ChatGPT

Designing a High-Performance Storage Analysis and Cleanup Tool (Rust + Tauri)

Abstract:
This report presents research-backed guidance for building a Rust-based disk space analysis and cleanup tool with a Tauri GUI. We address high-performance file scanning and deletion techniques on Windows 11 (NTFS), Rust concurrency patterns for processing large file sets, snapshot-based drift detection and verification, user experience (UX) design patterns for safe storage cleanup, and strategies for archiving large project directories. Each section compiles best practices from industry tools and literature, with emphasis on implementation details and design principles, to inform a robust and intuitive application architecture.

1. High-Performance Directory Scanning & Deletion on Windows 11 (NTFS)

NTFS Enumeration Speed: Modern disk analyzers like WizTree achieve **dramatic speedups** by bypassing traditional Win32 file APIs and reading the NTFS Master File Table (MFT) directly [1] [2]. The MFT is a metadata index of all files on an NTFS volume. By scanning this structure at the disk level, WizTree can list every file's name, size, and timestamps in seconds, even on terabyte drives [2]. This approach avoids the overhead of opening every directory and file via `FindFirstFile`/`FindNextFile` calls. Competing tools (WinDirStat, TreeSize) typically rely on standard OS file enumeration, which incurs a **system call per file** and extra layers of indirection [3]. As a result, WizTree is reported to be 20–50× faster on NTFS volumes than WinDirStat [4] [5]. For our tool, a similar strategy could be considered: on NTFS, use a native-binding (via the `windows` crate or FFI) to parse MFT records for maximum scan throughput (noting this may require administrative rights and careful parsing of NTFS structures).

Using Win32 APIs Efficiently: If MFT parsing is not feasible, we must optimize the Win32 enumeration calls. Use `FindFirstFileExW` with `FindExInfoBasic` to retrieve only essential attributes (skipping 8.3 name retrieval) and consider the `FIND_FIRST_EX_LARGE_FETCH` flag to request larger directory batches from the OS [6]. In practice, `FIND_FIRST_EX_LARGE_FETCH` sometimes yields modest improvements (it hints the kernel to fetch more entries per call) [6], though one analysis found it did not significantly alter the CPU cost per file [7]. The fundamental bottleneck is that each file still requires a kernel query (e.g. `NtQueryDirectoryFile`), and a fast NVMe SSD can end up **CPU-bound on a single thread** during enumeration [8]. Indeed, a test enumerating ~2 million files showed one logical core pegged at 100% with synchronous traversal [8]. To overcome this, **multi-threaded directory traversal** is crucial (discussed below). Additionally, we should disable or mitigate Windows features that add overhead per file. For example, if "8.3" filename generation is enabled on the volume, turn it off during scans if possible, as it can double the work of each file enumeration [3]. It's also wise to disable retrieval of Access Control Lists (ACLs) or other metadata unless needed, since those may trigger additional lookups.

Multi-Threaded and Overlapped I/O: A key performance win is to enumerate different subdirectories in parallel. By spawning concurrent tasks for independent directory trees, we can leverage multiple CPU cores and hide I/O latency [9] [10]. In C++ experiments, using `std::async` to walk subdirectories concurrently nearly halved overall scan time on an 8-core system (scanning 2 million files dropped from ~1.5 seconds single-threaded to ~0.77 seconds with parallel futures) [8] [11]. In Rust, one can achieve this with a thread pool (e.g. Rayon's work-stealing pool or custom threads). The idea is to issue multiple `FindFirstFileEx/FindNextFile` streams simultaneously on different directories. Windows' NTFS driver can handle many concurrent enumeration calls; on SSDs and NVMe, this parallelism helps saturate the device's I/O queues. Careful design is needed to avoid oversubscription: enumerating *too* many directories at once could flood the I/O channel or thrash the disk head on HDDs. A good strategy is a **breadth-first traversal** with a work queue: pop a directory, list its entries, and push subdirectories onto a thread pool or async task queue. The Rust `ignore` crate (used in ripgrep) and community tools like `jwalk` implement this pattern, showing 3×–4× speedups over naive single-thread walks [12] [13]. Notably, `jwalk` confines parallelism to the directory level (it won't speed up a single dir with millions of files, but will handle many dirs in parallel) [14], which aligns with NTFS behavior that already returns a directory's entries as a batch.

Deletion Performance: Deleting large numbers of files can also be optimized. On Windows, GUI deletions are slow partly because Explorer calculates file counts and shows progress, and also because by default it uses the Recycle Bin (which involves extra overhead per file). For maximal performance, *bypass the Recycle Bin* (use direct deletion) for bulk operations. The fastest method is typically the command-line approach (`DEL /F /Q /S`), which deletes files in bulk without UI updates [15] [16]. Our tool can emulate this by calling `DeleteFileW`/`RemoveDirectoryW` on each item in a tight loop. **Batch deletion strategies:** One approach is to perform deletions on multiple threads – e.g., a thread pool that pulls file paths from a queue and calls `DeleteFileW`. This can improve throughput on SSDs by parallelizing the file removal (each deletion updates the NTFS metadata and is I/O-bound). However, with millions of tiny files, the process can still bottleneck on NTFS serialization of metadata updates (MFT modifications). In practice, tools like *RapidDelete* achieve faster deletions by multi-threading and forgoing additional checks [17] [18]. Another trick is **"quarantine-rename then delete"**: first rename or move the target files into a single "to-be-deleted" folder (which is quick, as it's just metadata updates), then delete that folder in one go. Renaming can also shorten long paths (avoiding MAX_PATH issues) and ensure problematic filenames are handled. This strategy ensures that if something goes wrong mid-process, the originals haven't been permanently erased – they could be recovered from the quarantine. Some cleanup utilities rename files with a special extension or move them to a temp directory before deletion as a safety net.

Windows Defender and I/O Overhead: One often-overlooked factor is real-time antivirus scanning. Windows Defender can **dramatically slow down** operations that touch many files. For instance, simply *reading* file metadata or opening files for backup has triggered 10× slowdowns due to Defender scanning each file accessed [19]. When deleting thousands of files, Defender might scan each file upon open (even for deletion) or at least engage in extra work that slows the process. Thus, for performance, advise users to add the target folders to Defender's exclusion list or temporarily disable real-time scanning during cleanup (with appropriate warnings). In testing, disabling Defender's real-time protection reduced a null backup scan from 2.5 minutes to 10 seconds on Windows 10 [19]. Our Rust tool could also use the Windows Defender API to programmatically turn off scanning for the duration of operations (for advanced users in safe scenarios). At minimum, documentation should mention this interaction and suggest exclusions for developer build folders (which are often huge and filled with binaries that AV might inspect).

System Call Batching and Prefetch: At the OS level, NTFS already batches directory reads, but we should use large buffer sizes if interfacing at the NT API layer. Using the NT native `NtQueryDirectoryFile` can retrieve multiple entries per call (the `FIND_FIRST_EX_LARGE_FETCH` flag in the Win32 layer toggles this) [6]. Rust's `windows` crate allows calling these lower-level functions directly for fine-grained control. Another angle is **overlapped/asynchronous I/O**: initiate multiple asynchronous enumerations or deletions at once (using I/O completion ports or the newer `ReadDirectoryChangesW` for monitoring). However, overlapped directory enumeration is not straightforward with `FindFirstFileExW` (which isn't easily used with IOCP). Instead, launching threads might be simpler and just as effective, given the complexity of truly async directory I/O on Windows.

