shruggie-indexer — Technical Specification

- **Project:** `shruggie-indexer`
- **Repository:** shruggietech/shruggie-indexer
- **License:** Apache 2.0 (full text)
- **Version:** 0.1.0 (MVP)
- **Author:** William Thompson (ShruggieTech LLC)
- **Date:** 2026-02-24
- **Status:** AMENDED
- **Audience:** AI-first, Human-second

Table of Contents

---

1. Document Information

1.1. Purpose and Audience

This document is the authoritative technical specification for `shruggie-indexer`, a cross-platform file and directory indexing tool that produces structured JSON output with hash-based identities, filesystem metadata, EXIF data, and sidecar metadata entries. It serves as the single source of truth for the tool's behavioral contract, architecture, and release engineering — and as the primary reference for maintaining and extending the project.

The specification is written for an **AI-first, Human-second** audience. Its primary consumers are AI implementation agents operating within isolated context windows during sprint-based development. Every section is designed to provide sufficient detail for an AI agent to produce correct, complete code without requiring interactive clarification. Human developers and maintainers are the secondary audience — the document is equally valid as a traditional engineering reference, but its level of explicitness and redundancy reflects the needs of stateless AI execution contexts.

This specification describes:

- The complete behavioral contract of the tool, including all input modes, output schemas, configuration surfaces, and edge cases.
- The architectural decisions that govern the tool's design, including historical context for decisions rooted in the project's origins as a port of the PowerShell `MakeIndex` function (see §2.2).
- The repository structure, packaging, testing strategy, and platform portability requirements necessary to ship the tool as a cross-platform CLI utility, Python library, and standalone GUI application.

This specification does NOT serve as a tutorial, user guide, or API reference. Those artifacts are derived from this document but are separate deliverables.

1.2. Scope

In Scope

This specification covers the following:

- **Core indexing engine.** Filesystem traversal, file and directory identity generation (hashing), symlink detection, timestamp extraction and conversion, EXIF/embedded metadata extraction via `exiftool`, sidecar metadata file discovery and parsing, index entry construction, and JSON output serialization. The engine is a pure library with no presentation-layer dependencies — the CLI and GUI are thin frontends that delegate all indexing work to this layer.

- **Dependency architecture.** The tool relies on `exiftool` as its sole external binary dependency for embedded metadata extraction. All other data processing — hashing, timestamp derivation, JSON parsing, sidecar file handling — uses Python standard library modules or a small set of declared third-party packages. See §12 for the complete dependency inventory.

  > **Historical note:** The original PowerShell `MakeIndex` function depended on eight external pslib functions (`Base64DecodeString`, `Date2UnixTime`, `DirectoryId`, `FileId`, `MetaFileRead`, `TempOpen`, `TempClose`, `Vbs`) and two external binaries (`exiftool`, `jq`). The Python tool eliminates six of the eight function dependencies and the `jq` binary dependency through standard library equivalents and architectural improvements. See §2.6 for the full rationale.

- **Output schema (v2).** The complete JSON output schema for index entries as defined by the shruggie-indexer v2 schema. The v2 schema consolidates related fields into logical sub-objects (e.g., `TimestampPair`, `NameObject`, `HashSet`, `SizeObject`), adds explicit provenance tracking for metadata entries, includes a `schema_version` discriminator, and provides the structural foundation for MetaMergeDelete reversal operations. A v1-to-v2 migration utility for converting existing v1 index assets to the v2 format is planned but deferred to post-MVP.

  > **Historical note:** The v1 schema (`MakeIndex_OutputSchema.json`) documents the original MakeIndex output structure and remains available as a porting reference. The v2 schema is a ground-up restructuring that eliminates the v1 schema's flat field layout and structural redundancy.

- **Sidecar metadata configuration.** The sidecar metadata file discovery and classification system defines regex patterns for identifying sidecar file types (Description, DesktopIni, GenericMetadata, Hash, JsonMetadata, Link, Screenshot, Subtitles, Thumbnail, Torrent), per-type behavioral attributes (expected data formats, valid parent relationships), the exiftool file-type exclusion list, extension group classifications (Archive, Audio, Font, Image, Link, Subtitles, Video), and indexer include/exclude patterns. The regex patterns — including an extensive BCP 47 language code alternation for subtitle detection — are externalized into the typed configuration system and are user-modifiable.

  > **Historical note:** The sidecar configuration originates from the `$global:MetadataFileParser` PowerShell ordered hashtable in the original pslib library. The isolated reference script `MakeIndex(MetadataFileParser).ps1` (see §1.5) provides the complete original object definition.

- **CLI interface.** The command-line interface exposing all indexing operations — target selection, output mode control, metadata processing options, rename operations, ID type selection, and verbosity/logging configuration.

- **Python API.** The public programmatic interface for consumers who import `shruggie_indexer` as a library rather than invoking it from the command line.

