Basic File Management System — Fast, Minimal CLI for Everyday Files

Streamlined Python CLI for listing, creating, reading, writing, copying, moving, renaming, deleting, and searching files/directories; focuses on simplicity, responsiveness, and safety.

SAHIL CHOUSKEY

Interactive Terminal for Faster File Management

Lightweight, menu-driven CLI that simplifies common filesystem tasks and teaches filesystem APIs

1. Lightweight, interactive terminal utility for filesystem tasks
2. Supports listing, creating, reading, writing, copying, moving, renaming, deleting, searching
3. Menu-driven CLI improves discoverability and responsiveness
4. Designed for faster ad-hoc file management without GUI overhead
5. Also supports educational goals by exposing filesystem API usage
6. Makes routine tasks simpler for developers and learners

Fast, Low-overhead CLI for Common File Tasks

Lightweight, cross-platform terminal tool to simplify everyday filesystem operations for scripting and quick tasks

1 Users need a compact, cross-platform command-line tool for quick filesystem operations

2 Full-featured file managers or GUIs are overkill for scripting and simple tasks

3 Goal: simplify common file and directory management directly from the terminal

4 Benefit: efficient, straightforward interactions without GUI complexity or heavy resource use

Functional Requirements for CLI File Manager

Menu-driven cross-platform file operations with metadata preservation

Menu-driven interface for selecting filesystem tasks

List directory contents with type and size details

Display metadata including size and timestamps**

Create files and directories with automatic parent creation

Read, overwrite, and append file contents

Rename, copy (preserve metadata), move files and directories

Delete with confirmation prompts before removal

Recursive search using glob patterns

Consistent behavior across Windows, macOS, Linux

Non-Functional Requirements: Usability, Safety, Maintainability

Cross-platform, minimal-dependency CLI with safe defaults and testable design

Cross-platform: rely solely on Python standard library to minimize dependencies and ensure portability

Usability: interactive CLI with precise prompts and clear confirmation dialogs for responsive, readable workflows

Safety: safe defaults and confirmations for potentially destructive actions to reduce user errors and data loss

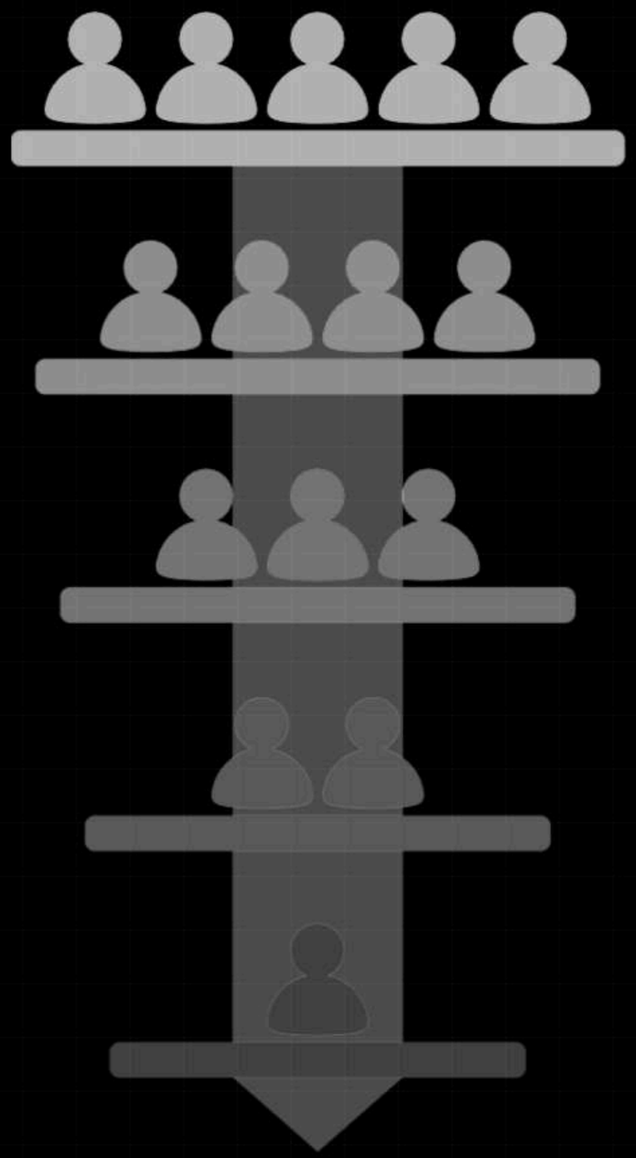
Maintainability: designed for extensibility and ease of testing; clear interfaces and modular code organization

Responsiveness & Clarity: prioritize prompt feedback and unambiguous messages to support fast, safe operations

Alignment with CLI best practices: minimal dependencies, predictable defaults, explicit confirmations

System Architecture — Single-Process CLI

Layered responsibilities from UI to Utilities with main entry script 'cse projeect.py' and optional test/docs folders



1

UI Layer

Controls menu loop, user prompts, and screen clearing.