Recommendations for Rust Implementation: For scanning, start with a robust crate like `walkdir` or `ignore` for cross-platform traversal, but on Windows customize it: use `ignore` crate's parallel traversal features or integrate `jwalk` for speed. If ultimate performance is needed, implement a custom traversal using `FindFirstFileExW` with optimal flags (and possibly fall back to raw MFT parsing for NTFS volumes). Ensure to handle path length issues by enabling the Windows long path APIs (preface paths with `\\?\` when needed). For deletion, use simple multi-threading – e.g., spawn N worker threads (tuned to storage device, maybe 4 to 8 for SSD) that each take a portion of the files to delete. If deleting extremely large directories, consider splitting the task: delete files first, then directories (empty dirs can be removed in a second pass), as Windows can sometimes handle deep directory removals better if files are gone. A neat trick is to call `RemoveDirectoryW` on a top-level folder that still contains files – on NTFS this will fail, but on failure one can recursively remove subitems. Alternatively, shell APIs (like `SHFileOperation` with the `FO_DELETE` flag) can delete whole directory trees and might internally optimize some aspects (or at least provide a progress UI), but they may be slower and don't suit a non-blocking CLI tool. For our high-performance design, direct Win32 deletions on multiple threads is the way to go.

Finally, always implement **safety checks**: count files and bytes before and after deletion to ensure all were removed; if any remain, report them. Also consider offering a "dry run" mode for deletion (which our app effectively has via the reviewable plan). Logging each deletion and any errors (sharing violations, etc.) will help users identify if something wasn't deleted. By combining these strategies – fast enumeration (or MFT read), parallel operations, and minimizing external slowdowns – the tool can safely scan huge project folders (1TB+, millions of files) in seconds to a minute, and delete hundreds of thousands of files in a similarly efficient manner (bounded mostly by disk I/O limits).

2. Rust Concurrency Patterns for Large Filesystem Jobs

Performing a filesystem scan or cleanup on directories containing millions of files is a **mixed CPU and I/O-bound task**. Effective concurrency in Rust can dramatically improve throughput by utilizing multiple cores and hiding I/O latency. We explore patterns including work-stealing thread pools, channels with producer-consumer pipelines, and minimizing locking overhead.

Rayon Work-Stealing: The Rayon library provides an ergonomic way to parallelize data processing and recursive algorithms. It uses a fixed-size thread pool and a *work-stealing* scheduler, meaning idle threads steal work from busy ones to balance load [20]. This is ideal for directory traversal: we can spawn tasks for subdirectories without worrying about spawning 1000+ OS threads. For example, using Rayon's `par_iter` or `scope()` we can implement a recursive walk: when visiting a directory, spawn a task for each subdirectory found [9] [10]. Rayon will distribute these tasks across the pool threads, typically using as many threads as CPU cores (configurable). The benefit is twofold: (1) CPU-intensive operations (like computing hashes or analyzing file metadata) scale with cores, and (2) I/O operations (which often block) can be issued in parallel, so while one thread waits on disk, others do useful work. In a forum discussion, it was noted that combining the `walkdir` crate with Rayon *"massively speeds up"* traversal of large trees [21]. Rayon ensures that you won't oversubscribe the CPU with too many threads; if you spawn 1000 tasks for 1000 dirs, they execute on, say, 8 threads, reusing them efficiently [22]. This model also simplifies memory safety: you can share an `Arc<AtomicUsize>` or similar counter for aggregated statistics without heavy locking, or use thread-local accumulators merged at the end.

Producer-Consumer with Channels: Another pattern is to decouple directory reading from processing using channels. For example, one thread (or a few) performs the directory walk, sending file metadata into a channel; a pool of worker threads receives these and performs analysis (size summation, filtering, etc.). Rust's channels (std or better, crossbeam's) provide thread-safe queues. This pipeline can smooth out irregular workloads: if reading directories is I/O-bound and spiky, the channel buffers file entries so workers can steadily digest them. A **lock-free queue** like a concurrent deque (e.g. crossbeam's SegQueue or work-stealing deque) may also be used for task distribution without the overhead of locking per task. The `ignore` crate uses a thread pool and channels to parallelize filesystem searching (for ripgrep), which serves as a practical reference for a pipeline approach.

Minimizing Lock Contention: A naive approach might use a global mutex-protected data structure (e.g. a global map of directory sizes) updated by all threads – but this would serialize much of the work. Instead, prefer **aggregation after parallel phase**. For instance, each thread can accumulate local results (say, a vector of `(path, size)` or partial tree) and only merge into the global structure at the end or at coarse intervals. If real-time progress updates are needed, use atomic counters for simple metrics (files scanned, bytes counted) – atomics are lock-free and have minimal overhead for single counters. For more complex shared state (like building a tree of directory nodes), consider a design where one thread is responsible for assembling the tree, while worker threads just send it intermediate data via a channel (thus only that one thread touches the data structure, avoiding locks altogether). Another tactic is to use *fork-join* style: parallelize the CPU-heavy parts but do combination in a single thread. For example, computing checksums on files can be done by N threads in parallel, but then collating those checksums into a report is trivial single-thread work.

CPU-bound vs I/O-bound Behavior: It's important to identify which parts of the job are CPU-intensive. Simply listing file names and sizes is usually I/O-bound (waiting on directory reads), whereas computing a SHA-256 hash of each file is CPU-bound (and also I/O-bound on file reads). Our tool likely will be scanning metadata (I/O-bound) and possibly computing fast hashes for drift detection (which can be CPU-bound if files are large). In an I/O-heavy workload, using more threads than cores can help hide latency – e.g. 16 threads on an 8-core system might achieve slightly faster deletion of tons of tiny files, because while many threads block on I/O, others proceed. In contrast, for pure CPU tasks like compressing data, you'd use at most the number of physical cores (or logical, if hyperthreading yields gains). As a guideline, for **1 GB to 1 TB projects** with millions of files, a thread pool on the order of 8–16 threads is reasonable for I/O-heavy tasks (NTFS can handle multiple operations in parallel). We might expose a tuning parameter or auto-detect the storage device type (e.g., fewer threads for HDD to avoid excessive seeking, more for SSD).

Rust Concurrency Primitives: If fine control is needed, Rust's `std::sync::mpsc` or better, crossbeam's channels, can coordinate work. One pattern: a directory walker thread pushes subdir paths onto a channel; a pool of worker threads pops from this channel and processes each directory (listing it and pushing any discovered subdirs back into the channel). This is effectively a breadth-first traversal using a thread pool. This approach must carefully handle termination (workers need to know when no more dirs remain). Using Rayon's recursive `join` or `scope` can be simpler, as it implicitly manages task completion. The Rust community has also produced crates like `rayon-core` (if custom scheduling needed) and `tokio` for async (though here, async is less useful since filesystem APIs are blocking; we'd use threads rather than async I/O).

Cancellation and Progress: Long-running operations should check for cancellation periodically. This could be as simple as an `AtomicBool` flag that workers check every few seconds or after each batch of files. If set, they can stop producing new tasks and unwind. Partial progress persistence might mean writing out the partially scanned directory list to a file or database if the operation is interrupted, so it can resume. This is complex to do exactly, but we can at least periodically save the intermediate "plan" to disk. For deletion, if it's interrupted, one should be able to resume or rollback (which might mean using the quarantine/rename strategy or recycling bin so that interruption doesn't leave half-deleted state without recovery).

Memory Safety and Footprint: Thanks to Rust's ownership model, we avoid common concurrency bugs (no data races on safe code). But we must still be mindful of memory usage: storing millions of file entries can consume a lot of RAM. A balance is needed between collecting enough information and not overloading memory or the GUI. We might stream results to the UI (e.g., show partial results incrementally). Using iterators that yield as we scan (rather than building giant vectors) is one approach, but if the UI must allow interactive sorting/filtering of all files, we may need them in memory or in an indexed database. For very large trees, an alternative is to keep only aggregate info in memory (like directory sizes) and load file lists on demand when user drills down – similar to how TreeSize can dynamically expand a folder when clicked.