- **Graphical user interface.** A standalone desktop GUI application built with CustomTkinter, providing a visual frontend to the same library code used by the CLI. The GUI is shipped as a separate release artifact alongside the CLI executable. The full GUI specification is defined in §10 of this document.

  > **Historical note:** The GUI design language (CustomTkinter, two-panel layout, dark theme, shared font stack) is modeled after the `shruggie-feedtools` GUI (see §1.5, External References).

- **Configuration system.** The externalized configuration architecture that replaces the original's hardcoded `$global:MetadataFileParser` object, including default values, file format, and override/merge behavior.

- **Logging and diagnostics.** The structured logging system that replaces the original's `Vbs` function, including logger hierarchy, log levels, session identifiers, and output destinations.

- **Packaging and distribution.** `pyproject.toml` configuration, entry points, standalone executable builds via PyInstaller, and release artifact definitions.

- **Testing strategy.** Unit tests, integration tests, output schema conformance tests, cross-platform test matrix, backward compatibility validation, and performance benchmarks.

- **Platform portability.** Cross-platform design principles and platform-specific considerations for Windows, Linux, and macOS — including filesystem behavior differences, creation time portability, and symlink/reparse point handling.

- **Security and safety.** Symlink traversal safety, path validation, temporary file handling, metadata merge-delete safeguards, and large file/deep recursion handling.

- **Performance considerations.** Multi-algorithm hashing in a single pass, chunked file reading, large directory tree handling, JSON serialization performance, and exiftool invocation strategy.

Out of Scope

This specification explicitly does NOT cover:

- **The broader pslib library.** `shruggie-indexer` originated as a port of the `MakeIndex` function from the PowerShell pslib library. Only that function and its dependency tree informed this project. The hundreds of other functions in the original library are unrelated.

- **The `Encoding` output field.** The v1 output schema included an `Encoding` key containing detailed file encoding properties derived from BOM byte inspection. This field is intentionally absent from the v2 schema. See §5.11 for the full rationale.

- **The v1-to-v2 migration utility.** A migration script for converting existing v1 index assets to the v2 schema format is a planned deliverable but is deferred to post-MVP. It is not part of the v0.1.0 release.

- **The original PowerShell source code.** The original `main.ps1` is closed-source and is not included in the `shruggie-indexer` repository. The original source code for the `MakeIndex` function — including its complete function body, parameter block, and all nested sub-functions — SHALL NOT be included in the repository in any form (neither as a reference file, nor embedded in documentation, nor as inline code comments). All behavioral knowledge required for implementation and maintenance is captured in this specification and the reference documents listed in §1.5.

1.3. Document Maintenance

This specification is maintained as a living document alongside the codebase. It is the authoritative reference for the project's intended behavior and architecture. When the specification and the implementation disagree, the specification is presumed correct unless a deliberate amendment has been made.

Updates to this document SHOULD be committed alongside the code changes they describe. Significant architectural changes — such as adding a new output field, changing the CLI interface, or altering the configuration format — MUST be reflected in this specification before or concurrent with the implementation.

The document header's **Date** field reflects the date of the most recent substantive revision. The **Status** field uses one of the following values:

Status	Meaning
`DRAFT`	The specification is under active development. Sections may be incomplete or subject to change.
`REVIEW`	The specification is believed complete and is undergoing review for correctness and consistency.
`APPROVED`	The specification has been reviewed and accepted as the implementation target.
`AMENDED`	The specification has been modified after initial approval to reflect post-release changes.

1.4. Conventions Used in This Document

Requirement Level Keywords

This specification uses the keywords defined in RFC 2119 to indicate requirement levels:

Keyword	Meaning
MUST / **MUST NOT**	An absolute requirement or prohibition. Implementations that violate a MUST or MUST NOT are non-conformant.
SHALL / **SHALL NOT**	Synonymous with MUST / MUST NOT.
SHOULD / **SHOULD NOT**	A strong recommendation. There may be valid reasons to deviate, but the implications must be understood and the deviation must be deliberate.
MAY	An item is truly optional. An implementation may include or omit the feature without affecting conformance.

These keywords are capitalized when used in their RFC 2119 sense. Lowercase usage of these words carries their ordinary English meaning.

Typographic Conventions

- `Monospace text` denotes code identifiers, file paths, CLI flags, configuration keys, and literal values.
- **Bold text** denotes emphasis or key terms being defined.
- *Italic text* denotes document titles, variable placeholders, or first use of a defined term.
- `§N.N` denotes a cross-reference to another section of this specification (e.g., §5.2 refers to section 5.2, "Top-Level IndexEntry Fields").

