition diagram will visualize the application's behavior during various user actions. Both models will be finalized and included prior to project submission.*

Appendix C: Issues List

*Several implementation and design considerations remain open as the File Watcher system approaches completion. These items reflect features under development, planned enhancements, or unresolved decisions that may affect the final scope of the project. One of the current development goals is to implement a **CSV export feature** in the Query Window which allows users to export filtered results for use in spreadsheet tools or reporting. The internal logic is complete, but the file dialog integration for both macOS and Windows is undergoing testing. Similarly, the **iconography and visual styling** of the toolbar is being revised to provide platform-consistent visuals; macOS users will see native-style icons and menu alignment appropriate for Aqua/HIG standards. Another ongoing task is the improvement of **query filtering logic**. At present, the user can search based on file extension and timestamp. Additional filters for event type and full-text file path search are planned for inclusion in the final release. Moreover, handling edge cases such as **rename operations across volumes** (i.e., between different drives or partitions) has proven complex due to differing file system behaviors on macOS versus Windows. These cases are under evaluation and unit tests are being developed to handle them. The **Help and About menu** functionality are complete, but the optional inclusion of a downloadable PDF manual is still pending a decision on packaging and distribution. Finally, while multidirectory support and concurrent monitoring sessions have been requested, they are deferred to a potential v2.0 release due to increased architectural complexity. This issues list will remain dynamic and will be updated continuously until the final deployment and submission milestone. It serves both as a project management tool and a roadmap for future improvements beyond the current academic scope.*