In summary, for our Rust implementation we recommend: use Rayon for straightforward parallel recursion (it's simple and proven), employ channels if a producer-consumer model fits, minimize shared mutable state by partitioning work, and carefully choose thread counts. For a 1TB project with millions of small files, we might default to e.g. 8 threads for scanning and 4 threads for hashing (if hashing is enabled), adjusting if we detect an HDD (where too much concurrency could just increase seek thrash). Ensure to document these choices and perhaps let advanced users configure thread counts. With these patterns, Rust can handily exploit modern multi-core systems and high-speed NVMe storage, making even the largest cleanup tasks complete in reasonable time.

3. Directory Snapshots, Drift Detection & Incremental Verification

To ensure safe cleanup, our tool will create a "snapshot" of the directory state when generating the cleanup plan, then later verify nothing has unexpectedly changed (drifted) before executing deletions. Designing this requires balancing thoroughness with performance.

Snapshot Contents: At minimum, a snapshot entry for each file should include its path (relative to the root scanned), size in bytes, and last modification timestamp. These serve as a **fingerprint** for change detection. By default, most tools (like rsync) rely on mod-time and size as a fast-change check [23]. Rsync's *quick check* algorithm treats a file as unchanged if the timestamp and size match between source and destination [23] – this is very efficient (no reading file contents) and usually reliable. Our use case is similar: if after plan creation a file's mod-time or size differs from the recorded snapshot, we flag it as "drifted." If both are identical, we assume it's the same content. The chance of content changing but maintaining identical timestamp and size is low in typical scenarios (it would require an adversary or unusual process deliberately restoring those attributes), so this heuristic is acceptable for non-cryptographic integrity checking.

Fast vs Deep Verification: For additional safety, we can incorporate hashing. A **fast non-cryptographic hash** like xxHash or a cryptographic but speedy hash like BLAKE3 can act as a stronger fingerprint. For instance, computing a 64-bit xxHash for each file during snapshot gives a high probability of detecting any content change, with much less overhead than SHA-256. BLAKE3 is intriguing because it's designed for speed (using SIMD and multithreading internally) and can hash at hundreds of MB/s on modern CPUs. We could make hashing optional or only apply it to files above a certain size or of certain types. Many backup tools take a hybrid approach: rely on size & date for quick check, and use checksums for additional validation if needed [24]. Notably, rsync's `--checksum` option forces reading and hashing all files (MD5 by default) to decide if they changed, but the manual warns this can *"slow things down significantly"* due to all the extra I/O [25]. In our context, computing hashes on a multi-GB tree will indeed add time – possibly not worthwhile unless the user explicitly requests a thorough verification (or perhaps for very critical files). A compromise: use mod-time/size for drift detection, but if a file is flagged as changed, then (before deletion) optionally compute a hash of both the snapshot version (if still present in some form) and current version to see if content actually differs. However, since we aren't keeping file copies, that might not apply – so more likely, hashing would be done at snapshot time and used later to confirm identity.

Unique Identifiers (IDs): NTFS and many filesystems assign unique inode numbers or file IDs to files. For example, NTFS has an MFT record number which can serve as a unique ID [26]. If we store the file ID in the snapshot, we gain the ability to detect renames or moves: a file that was renamed will have the same ID but a different path. We could then treat that as "not truly deleted, just moved" and perhaps update the plan. However, using IDs gets tricky if the volume isn't NTFS or if between snapshot and execution the OS reused an ID for a new file (unlikely in short intervals unless the original file was deleted and another created). Still, it's an advanced feature. Simpler: if a file path from the plan is missing at execution time, we can scan for files with the same ID or content elsewhere – but that might be overkill. In most cases, if the path is gone or changed, we just report it as drift (the user might have renamed it manually) and skip deleting it, since our plan can't find it.

Filesystem Timestamp Granularity & Skew: Different filesystems have different timestamp precision. NTFS timestamps have 100ns resolution [27], so essentially continuous time, whereas FAT32 only records modified times to 2-second increments. On macOS, HFS+ had 1s, APFS has nanosecond. In Windows, the `FILETIME` we get is in 100ns units UTC. Normally, drift checks will occur soon after snapshot, so clock skew isn't an issue (we use the filesystem's stored time, not the current clock). However, one corner case: if files are on a network drive or if the system clock was changed, the "last modified" time of new files might appear earlier than our snapshot time. It's unlikely to cause confusion unless someone restored an older file with an old timestamp. We might consider recording the snapshot creation time as a baseline and warn if any file has a mod-time in the future relative to the snapshot (which could indicate clock issues). Generally, trusting the FS mod timestamps is fine.

Cheap vs Deep Validation: For drift detection just before deletion, the **cheap check** is to stat each file (get current size & mtime) and compare to snapshot. This is very fast – essentially one `stat` call per file. If a file is missing entirely, that's a drift (someone or something removed it after plan was made). If a new file has appeared in a directory that was to be deleted, that's also drift (we should detect new files by scanning directories again and seeing any unknown entries). Depending on policy, we might prompt the user whether to include the new files or cancel deletion of that folder. For deep validation, if we had stored hashes, we could (if paranoid) re-hash the file now and compare to the stored hash – but doing so for all files is expensive. It might be acceptable for small numbers of critical files (maybe offer an "verify integrity by hash" toggle). Tools like borg backup or restic do store hashes and validate them on restore, but those are backup scenarios where thoroughness is key. In a cleanup tool, perhaps consistency (no unexpected changes) is more important than cryptographic integrity.

Incremental Verification Strategies: One idea is **incremental hashing** – e.g., if we store a rolling hash or chunk hashes, we could detect which part of a large file changed. This is likely overkill. Instead, consider **incremental plan updates**: if minor drift is found, the tool could present a revised plan (for example, "File X was modified since planning – do you still want to delete it?"). This becomes an interactive step: the user could decide to keep that file if it changed. Our tool can facilitate this by highlighting drifted items in the UI prior to execution.

Plan File Format (Human-Editable): Using a structured text format like TOML or YAML is recommended for clarity. For example, a plan file in TOML might look like:

```toml
[[entry]]
path = "build/obj/temp.o"
size = 123456
modified = 2025-11-19T18:30:22Z
action = "delete"

[[entry]]
path = "build/cache/"
size = 0
modified = 2025-11-10T10:00:00Z
action = "purge_dir"
```

Each entry can represent either a file or directory and the intended action. Storing a **snapshot of metadata** alongside the action allows verification. The plan could also have a header with the base path and snapshot timestamp, etc. We may also include a version number for the plan format for future-proofing. Since users might edit this file, it must be readable – hence using clear keys like `path`, `size`, `modified`, rather than positional CSV columns, is preferable.

When reading the plan at execution time, the app will parse this file and for each entry confirm: does the item still exist? If yes, does its size & timestamp match? If any mismatch, mark that entry as needing user review. For efficiency, we can batch stat calls by directory (e.g., use `std::fs::read_dir` on a parent directory to get all children, because often drift checking will be done on a per-directory basis rather than calling stat for each path separately, which might be slower if done in arbitrary order). However, given modern NVMe speeds, simply iterating through the plan and `metadata()` on each path likely is fine even for hundreds of thousands of files.

Handling Renames and Moves: Without file IDs, a rename appears as "old path missing, new path unknown." The safe approach is to not delete anything we can't positively identify. If an entry is missing, we warn the user and skip it (or treat it as already gone). If a completely new path exists that wasn't in the plan, we likewise warn (e.g., "Directory X has new files Y, Z. They were not included in the original plan and will be preserved unless you choose to add them to the cleanup now.”). A sophisticated approach, if we suspect a rename, could be to try to find files with the same size & timestamp somewhere else in the tree, but this can be complex and not guaranteed. Simpler is to assume that if the user moved a file out of the tree or renamed it, it’s no longer where we intended to delete, so we do nothing to it. The presence of new files means the user (or a process) possibly created something new – it might be unsafe to blindly delete those without review.