2

Controller

Maps menu choices to operations and routes commands.

3

Operations Layer

Performs filesystem actions: list, read, write, copy, move, delete, search.

4

Utilities

Supports path prompting, formatting, and timestamp display.

5

Entry Script

Primary script 'cse projeect.py' is the main entry; optional tests and docs directories.

Design Diagrams Overview

Visual models that clarify structure, behavior, and component responsibilities

Behavioral Diagrams

Use Case Diagram — identifies **User** actor and core operations: listing, info display, creation, reading/writing, copying/moving, renaming, deleting, searching

Workflow Diagram — maps interaction flow from menu display through operation execution to result presentation

Sequence Diagram — details message exchanges among **User**, **UI**, **Controller**, operations, and **Filesystem**

Structural Diagrams

Component Diagram — maps major parts: **UI**, **Controller**, **FileOps**, **Utils**

FileRecord (optional) — simple entity for future logging when persistent storage is added; no ER diagram currently applicable

Purpose & Value

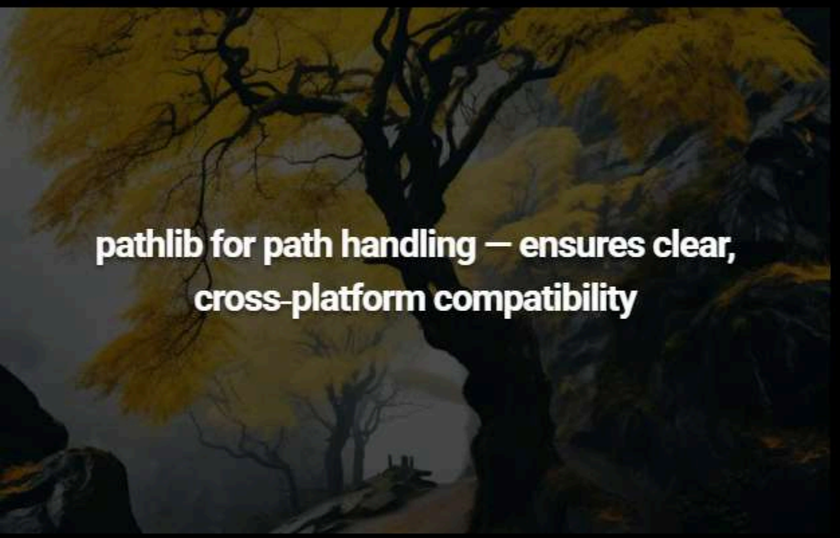
Clarify responsibilities and interfaces across UI, controller, and file operations

Expose message flow and error/response paths for testing and implementation

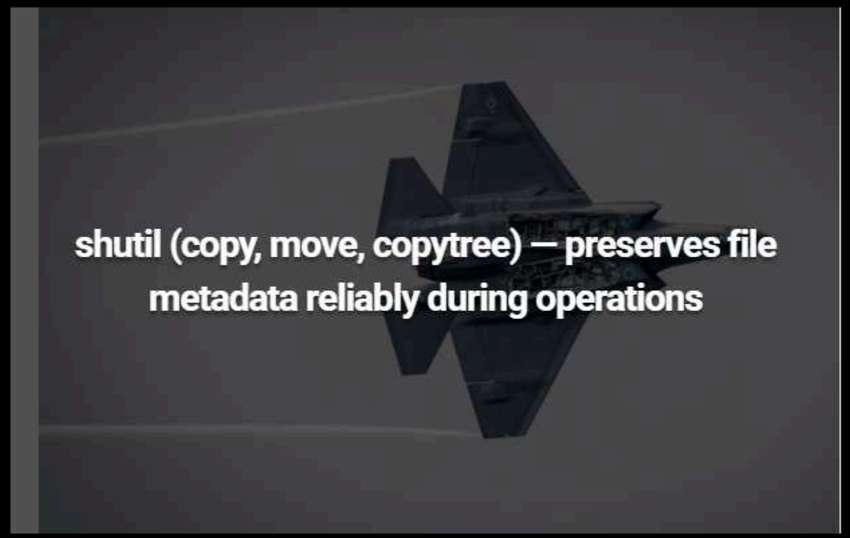
Provide a roadmap for adding persistent logging later via **FileRecord**

