ChatGPT

Rust Storage Cleanup Tool – Development Plan

The goal is a Rust-based disk analysis/cleanup utility with both a CLI and a Tauri GUI. It should scan project directories recursively, identify large files and common build/cache artifacts (like Rust `target/`, Node's `node_modules/`, etc.), and generate a human-editable cleanup plan (in TOML). The plan lists actions (Delete, Keep, Review, later Archive) per item. Before executing any deletions, the tool re-verifies each file's size/mtime (and optionally its content hash) to detect any drift from the original scan. By default deletions are permanent, with an **optional** Recycle-Bin mode (using e.g. trash-rs on Windows/Linux). NTFS-specific optimizations (like bulk-rename-then-delete tricks) and archiving flows will be layered in later MVPs. We ensure the architecture cleanly abstracts OS differences (Win vs Linux) so Linux support can be added after initial Windows focus.

Milestone 1: Directory Scan & Plan Generation

Goals/Deliverables: A CLI prototype that recursively scans a given directory and identifies "cleanup candidates" (by size, patterns, etc.), and outputs a **TOML plan file**. The TOML plan lists each candidate file or directory and a default action (Delete/Keep/Review). Users can edit this file manually before applying it.

Key Modules/Components:
- **File Traversal Module:** Use a crate like `walkdir` or `ignore` to walk the directory tree. (The Fast Delete tool uses *walkdir* for traversal [1].)
- **Artifact Detection Module:** Heuristics to flag large items: e.g. files exceeding a size threshold, or known build artifact directories (`target/`, `node_modules/`, `build/`, etc.). We may borrow ideas from existing tools like *cargo-wipe* (which looks for Rust `target` markers) and *Tin Summer* (which finds build artifacts) [2] [3]. For MVP1, focus on simple rules (file size, and a few common folder names).
- **Plan File Module:** Represent the cleanup plan in a TOML structure. Use a TOML library to generate a file that lists each path, its size, mtime, and a default "action = delete/keep/review".

Prototype/Spike: Test the scanner on large sample directories to measure performance and memory. Prototype writing/reading the TOML plan; verify that users can manually tweak it and reload it. (Also decide on format: e.g. one table per item, or list of tables.)

Milestone 2: Plan Verification & Deletion (CLI)

Goals/Deliverables: Extend the CLI so it can **apply** a plan. On the "apply" command, the tool reads the TOML plan, first **verifies no drift** (by checking each file's current size and mtime against what's recorded in the plan). If any file changed, warn the user and halt. If all is consistent, perform the deletions or other actions as specified. For now implement **single-threaded deletion** (one file at a time) using `std::fs::remove_file` or `remove_dir` for permanent deletions.

Key Modules/Components:
- **Plan Parser:** Load the TOML plan into Rust structs.

- **Drift Detector:** Compare recorded metadata (size, mtime) with the on-disk state. (Optionally, compute a fast hash (e.g. BLAKE3 [4]) if the user requests stronger verification.) BLAKE3 is extremely fast and parallelizable [4], making it feasible to hash large files if needed.
- **Deletion Module:** Perform the actual delete operations. For permanent deletion, on Windows this is the `DeleteFileW`/`RemoveDirectoryW` WinAPI (via Rust's `std::fs`). For later Recycle-Bin mode, we'll use a crate like *trash-rs* [5] or call `SHFileOperation` with `FOF_ALLOWUNDO` to send files to the Recycle Bin [6].
- **Logging/Confirmation:** Summarize actions and confirm with user.

Prototype/Spike: Create a test plan file manually and modify a listed file (change its contents or mtime) after scanning; ensure the tool detects drift and aborts. Verify deletion works on a mix of files/directories. Initially, ignore performance; focus on correctness.

Milestone 3: Performance & Concurrency Enhancements

Goals/Deliverables: Improve speed by **multithreading** and add options like Recycle-Bin mode. The CLI now should delete files in parallel (e.g. using rayon or a thread pool) and show progress. Also add command-line flags (`--dry-run`, `--use-recycle`, etc.) and ensure the tool can handle very large file sets efficiently.

Key Modules/Components:
- **Parallel Execution:** Integrate a thread pool (e.g. using *rayon*) so that file deletions (and possibly scanning) run concurrently. For example, the Fast Delete tool uses a rayon threadpool to delete files in parallel and reports progress [7]. We will similarly use *indicatif* for a progress bar and *num_cpus* to size the pool [1].
- **Cross-Platform Abstraction:** Encapsulate OS-specific logic. For example, define a trait or utility functions for "delete_file(path, permanent: bool)". On Windows, permanent delete uses `std::fs::remove_file` (WinAPI `DeleteFile`), whereas `--use-recycle` uses the *trash-rs* crate or `SHFileOperation` (with `FOF_ALLOWUNDO`) [6] [5]. On Linux, permanent delete is just `unlink`, and recycle-bin could call `gio-trash` or similar. This abstraction layer ensures our core logic is mostly OS-agnostic.
- **Progress Reporting:** Show a real-time progress bar or counter in the CLI (using *indicatif*). Ensure the GUI can hook into this progress reporting later.

Prototype/Spike: Benchmark deletion on a large directory. Compare single-thread vs multithread. Test `--use-recycle` on Windows (using *trash-rs* library [5]). Ensure parallel deletion doesn't violate any filesystem constraints (e.g. not deleting parent directory while children threads are active).

Milestone 4: Tauri GUI Integration

Goals/Deliverables: Build the Tauri-based graphical interface *in parallel* with CLI. The GUI should allow users to select directories, trigger a scan (updating the TOML plan), review/edit the plan in a user-friendly way (e.g. table of items with actions), and apply the plan with progress feedback. The GUI can lag by one milestone, but should follow the same core logic.