Storing Hashes or Signatures: If we choose to store hashes in the plan, use a fast algorithm like BLAKE3 or xxHash. BLAKE3 has the advantage of being cryptographically strong (unlikely to collide) while being extremely fast in Rust (there’s a well-optimized crate). For example, storing a 256-bit BLAKE3 for each file gives strong assurance of identity. The plan file could optionally include a hash field (perhaps only for files above a certain size threshold or when the user requests an “integrity plan”). We must then confront the performance of computing those at snapshot time – it could double or triple scan time for large files. Perhaps allow the user to check a “deep snapshot” option.

Rsync and Borg Inspiration: Rsync’s approach, as mentioned, is quick check by default, full checksum on demand [25]. Borg backup, by contrast, always chunkifies and hashes data (since deduplication is a goal). Borg uses SHA-256 or BLAKE2 for content and maintains an index of those. For our needs, dedup isn’t important, and we are not transferring data, just verifying local state. Therefore, the rsync model of lightweight checks with optional heavy checks is apt. Another inspiration: Syncthing. Syncthing uses periodic full directory scans and has a database of file metadata and block hashes [28] [29]. It detects changes by comparing mod-time/size in scans, then if a file changed, it *rehashes the whole file* to know its blocks [29]. We can analogously defer hashing until needed: assume no change if times match; if a time or size changed (suggesting modification), we could in theory hash the new version to see if maybe the content is actually the same (rare but possible if a timestamp changed due to copy or touch command). However, likely not worth it – if mod-time changed, we treat it as changed.

Efficiency of Validation: The drift check just before execution should be much faster than the initial scan, because we’re typically only stat’ing files, not reading directory contents recursively (we already have the list from the plan). It’s O(N) in number of entries. Even for N = 1 million, 1e6 stat calls might take on the order of a couple of seconds on an SSD – quite acceptable. (If needed, we could parallelize the stat calls using Rayon as well, but it’s probably fine sequentially or with minor parallelism, since NTFS can handle many simultaneous reads of metadata.)

Reporting and Handling Drift: UX-wise, after verification, present the user with any discrepancies: e.g., a dialog or list: “3 files have changed since the plan was created.” For each, show the old vs new size/date, and maybe the new status (“file missing” or “new file added” or “modified”). The user can then decide: proceed anyway (delete them if they were planned for deletion despite changes?), skip those items, or re-generate the plan. The safest default is to skip any item that changed or was not in plan, and inform the user that those were skipped. For instance, if a file was modified and originally marked to delete, perhaps we should not delete it without reconfirmation, because the user might have edited it and no longer want it deleted.

Storing Plan Data Efficiently: Since the plan is human-editable, we favor clarity over minimal size. A few hundred thousand entries in TOML is large but still a few tens of MB at most, which is acceptable. YAML is another option but can be slower to parse. TOML tends to be simple and readable. Alternatively, we could use JSON, but it’s less friendly for manual editing. An **append-only log** format could be used (to avoid rewriting huge files) but likely unnecessary; writing the plan once is fine. After execution, we might archive the plan or produce a summary of what was done (for record-keeping).

In summary, our snapshot and drift detection will rely primarily on **metadata fingerprints (mtime + size)** for speed [23], with optional hashing for increased certainty in critical scenarios. We will leverage filesystem IDs where available to catch moved files. The verification step will be efficient (just stat calls) and will err on the side of caution – any mismatch or new file will pause the process for user input. By taking this approach, we ensure that the filesystem state at execution still matches the user-approved plan, preventing accidents like deleting newly created files or the wrong versions of files.

4. User Experience Patterns for Storage & Cleanup Tools

A storage analyzer/cleanup tool must present complex filesystem data in a way that's **intuitive, informative, and minimizes risk**. We draw lessons from popular tools like DaisyDisk, TreeSize, WinDirStat, and others to craft a modern Tauri-based GUI that feels as polished as a classic Apple application (in the spirit of Steve Jobs/Jony Ive design ideals). Key UX patterns include visual hierarchy of storage usage, interactive selection of items to remove, clear indications of actions (delete, archive, keep), and confirmation/undo workflows to protect user data.

Visualizing Space Usage: Users should immediately see what's consuming space. Tools approach this with two complementary views: - A **hierarchical tree view** showing directories and their sizes, much like an Explorer or Finder tree but annotated with size and maybe percentage of parent. TreeSize, for example, presents an expandable folder tree with a column for size, allowing users to drill down into large folders quickly [30]. - A **"largest items" list** that disregards hierarchy and simply lists the biggest files/folders anywhere in the selection. This was WinDirStat's approach (the top pane lists items sorted by size) and is also present in DaisyDisk's interface (clicking a segment lists its contents by size). This helps users quickly find space hogs without manually traversing folders. - **Treemaps and other visuals:** WinDirStat popularized the treemap – a rectangular mosaic of colored blocks where each block's area corresponds to file size. WizTree and Disk Inventory X have similar visuals. DaisyDisk uses a **sunburst (ring) chart** – an interactive, colorful wheel representing folder sizes [31]. These visuals make it easy to spot large files and also convey proportion (e.g., one huge block taking 50% of the area). We should consider including a treemap or sunburst view in our Tauri app. Such a view should be *interactive*: e.g., clicking a segment zooms into that folder, or hovering shows the file name. These visual metaphors turn a raw list of numbers into an immediate understanding of "what's taking up my disk." They also add to the **delight** factor (DaisyDisk is often praised for its beautiful interface [31]).

Our UI can combine these: perhaps a split view with a directory tree or list on one side and a graphical representation on the other. For example, **WizTree's UI** shows a tree on the left and a treemap on the right [32] (see image below). The two are linked – selecting a folder in the tree highlights the corresponding region in the treemap.

Example of WizTree's interface combining a hierarchical tree (left) with a treemap visualization (right), allowing intuitive exploration of disk usage.

Intuitive Hierarchy Navigation: The user should be able to navigate down into subfolders fluidly. This means supporting double-click or expand/collapse in the tree, breadcrumb navigation (a horizontal path that they can click to go up levels), or even a search function to locate folders by name. TreeSize and

DaisyDisk both allow expanding subfolders in place. DaisyDisk specifically lets you click a segment of the ring to drill down, with a smooth animation and an updating breadcrumb. A **breadcrumb trail** at the top of the window helps users orient themselves after drilling deep.

Largest Files and Filters: Provide a "Top 50 largest files" list or similar. Many users hunting for space will appreciate a quick way to see the single biggest files. This can be a separate tab or an integrated view. We might also allow filtering by file type (show me all ISOs, or all *.log files over 1GB, etc.), though that's an advanced feature. Some tools also highlight potentially deletable items like caches – for instance, showing a category "Temporary files" or "Cache" if patterns match (we could detect `node_modules`, `target` (Rust), `*.tmp` files, etc., and tag them).

Action Tags and Bulk Edits: Each file or folder in the plan will have an associated action: Delete, Archive, or Keep (or perhaps "Ignore" if we leave it alone). In the UI, this could be represented by an icon or tag next to the item. For example, a trash can icon for delete, a box icon for archive, a check or lock icon for keep. Users should be able to change these easily: e.g., click the icon to cycle through actions, or right-click context menu "Mark for deletion/archive/keep". Bulk selection is important – if a user wants to mark 100 items for the same action, they should multi-select (using Shift/Ctrl click or a "Select All" in a folder) and apply an action to all. To facilitate bulk changes, we could allow selecting a directory and marking the whole directory for purge (meaning delete everything inside). In fact, one planned feature is "purging" a folder: essentially equivalent to delete contents + remove folder. The UI could offer this as a single action (maybe a special icon on folders, like a broom icon meaning "clean out"). The plan's advanced options like "archive then delete" for a folder could be presented in an **actions dropdown**.