Design Decisions & Rationale

Why choices prioritize safety, cross-platform reliability, and ease of use

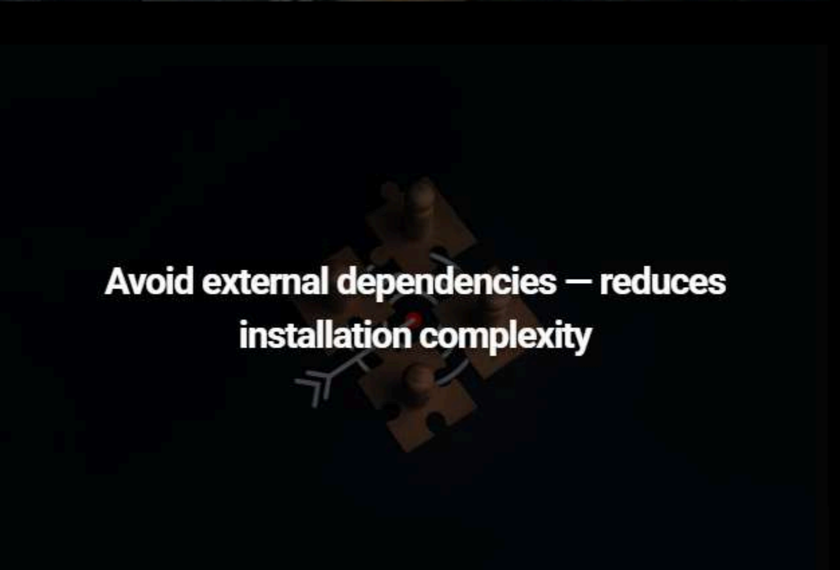


pathlib for path handling — ensures clear, cross-platform compatibility



shutil (copy, move, copytree) — preserves file metadata reliably during operations

Menu-driven CLI — simple, accessible interface for diverse users



Avoid external dependencies — reduces installation complexity

Safe defaults — confirmation prompts before deletion to prevent data loss

Implementation Details: File Manager Core

Key functions, libraries used, and critical fixes for robust CLI behavior

Main loop dispatches user menu to dedicated functions: list_dir(), show_info(), create_file(), read_file(), write_file(), append_file(), delete_path(), rename_path(), copy_path(), move_path(), make_dir(), search_files()

Libraries leveraged: pathlib, shutil, glob, datetime, os

Search refined: use recursive globbing and corrected pattern handling for search_files()

Entry point fixed to if __name__ == '__main__': to ensure correct module execution

File naming reviewed for clarity and safety; conventions clarified to avoid unsafe names

Screenshots & Results: CLI Workflow Snapshot

Key interactions, outputs, and saved visual evidence for verification and learning

1. **Main menu** view showing primary commands and navigation options
2. **Directory listings** output illustrating file permissions, sizes, and timestamps
3. **File creation & reading** dialogs demonstrating input prompts and file content display
4. **Copy & move** operations with confirmation prompts captured for UX verification
5. Screenshots saved under **/docs/screenshots** and referenced in the deck

Testing Approach — Manual & Automated

Practical scenarios, automated pytest guidance, and key risk areas

Manual testing: create nested directories/files; write, read, append; copy, move, delete with confirmation prompts

Automated testing: use pytest with temporary directory fixtures to simulate filesystem safely

Core test scenarios: file creation, writing, appending, deletion (include **dry-run** checks), copying files and directories

Edge & risk areas: permission errors, non-existent paths, overwrite conflicts between files and directories

Recommendation: combine manual verification for UX prompts with pytest for repeatable, isolated checks

Key Development Challenges

Concise explanations of technical and UX complexities

Cross-platform paths — handling OS path differences and timestamp semantics across environments

Safe deletion semantics — distinguishing recursive vs non-recursive removals to prevent data loss

CLI multi-line input — capturing robust multi-line user input while keeping parsing and usability reliable

Destructive prompts UX — clear, recoverable confirmations and error flows for destructive actions

Key Learnings from the Filesystem Tooling Project

Practical lessons for cross-platform file handling, safe CLIs, and teaching-focused tooling

pathlib simplifies cross-platform file code by abstracting OS differences

shutil handles complex file operations and preserves metadata, reducing code complexity

Interactive CLI needs strict **input validation** and confirmation to ensure safety

Small, focused tools effectively streamline developer workflows and teach filesystem APIs

Summary: combine **pathlib**, **shutil**, and robust CLI patterns for safe, maintainable tooling

Future Enhancements to Improve Safety, Debuggability, and UX

Planned improvements with brief rationale for each

Undo / transaction via trash — recover from mistakes by staging deletions in a trash-based system