Key Modules/Components:
- **Tauri Frontend:** A web-based UI (e.g. built with a JavaScript framework) that interacts with the Rust backend. The UI will display scan results, allow editing actions (perhaps editing the TOML text or using form

controls), and show progress/status. It should disable applying the plan if drift is detected.
- **Tauri Backend Commands:** In Rust, define Tauri commands (via `#[tauri::command]`) for actions like `scanDirectory(path)`, `applyPlan(path)`, `getProgress()`, etc. These commands call into the same Rust modules built for the CLI. Use message-passing/events to stream progress updates to the UI (Tauri supports emitting events from Rust that JS can listen to).
- **Cross-Platform UI:** Tauri uses the OS's native WebView (Edge WebView2 on Windows, WebKit on Linux/macOS) [8]. Notably, WebView2 is *preinstalled on Windows 11* [9], so the user shouldn't need to install anything extra. Because Tauri apps compile to a Rust binary, the app will be small and performant [10] [11].
- **UI/UX Considerations:** Keep the GUI simple (maybe a file-tree view or list with checkboxes). Ensure the plan file remains editable (or represent changes made in UI back into the TOML).

Figure: Simplified Tauri architecture combining a WebView-based frontend with a Rust backend [10]*. The Rust core (file I/O, scanning, deletion) is exposed via commands, and the WebView (UI) invokes them via message passing.* In practice, Tauri allows any JavaScript framework for the UI and communicates with Rust through secure, asynchronous commands [10] [11]. We will wire the scanning/plan logic into Tauri commands so the GUI can run a scan, display results, and kick off deletions. The UI will show a progress bar (e.g. `<progress>` element) updated by backend events.

Prototype/Spike: Create a minimal Tauri app with a button that calls a Rust command (`scanDirectory`) and displays output. Test that Rust's async tasks don't block the WebView. Ensure on Windows the WebView2 renderer works (it's built-in on Win11 [9]). Establish file dialogs for directory selection.

Milestone 5: Advanced Features & Cross-Platform Prep

Goals/Deliverables: Add remaining MVP features and prepare for future Linux support. This includes NTFS-specific performance tricks and an "Archive" action. Also verify that core functionality works on Linux (path separators, APIs, etc.) even if Linux GUI is not yet finalized.

Key Modules/Components:
- **NTFS Optimizations (Future MVP):** Research and optionally implement faster deletion strategies on Windows 11's NTFS. For example, one can sometimes speed up bulk deletion by renaming files/directories first or using low-level APIs (`SetFileInformationByHandle` with `FileDispositionInfoEx`) to delete without extra journaling. These are advanced optimizations we'll spike on (they are Windows-specific).
- **Archive Action:** Introduce a new plan action `Archive`. When chosen, the tool will compress the target file/folder (e.g. into a ZIP) and then delete the original. Implement using a compression crate (like `zip` or `tar`+gzip) and test file integrity.
- **Cross-Platform Abstractions:** Ensure the codebase compiles cleanly on Linux/macOS. Abstract away any Windows-only bits behind `#[cfg(windows)]`. For example, on Linux the "delete" action is just `std::fs::remove`, and an `--use-recycle` option could call `gio-trash`. The UI (Tauri) already works cross-platform, but verify on a Linux machine that file system operations behave as expected.
- **Packaging/Distribution:** Add build scripts or CI to package the app (for Windows maybe an MSI or EXE, for Linux an AppImage or DEB in future). Use Tauri's bundler which supports Windows, macOS, Linux [11].

Prototype/Spike: On Windows 11, experiment with `SetFileInformationByHandle` (Rust's `windows` crate or FFI) to do fast deletes, and measure any speed gain. Test the archive process on large folders. On Linux, try the CLI to confirm it lists files and deletes properly (using Rust's standard library calls).

Throughout all milestones, we maintain a **layered design**: a core Rust library for scanning/planning/deletion, with a CLI front-end and a Tauri GUI that both call into it. Windows-specific concerns (Recycle Bin API, NTFS flags) are behind a thin OS abstraction layer. By milestone 5 we will have a working Windows 11 tool with fast, parallel operations and a responsive GUI, and a clear path to adding Linux support (simply by implementing the OS layer for Linux).

Sources: We draw on examples of existing tools and documentation: e.g., the Fast Delete project demonstrates multithreaded deletion with Rayon [7] and progress reporting [1]. Windows APIs like `SHFileOperation` are documented by Microsoft (deleting with `FOF_ALLOWUNDO` sends files to Recycle Bin [6]). The trash-rs crate abstracts sending files to the Recycle Bin/Trash [5]. Modern hash functions like BLAKE3 can verify file contents very quickly [4]. And Tauri's documentation shows it uses the OS WebView (Edge WebView2 on Win) for a lightweight cross-platform GUI [8] [10].

---

[1] [7] GitHub - Himasnhu-AT/frm: Faster file deletion written in Rust, alternative to rm
https://github.com/Himasnhu-AT/frm

[2] [3] Cleaning up large Rust binaries in target directories
https://sts10.github.io/2021/10/26/rust-target-directory-clean-up.html

[4] GitHub - BLAKE3-team/BLAKE3: the official Rust and C implementations of the BLAKE3 cryptographic hash function
https://github.com/BLAKE3-team/BLAKE3

[5] GitHub - Byron/trash-rs: A Rust library for moving files to the Recycle Bin
https://github.com/Byron/trash-rs

[6] SHFileOperationA function (shellapi.h) - Win32 apps | Microsoft Learn
https://learn.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shfileoperationa

[8] [9] Webview Versions | Tauri
https://v2.tauri.app/reference/webview-versions/

[10] Tauri Architecture | Tauri
https://v2.tauri.app/concept/architecture/

[11] Tauri 2.0 | Tauri
https://v2.tauri.app/