Risk Mitigation and Confirmation: Because deletion is destructive, multiple safeguards in UX are warranted: - **Review screen:** After the user crafts the plan (or after auto-generation), they should see a summary: "You have tagged 350 files (2.1 GB) for deletion and 3 folders (5 GB) for archiving." We can highlight if any of the items seem risky (for instance, if system folders somehow got included, or if an item was modified recently). This screen would essentially be the plan view where they can still make changes, but it serves as a final review. - **Confirmation dialogs:** When the user hits "Execute cleanup," a modal should require confirmation (possibly even requiring them to type a phrase like "DELETE" if we want extra safety for large operations, though that might be overkill). At minimum: "Are you sure you want to permanently delete these files? This action cannot be undone." If archiving is involved: "Are you sure you want to archive and remove these files?" - **Undo hints:** Once deletion is done, it's usually permanent (unless we moved to Recycle Bin). To offer some form of undo, one strategy is to integrate the OS's recycle mechanism for normal deletes (so users can restore from Recycle Bin if needed). We can make this a setting: "Use Recycle Bin for deletions" (slower, but safer) vs "Permanent delete". If using the Recycle Bin, then "undo" is simply restoring from it. If permanent, we could make our own lightweight "backup": for instance, moving files to a special folder instead of outright deletion, and telling the user "Files have been moved to X, and will be deleted after 7 days" – giving an opportunity to recover. However, this doubles space usage temporarily, which might defeat the purpose. Alternatively, if the user enabled archiving, they have the archive file as a backup of what was deleted. We should definitely encourage using the archive option for any user who is nervous about losing data (e.g., "Archive and then delete" as a recommended action for large folders, so they have a fallback).

Another subtle safety feature is **preventing accidental selection** of critical items. Our tool should probably *by default exclude system directories* like `C:\Windows` or `C:\Program Files` unless the user explicitly chooses them. And if they do, we could warn "You are scanning a system directory. Deleting files here can damage your OS.” Similarly for home directory, we should warn if they try to delete, say, all of `Documents`. Professional users might still do it, but a confirmation or prominent warning label on such items is good. Some cleanup tools detect system files and either hide them or mark them in red to discourage deletion. WinDirStat, for instance, doesn’t hide anything but relies on user caution. Given our target is developers/ power users, we assume some knowledge, but extra warnings won’t hurt for obviously dangerous actions.

DaisyDisk and Apple Design Influence: DaisyDisk is often cited as a prime example of **clean, beautiful UX** in this space. It emphasizes *minimalism, clarity, and interactive feedback*. Some principles we can adopt: - **Clean layout:** Avoid cluttering the interface with too many buttons or information at once. Show what’s needed contextually. For example, DaisyDisk’s main view is just the disk visual and a list; advanced options are tucked away. - **Consistency and aesthetics:** Use a modern, flat design with pleasing color schemes. Jony Ive-era design focuses on *content over chrome* – our app chrome (window borders, panels) should be subtle, letting the data visualization and file names stand out. Employ consistent icons that users can intuit (a trash can for delete, a box for archive, etc.). Maintain spacing and alignment so nothing feels cramped or haphazard. - **Smooth animations:** If possible in Tauri (via the web frontend), add transitions when navigating (e.g., expanding a folder might animate or the treemap might smoothly zoom in). This both looks polished and helps the user cognitively follow the change of context. - **Responsive feedback:** Whenever the user triggers an action, give immediate visual feedback. For instance, if they mark a file for deletion, maybe that row turns a shade of red or the icon appears filled. If they check a box to select multiple files, update the status bar to say “X items selected (Y GB)”. Progress bars during scanning and deletion should be present but not intrusive – perhaps a small progress indicator in the corner or integrated into the UI elements being populated. Many tools show a scanning progress (TreeSize shows “Scanning… 60%” in status bar). Our tool, being fast, might complete scan quickly, but for very large drives we’ll show progress (and not block the UI while scanning).

Virtualized Lists: Our interface may need to display potentially tens of thousands of files if a user expands a large folder. Rendering that many DOM elements in a webview can be very slow. Thus, **virtualization** is crucial – only render the list items visible in the scroll viewport, reusing elements as the user scrolls. Libraries like react-window or similar can handle this if our front-end is React/Vue, etc. This ensures the UI stays snappy even with huge data sets. Similarly, the tree view should virtualize nodes (rendering a giant tree is otherwise expensive). We might implement a custom virtualized tree component or leverage an existing one in our chosen web stack.

Smooth Progress Updates from Rust: Tauri allows sending events from Rust to the webview. We should stream progress updates (like “scanned X files, Y GB so far”) at a reasonable rate (perhaps a few times per second) to show in a progress bar. This must be done carefully to avoid flooding the message loop (sending thousands of events per second would overwhelm the UI). Throttling updates to, say, 10 per second is plenty. The progress UI could be a simple determinate bar if we know the total ahead (we might not know total files until we scan them, though we could estimate based on initial directory count). Alternatively, a spinner plus a count of files found so far can reassure the user that scanning is active. Non-blocking UX means the user can start exploring partial results while scanning continues (like TreeSize shows partial results progressively). Implementing that requires threads: our Rust backend can scan and periodically send partial data (e.g., “folder X now has size Y”) so the UI can update nodes incrementally. This is complex but improves the experience for very large scans (the user can start marking obvious large files for deletion even before the entire scan finishes).

Editing the Plan Interactively: Users should be able to modify the proposed actions easily. If the app auto-marks certain items (like build caches) for deletion, each such item should have a clear marker (maybe a checkbox ticked for "delete"). The user can uncheck it to keep it. Conversely, items not marked can be checked. This parallels the design of some installers (where you check the components to install). A side panel could list "Planned actions" – e.g., "Delete (5 items, 2.5 GB), Archive (1 folder, 1 GB)". The user could click those categories to review exactly which items are in them, providing a form of grouping.

Previews and Details: For risky deletions, the user might want to peek at the file to ensure it's the right one. A convenient feature is a quick preview: maybe pressing Space or a "Preview" button to open the file (or show a text excerpt if it's a text file, image thumbnail if image, etc.). This can be implemented by invoking the system's default opener (or a custom preview for common types). At minimum, a right-click "Open in Explorer" or "Reveal in Finder" is valuable – WizTree and DaisyDisk offer that [33]. This allows the user to manually inspect a file before deleting.

For archives, if we plan to create an archive, a preview might list what will be archived. If we integrate an archive library in the app, we could allow browsing the archive file before deletion (though at that point it's equivalent to just looking at the original structure). Another idea: after archiving, allow the user to easily open the archive (if it's zip, just double-click to open with system archive viewer).

Safe Defaults and Undo: To align with "safe cleanup," default to non-permanent deletion. Perhaps the first time a user runs it, default all deletions to the Recycle Bin (on Windows) or Trash (on Mac). Provide a toggle "Permanent delete" for those who explicitly want speed and know what they are doing. This way, if a user accidentally deletes something they needed, they can recover it easily. If using Recycle Bin, one challenge is performance (moving 100k files to Recycle Bin can be slow due to additional overhead). We might then offer two execute modes: "Safe delete (slower, recoverable)" and "Fast delete (permanent)".

Designing for Apple-Like Polish: Emulating a "Jony Ive Apple design" means focusing on *simplicity and elegance*. Some concrete tips: - Use plenty of whitespace and gentle fonts. Don't overload with dense text or technical jargon. For example, show "2.5 GB" instead of "2684354560 bytes" (but allow detailed info in a tooltip if needed). - Use color sparingly and meaningfully: DaisyDisk uses bright colors in the chart but overall the window isn't full of gaudy buttons. We might use a neutral or dark background for the treemap to let colored blocks stand out, or a light theme with pastel colors. - Icons should be **flat, modern glyphs** (perhaps using an icon set or Apple SF Symbols style icons if possible). Avoid overly skeuomorphic or outdated icons. - Consistency: All action buttons (like "Delete" or "Archive") should be placed predictably (e.g., always at bottom-right for primary action). Follow platform conventions – on Windows, typically the action buttons in dialogs are on the right; on Mac, they're on the right as well (with Cancel to the left of OK). - Perhaps include a **preview gauge** in the main view (like DaisyDisk's initial scan screen shows each drive with a gauge). If our tool can manage multiple root folders or drives, showing an overview of each could be nice (though initially it might be one folder at a time). - Provide **helpful messaging**: e.g., if nothing substantial is found to delete, say so: "No large files found in this directory that are obvious candidates for cleanup." Many Apple interfaces include friendly, guiding text for empty states.