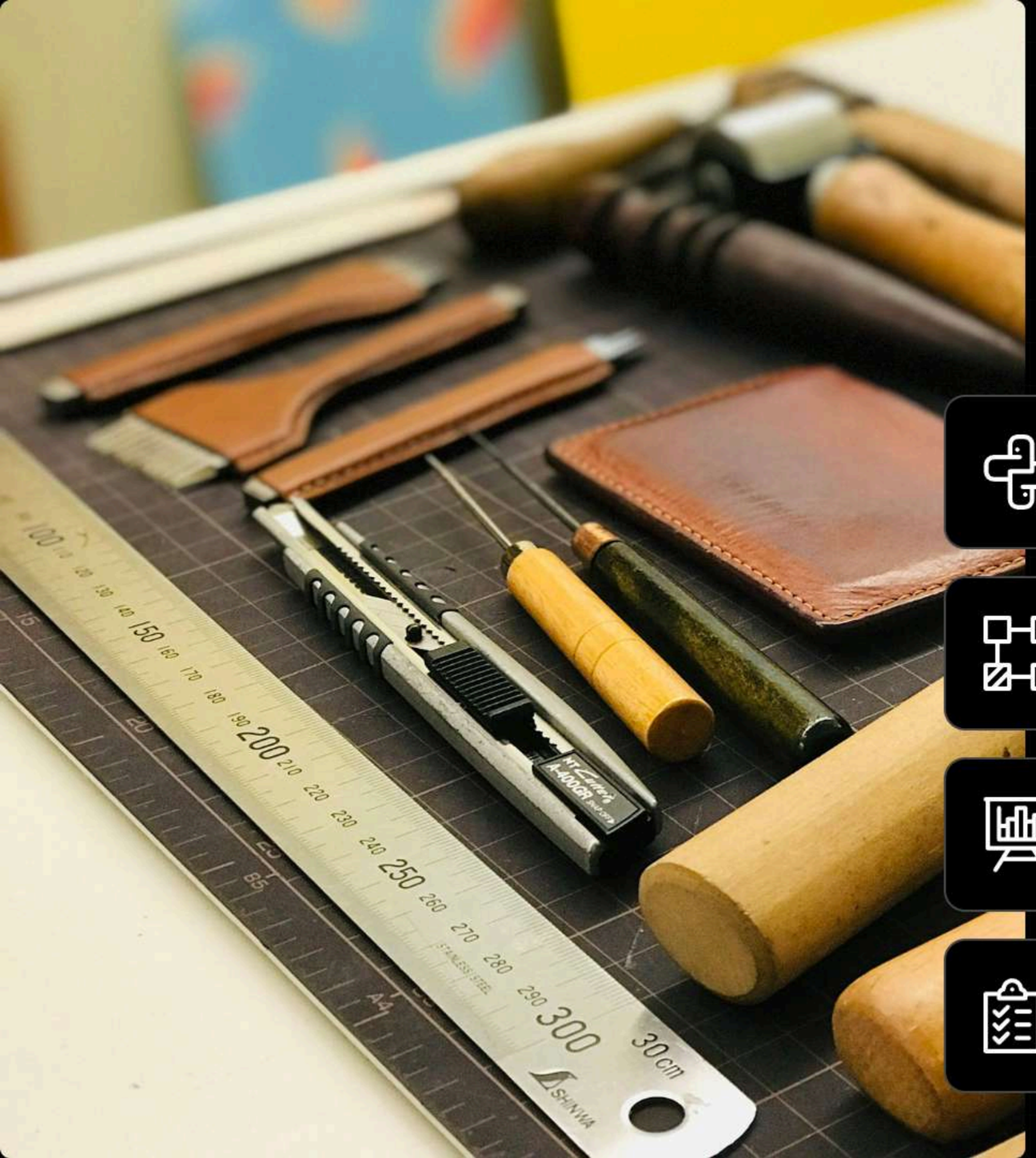
Logging & verbose mode — improve debugging and usage transparency with detailed logs

Dry-run mode — preview risky operations without making changes to avoid unintended effects

Optional JSON/YAML config — let users set defaults like base directory and preferred editor

Text-based UI (TUI) — enhance usability with curses/textual-based interactive interface

Expand unit tests & add CI — bolster code quality through broader tests and continuous integration



Key References for Filesystem & CLI Tooling

Authoritative docs and tools to implement filesystem utilities and improve CLI design

pathlib, **shutil**, **glob**, **os** — official Python documentation for comprehensive API details

argparse, **click** — CLI design and scriptability best practices and libraries

Diagramming: **PlantUML**, **draw.io** — visual design and architecture diagrams

Use these references for robust filesystem utilities and clear CLI interfaces