UX for Advanced Actions: Features like "archive then delete" should be presented clearly. If a user selects a folder and chooses "Archive", the UI might prompt for archive format or location. A simple default could be archiving in place (creating a .zip in the same folder or a centralized "Archives" folder). DaisyDisk doesn't have archiving, but some tools like NCleaner or others might. We should ensure that after archiving, the original files aren't deleted until the archive is confirmed successful – in UX terms, perhaps show an intermediate status “archiving…” and then once done, show “archived to X.zip, ready to delete originals” and then proceed. If an archive fails, present an error and do not delete.

Examples of Safe and Intuitive Flows: - *WinDirStat style:* WinDirStat, though dated, is intuitive in that you click on a big colored block and the file is highlighted in the list – this link between visual and list is key. We should ensure our UI has similar linking (clicking a visual element selects the file/folder in text list or vice versa) to cater to both visually-driven and text-driven users. - *TreeSize style:* TreeSize (especially the free version) is basically a tree with sorted sizes. It’s straightforward and safe (doesn’t automatically delete anything, user must right-click and delete). We can incorporate a similar approach but with integrated action selection. Perhaps an **inline control** in each row for selecting action (like a checkbox or dropdown). However, that could clutter the UI; an alternative is selecting items and then choosing an action from a toolbar or context menu. - *DaisyDisk style:* DaisyDisk has a distinctive interactive chart; its deletion workflow is also interesting: you drag files into a “collector” (a pane that lists items to delete), and then when ready, you press “Delete” once to remove all collected items. This is very user-friendly – it feels like shopping cart: you pick what you want to delete, then checkout. We might emulate this: an “Action basket” where users add items they want to delete or archive. Then confirm applying those actions. This way, users take a deliberate step to finalize selection, reducing accidental deletions. It also gives a chance to review the list in that basket. This design fits well with a polished UX ethos.

Performance in UI: Our UI should remain responsive even with large data. Techniques: virtualize lists (as discussed), do heavy computations (like sorting a huge list) off the main thread (in web, that might mean using Web Workers or doing it incrementally). Also, avoid blocking animations or excessive re-renders. We will need to test with e.g. a million-file listing (though typically we wouldn’t show all million at once; the tree structure limits how much is visible).

Final Touches: Provide an **“Appendix” or help section** (maybe in an “About” or “Help” menu) with links to external references or guides (the user requested external examples in an appendix, likely meaning they appreciate references to existing designs or studies). We could include a “Learn more” that opens documentation or a web page on best practices for cleaning disk space (including perhaps linking to DaisyDisk or others as inspirations – in a real app maybe not, but in documentation form it’s fine to cite influences). However, within the app UI, keep it clean – maybe a simple “Help” button that explains what each action does (for example, “Archive will compress the folder into a .zip file for storage”).

By adhering to these UX patterns – **clear visualization, interactive plan editing, and protective confirmations** – we create a user experience that is powerful for advanced users yet approachable and safe for cautious users. The interface will strive to embody the Apple-like qualities of being *“beautiful and useful”* [34] . In practical terms, that means every UI element has a purpose and looks good doing it, and the overall workflow (scanning, reviewing, executing) feels smooth and confidence-inspiring. We will iterate on design with user feedback to polish it to perfection, remembering the DaisyDisk principle: *“Make great design, then implement it in code”* [35] – the user experience should lead the technology, ensuring the tool is not just effective but a joy to use.

5. Archiving Strategies for Large Directories & Mixed Content

Before deleting large project folders or build artifacts, users may want to **archive** them for long-term storage. Implementing an efficient “archive then delete” workflow in Rust requires choosing the right archive format and ensuring the process is reliable and atomic. We compare popular archive formats and outline best practices for archiving within our cleanup tool.

Format Comparison – ZIP vs TAR vs 7Z: The **ZIP format** is ubiquitous and supported out-of-the-box on Windows (File Explorer can open .zip) and other OSes. It's a good default for user convenience. However, standard ZIP uses the Deflate compression algorithm, which is dated – it's not very fast by modern standards and compression ratio is moderate. For very large data, ZIP files also require the ZIP64 extensions (for files >4GB or >65k files), which most modern zip libraries support but some old software might not. On the performance front, a Rust bench noted that creating a tarball compressed with modern algorithms can outperform deflate-zip both in speed and ratio [36]. **TAR** (tarball) by itself is just an archive of concatenated files with no compression – typically one would then compress the .tar with Gzip, Bzip2, or newer algorithms like Zstd. Tar has advantages: it preserves Unix permissions and metadata well, and it's **streamable** (you can write files to tar sequentially and even start compressing on the fly). It has no inherent size limits (beyond some format specifics which in POSIX tar have been extended to handle very large files). The downside is that to extract a single file from a tar.gz, the program has to potentially read through all preceding data (unless an index was built). But for our scenario (archiving a whole directory as a backup, not for frequent random access), tar is fine. On Windows, tar isn't natively openable without an external tool (though Windows 10 added tar support via command-line and Explorer can now handle .tar.gz in some versions, since they added libtar/libzip internally for WSL). Still, average users might not know what to do with a .tar.zst file.

TAR+Zstd: Zstandard is a state-of-the-art compression algorithm providing **very fast compression** and decompression with good ratios [37]. It often beats zip's deflate in both speed and compression by a large margin [38]. For example, compressing data with Zstd level 3–5 can reduce size ~40-45% and is much faster than deflate, while higher levels (7+) approach or exceed deflate's compression but still at decent speed [38]. In one experiment, Zstd at level 7 achieved 50% size reduction vs the original (similar to a max-level deflate or Brotli) but in 1.2 seconds, whereas Brotli level 9 (an extremely high compression, deflate alternative) got 54% but took 4.2 seconds [38]. This shows Zstd's sweet spot: by level 5-7 it gives big compression gains at relatively low CPU cost. Many projects (including package managers, docker, etc.) have adopted Zstd for this reason. We could implement archiving by writing a tar stream and feeding it through Zstd compression on the fly. The Rust `tar` crate combined with `zstd` crate makes this straightforward. The output would be a `.tar.zst` file. However, user-friendliness of that format is a consideration: not everyone has a .zst extractor handy. Yet, tools like 7-Zip (a common tool among developers) are adding support for Zstd-compressed archives [39]. As of mid-2025, 7-Zip does support .tar.zst (and even Zstd inside .zip as an extension). The built-in Windows Explorer does *not* support Zstd in zip yet, nor .tar.zst, though this could change in the future [39]. If our target users are power users, they likely can handle a .tar.zst (and we could always include a small note like "You can use 7-Zip or similar to open this archive").

ZIP with modern compression: Interestingly, the ZIP format is extensible: it has entries for different compression methods. Zstd compression *within* a .zip file is actually defined (method 93 in the spec) [40]. This means we could create a .zip where each file is compressed with Zstd instead of deflate. The advantage: it's still a .zip file (so users will try to open with Explorer), but the disadvantage: as of now, Windows Explorer does not support Zstd compressed entries [39]. 7-Zip support is coming, but not everyone will have that immediately [39]. So a user might double-click the archive and get an error or see an empty archive. Therefore, unless we know the user has the right tool, using non-standard compression in .zip could be confusing. We might stick to deflate in .zip for compatibility, or offer an option: "Use advanced compression (Zstd) – note: requires 3rd party tool to open".

7z format (7-Zip): The 7z format (not to be confused with 7-Zip the program) offers very high compression (LZMA/LZMA2 algorithms) but compression speed is slow and memory usage high. It shines for maximum compression on relatively small data sets (or where time isn't an issue). For our use, speed is more important (we want archiving to be part of a quick workflow). Also, integrating a 7z library in Rust is not as straightforward; there's no well-known pure Rust 7z library (most would shell out to 7z.exe or use a C library). So likely we avoid 7z format.

Recommendation: Default to creating a **ZIP archive (ZIP64 enabled) with Deflate** for broad compatibility. This will compress moderately and anyone can open it easily. But also offer an advanced setting for "Faster, high-ratio archive" which could produce a .tar.zst. Or perhaps the tool can create both: e.g., if a directory is extremely large, maybe tar+zst is better suited (because compressing 1 million files in a zip also has overhead of compressing each individually, whereas tar compresses across file boundaries somewhat, improving ratio if there are many small similar files). Tar+zst will generally produce smaller archives than zip+deflate for large sets of files, because tar allows compression to see redundancy across file boundaries. With zip, each file is deflated separately, so you miss cross-file compression opportunities [41] [42]. For example, lots of repeated patterns spread across multiple files will compress better as one stream.

Streaming & Memory: We must ensure we **stream the archive creation** (not accumulating everything in RAM). The Rust `tar::Builder` can append files from disk one by one, reading each file and writing to output (which could be a `zstd::Encoder` wrapping a `File` for the .tar.zst, or a `zip::write::ZipWriter`). This way, even a 100GB directory can be archived with low memory usage (just a buffer for each file read). We should use buffered I/O for efficiency. Multi-threading the compression is another consideration: BLAKE3 (for hashing) and Zstd (for compression) both have multithreaded modes. For example, zstd has a concept of threading by splitting input into chunks – but in a streaming context, using multiple threads is tricky because you need random access for that. Instead, since our CPU will likely not be the bottleneck if using a fast algorithm (I/O will), single-thread compression might suffice. If we were compressing to a slower medium or doing heavy compression (level 15+), then multi-thread might help. We can gauge and maybe expose "compression level" setting.

Post-Archive Verification: Ensuring the archive is correct is paramount since we plan to delete originals. A few steps can help: - After writing the archive, flush and close it properly (ensuring all headers/TOC are written). In case of ZIP, the central directory must be written at end; in tar, not much to do except writing end-of-archive markers. - We can then attempt to **re-open the archive and list its contents** to verify integrity. For a tar, we could attempt to read it back with the `tar` crate (or at least open and check that the last file entry and end-of-archive marker are present). For zip, the `zip` crate can be used to read the directory. This catches cases like a failure in finalizing the file. - Optionally, compute a **checksum of the archive file** and store it (maybe in a log or in the archive filename itself) so the user can verify it later. But that might be overkill. - Another robust method is to do a quick comparison: for each file we archived (we still have the metadata list), compare file counts and total size. If the archive's uncompressed total bytes (sum of all entries' original size) matches the sum we intended to archive, likely everything got in. The archive format often records uncompressed sizes for each entry (ZIP does; tar is literally the uncompressed data). So a mismatch would indicate something missing. - We could even sample a couple of files: extract one or two files from the archive to a temp location and binary-compare to original (but by the time we run archive, we may have already started deletion if we are doing it streaming? We should probably do *archiving completely first*, then deletion).

Atomicity and Safety: The ideal scenario is **never delete original data until we know the archive is safely written**. So the tool should: 1. Create the archive in a **temporary location** (e.g., same parent directory or a temp folder on the same drive to ensure move operations are atomic). 2. Optionally verify the archive as described. 3. Only then proceed to delete the originals. 4. If deletion is completed successfully, we might move the archive to a final intended location (or if we wrote it in place with a temporary name, rename it to the final name). 5. If anything fails in steps 1-3, abort and leave everything in place (perhaps report error, and maybe offer to clean the partial archive if it's incomplete).

By doing this, we ensure we don't end up with half-archived and deleted data. Using the same drive for temp archive is important because moving a large file across drives isn't atomic and could fail; plus it would duplicate data usage. If the user chooses to archive into the same folder as source, that could be problematic (we shouldn't include the archive itself in the archive!). So best to default the archive output either one level up or in a user-specified archive directory. For instance, if cleaning `C:\Projects\BigProj`, perhaps create `BigProj.zip` in `C:\Projects` by default.

Grouping and Patterns: Often project outputs are structured (e.g., multiple "runs" or versioned builds). The tool could assist by detecting these patterns. For example, if a project has subfolders `build_01`, `build_02`, ..., `build_10`, perhaps the user might want to archive all but the latest. We could highlight sequentially named folders and their sizes, maybe suggesting "Archive older runs". Similarly, if we find a bunch of log files like `log1.log, log2.log,... logN.log`, we might offer to archive them into one file or delete them in bulk. This is more of a nice-to-have pattern detector. Implementation might rely on regex or numeric suffix detection.

One-Click Archival UI: In the UI, archiving an item should be as simple as marking it for archive. Possibly, we allow archiving of entire directories primarily (archiving individual files is less common, user could just leave or delete individual large files). When the plan is executed, those marked for archive trigger the above process. We should show progress to the user: e.g., "Archiving folder X (1.2 GB)... [progress]", then "Archive complete. Deleting original...". If archive fails, show an error and do not delete originals, as mentioned. If multiple archives are to be made (e.g., archiving 3 separate directories), decide whether to do them sequentially or concurrently. Likely sequentially is better to avoid disk contention and huge memory usage if both try to compress at once. We can queue them: archive first folder, verify, delete originals, then move to next. That way if one fails, we can stop without affecting the others.

Archive Format Speed Considerations: If speed is paramount (maybe the user just wants to quickly compress a folder to save space and doesn't care about portability), we could even implement an "Store then compress in background" approach: e.g., move the folder to a temp location (as a way to reclaim space immediately from the working area), then compress it in background thread while user's main cleanup continues. But this is complex and perhaps unnecessary given speeds of modern CPUs.

Rust Libraries: For ZIP, `zip-rs` crate is available. It supports writing ZIP64 and compressing entries with deflate by default. It might not yet support Zstd compression (unless we use an external crate or contribute a feature). But we can always post-process a zip (not easily though). Tar and `zstd` crates will do tar.zst easily. Both approaches are pure Rust, so cross-platform and not requiring external binaries.

Integrity and Checksums: If we want an extra layer, we could generate a manifest file (like a .sha256 or .md5 list for all files) and include it in the archive. But that might be overkill since we plan to trust the archive as is. It's good to note tar has no built-in checksum for file contents (only header checksums); zip has CRC32 for each file which will detect bit-level corruption to some degree. So zip has an advantage that if you try to extract and a file's CRC doesn't match, it warns of corruption. Tar+zst, you rely on the zst frame checksum and nothing per file, but zstd frames usually do include a checksum for the compressed block (optional, but on by default for streaming API it may be on). We should ensure to enable zstd checksums. The `zstd` crate by default enables a content checksum at end of stream I believe (or we can turn it on). That way, if the .tar.zst is corrupted, the decompressor will know.

Archiving Mixed Content: Mixed content might include already-compressed files (videos, zips, etc.) that don't compress further. Deflate or Zstd will detect incompressible data and not waste much time (they'll output near same size). That's fine. If a folder has many small files, an archive is beneficial because it consolidates them (fewer files on disk, less overhead). For huge singular files, archiving might not reduce size (if e.g., it's a .ISO or .mp4). We might even detect that and suggest either skipping compression (store-only) or not archiving at all but just moving the file. But as a simple approach, we just compress anyway unless user chooses "store without compression".

After Archival – user guidance: After we archive and delete, the user now has an archive file that is maybe not small, but at least it's one file. We should maybe tell them "Archive created: X (you saved Y bytes, compression ratio Z%)." This gives a nice sense of accomplishment and also informs them if compression was effective. For example, "ProjectBackup.zip created (500 MB, was 1.2 GB, compressed to 42% of original size)." This feedback is useful and educational.

Example scenario: The user chooses to archive a folder "old_builds/". The tool creates `old_builds_2025-11-19.zip` in the parent directory (appending date for uniqueness maybe). It compresses the content, shows progress. Once done, it closes the zip, verifies the central directory, perhaps prints a success message in log. Then it recursively deletes the `old_builds/` folder. If deletion completes, we mark the whole task done. If anything goes wrong at any stage, we halt and show an error. The archive file remains (if archive step failed, perhaps delete the partial archive to avoid confusion; if archive succeeded but deletion failed on some files, we might keep both and warn user to manually check).

By following these archiving strategies, we ensure the **"purge → archive → delete"** workflow is fast, safe, and user-friendly. We leverage efficient Rust libraries and algorithms (favoring **Zstd** for performance [38] and using formats appropriately) to handle large, messy folders. Crucially, we always prioritize data integrity: no file is removed until its archived counterpart is secured. With options for both mainstream (zip) and advanced (tar+zstd) archiving, users can choose their preferred trade-off between **maximum compatibility** and **maximum efficiency**. This archival feature adds a layer of safety to cleanup operations, making the tool ideal for developers who want to clear disk space without permanently losing the ability to restore those project artifacts if needed.

References:

1. Antibody Software – *WizTree Features & FAQ* (2025) – Describes WizTree's use of NTFS Master File Table for fast scanning [1] and accurate size reporting with hardlink handling [43].
2. Antibody Software – *WizTree vs WinDirStat Speed Test* – Empirical speed comparison showing WizTree's 22×–46× faster performance on SSDs/HDDs by reading the MFT [44] [5].
3. Carlos Delgado, *Our Code World* (Oct 2025) – *Why is WizTree so fast?* Explanation of MFT scanning, avoiding Windows file API overhead [2] [45].

4. Alexander Riccio (2015) – *FindFirstFileEx vs FindNextFile performance* – In-depth analysis of Windows file enumeration: overhead of system calls per file [3] and effects of FindExInfoBasic and FIND_FIRST_EX_LARGE_FETCH [6]. Also demonstrates one core saturation and need for parallelism [8].
5. Alexander Riccio (2015) – *Parallel directory enumeration experiment* – Demonstrates using futures/async to enumerate directories in parallel, significantly reducing total time [11] [46].
6. Rust User Forum (2020) – Discussion of parallel directory walking using Rayon and the `jwalk` crate, noting 3-4× speed improvements over single-threaded and considerations on shallow vs deep directory structures [13] [14].
7. Rust User Forum (2022) – *Why is Rust traversal slower than Node* – Suggests using `walkdir` + Rayon to massively speed up directory walking in Rust [21].
8. Rsync Manual Page – Describes the default quick-check (file size and mtime) and the `--checksum` option which uses MD5 hashes at cost of heavy I/O [23] [24].
9. Syncthing Docs (2021) – *Understanding Synchronization* – Explains that Syncthing detects changes by mod-time/size and then rehashes blocks (SHA-256) when changes are detected [29], emphasizing cheap initial check followed by deep scan if needed.
10. OSXDaily (2016) – *Best Disk Space Analyzers (Mac)* – Notes DaisyDisk's beautiful, intuitive interface using an interactive colorful wheel and fast scanning [31], and highlights OmniDiskSweeper's sorted list approach [47].
11. DaisyDisk Blog (2011) – *Design principles behind DaisyDisk* – Emphasizes design-first, focusing on user experience over feature count [48] [35] and ensuring every solution is elegant and needful [34] – reflecting the Apple design ethos.
12. Martin Brinkmann, gHacks (2017) – *Deleting large folders in Windows super fast* – Recommends command-line deletion and explains Windows Explorer's slowness due to preparing file counts and UI updates [49] [50]. Provides steps for `DEL /F/Q/S` and `RMDIR /Q/S` which skip confirmation and output for speed [51] [52].
13. Microsoft Q&A (2023) – *Mass deletion performance* – Suggests that using command-line deletion (`del /f/s/q > nul`) is fastest and references a third-party "Rapid Delete Pro" tool for even faster deletions [53] [54].
14. Nick B., *Zstd in ZIP vs Brotli tarballs* (2023) – Experiment showing switching from zip(deflate) to tar+brotli then to zip+Zstd. Concludes Zstd within zip gives faster compression and equal/better ratios, with upcoming tool support [36] [39]. Provides data: Zstd level 5 ~45% size, much faster than Brotli level 9 ~54% [38].
15. Gregory Szorc (2017) – *Better Compression with Zstandard* – Notes that zip typically compresses each entry independently (can use Zstd but not widely supported yet) [40], whereas tar + external compression can exploit inter-file redundancy. Also confirms Windows default tools don't yet support Zstd in zip [39].

---

[1] [32] [43] WizTree - The Fastest Disk Space Analyzer
https://diskanalyzer.com/

[2] [33] [45] WizTree: a free Windows utility used to analyze disk space usage | Our Code World
https://ourcodeworld.com/articles/read/1685/wiztree-a-free-windows-utility-used-to-analyze-disk-space-usage

[3] [6] [7] [8] [9] [10] [11] [46] How fast are FindFirstFile/FindFirstFileEx, and CFileFind – actually? | Alexander Riccio
https://ariccio.com/2015/01/13/how-fast-are-findfirstfilefindfirstfileex-and-cfilefind-actually/

4 5 44 WizTree vs WinDirStat
https://diskanalyzer.com/wiztree-vs-windirstat

12 Byron/jwalk: Filesystem walk performed in parallel with ... - GitHub
https://github.com/Byron/jwalk

13 14 20 22 What's the fastest way to read a lot of files? - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/whats-the-fastest-way-to-read-a-lot-of-files/39743

15 16 49 50 51 52 How to delete large folders in Windows super fast - gHacks Tech News
https://www.ghacks.net/2017/07/18/how-to-delete-large-folders-in-windows-super-fast/

17 18 53 54 quickest way to perform a mass deletion of files (Windows 11 Pro 22h2) - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/4124491/quickest-way-to-perform-a-mass-deletion-of-files-(

19 Windows Defender causes 10X+ slowdown - Getting Help - restic forum
https://forum.restic.net/t/windows-defender-causes-10x-slowdown/925

21 Why is RUST traversing directories slower than node - code review - The Rust Programming Language Forum
https://users.rust-lang.org/t/why-is-rust-traversing-directories-slower-than-node/80916

23 24 25 Rsync copy only changed files; ignore file modification time stamps - Unix & Linux Stack Exchange
https://unix.stackexchange.com/questions/366926/rsync-copy-only-changed-files-ignore-file-modification-time-stamps

26 WizTree Guides
https://diskanalyzer.com/guide

27 Windows timestamp accessor library - Project Nayuki
https://www.nayuki.io/page/windows-timestamp-accessor-library

28 29 Understanding Synchronization — Syncthing documentation
https://docs.syncthing.net/users/syncing.html

30 31 47 3 of the Best Disk Space Analyzers for Mac OS X
https://osxdaily.com/2016/04/29/best-disk-storage-analyzers-mac/

34 35 48 DaisyDisk | Blog | 10 basic principles behind DaisyDisk
https://www.daisydiskapp.com/blog/2011/05/29/10-basic-principles-behind-daisydisk/

36 38 39 40 42 There and back again with zstd zips | nickb.dev
https://nickb.dev/blog/there-and-back-again-with-zstd-zips/

37 How much faster is making a tar archive without gzip? - Hacker News
https://news.ycombinator.com/item?id=33130440

41 Better Compression with Zstandard - Gregory Szorc's Digital Home
https://gregoryszorc.com/blog/2017/03/07/better-compression-with-zstandard/