Terminology

Term	Definition
Original	The PowerShell implementation of `MakeIndex` and its dependency tree within the pslib library (`main.ps1`). Used in historical notes and deviation callouts when describing behavior that informed the current design.
v1 schema	The original MakeIndex output format as defined by `MakeIndex_OutputSchema.json`. Retained as a porting reference. The tool does not target v1 output.
v2 schema	The current output format defined by `shruggie-indexer-v2.schema.json`. This is the target schema for all implementation work.
Index entry	A single structured data object representing a file or directory in the v2 output schema. Defined in §5.
Sidecar file	An external metadata file that lives alongside the file it describes, identified by filename pattern matching (e.g., `photo.jpg` may have a sidecar `photo_meta2.json`).
Content hash	A cryptographic hash computed from the byte content of a file. Used for file identity. Directories do not have content hashes.
Name hash	A cryptographic hash computed from the UTF-8 encoded bytes of a file or directory name string. Used for directory identity and as a secondary identifier for files.
HashSet	A v2 schema object containing hash digests for a given input. Always includes `md5` and `sha256`; optionally includes `sha512`. Replaces the v1 schema's separate `Ids`, `NameHashes`, `ContentHashes`, `ParentIds`, and `ParentNameHashes` fields.
NameObject	A v2 schema object pairing a `text` string with its `hashes` (a HashSet). Replaces the v1 pattern of separate `Name`/`NameHashes` and `ParentName`/`ParentNameHashes` field pairs.
TimestampPair	A v2 schema object containing both an `iso` (ISO 8601 string) and `unix` (milliseconds since epoch) representation of a single timestamp. Replaces the v1 pattern of separate `TimeAccessed`/`UnixTimeAccessed` field pairs.
Identity (`id`)	The primary unique identifier assigned to an index entry, selected from one of the computed hash algorithms (MD5 or SHA256) based on the `id_algorithm` field. Prefixed with `y` for files, `x` for directories, `z` for generated metadata entries.

Term	Definition
StorageName (`storage_name`)	The deterministic filename derived from an item's `id` and extension (e.g., `yA8A8C089A6A8583B24C85F5A4A41F5AC.exe` for a file, `x3B4F479E9F880E438882FC34B67D352C` for a directory). Used when the rename operation is active.
MetaMerge	The operation of folding sidecar metadata into the parent item's `metadata` array during indexing.
MetaMergeDelete	An extension of MetaMerge that queues the original sidecar files for deletion after their content has been merged into the parent item's metadata. The v2 schema's sidecar metadata entries carry sufficient filesystem provenance (path, size, timestamps) to support reversal of this operation.
In-place write	Writing individual `_meta2.json` or `_directorymeta2.json` sidecar files alongside each processed item, as opposed to writing a single aggregate output file.
MetadataFileParser	The configuration object governing sidecar metadata file discovery and classification. Externalized into a typed configuration structure. See §7.

Code Examples

Code examples in this specification use Python syntax unless otherwise noted. Examples marked with `# Historical reference (PowerShell)` show the original implementation's approach for comparison purposes. Code examples are illustrative — they demonstrate intent and structure but are not necessarily the exact implementation. The implementation SHOULD follow the examples' intent while MAY differing in specific variable names, error handling details, or stylistic choices.

1.5. Reference Documents

The following documents inform this specification. All repository paths are relative to the repository root and resolve within the live repository.

Planning

Document	Path	Description
Implementation Plan	`.archive/shruggie-indexer-plan.md`	Sprint-based implementation plan that guided initial construction through Sprints 1.1–3.3 and the v0.1.0 release. All sprints complete. Archived — retained for historical reference only.

Output Schema

Document	Location	Description
v2 Schema (canonical)	schemas.shruggie.tech/data/shruggie-indexer-v2.schema.json	The canonical JSON Schema definition for the v2 index entry output format. This is the target schema for all implementation work. Defines all fields, types, nullability, required properties, definitions (`NameObject`, `HashSet`, `SizeObject`, `TimestampPair`, `TimestampsObject`, `ParentObject`, `MetadataEntry`), and structural constraints. §5 of this specification interprets and extends the schema with behavioral guidance but does not supersede it for field-level definitions.
v2 Schema (local copy)	`./docs/schema/shruggie-indexer-v2.schema.json`	Local copy of the canonical v2 schema, committed to the repository. Used by the documentation site (§3.7) and as a local validation reference. This file MUST be kept in sync with the canonical hosted version.
v1 Schema (porting reference)	`./docs/porting-reference/MakeIndex_OutputSchema.json`	The JSON Schema definition for the original MakeIndex v1 output format. Retained as a porting reference for understanding the original implementation's output structure and for informing the eventual v1-to-v2 migration utility. shruggie-indexer does NOT target this schema.

Operations Reference

Document	Path	Description
Operations Catalog	`./docs/porting-reference/MakeIndex_OperationsCatalog.md`	Categorized inventory of all logical operations in the original `MakeIndex` and its dependency tree, mapped to recommended Python modules with improvement notes. The primary architectural reference for the original-to-Python mapping.

Configuration Reference

Document	Path	Description

Document	Path	Description
MetadataFileParser Object	`./docs/porting-reference/MakeIndex(MetadataFileParser).ps1`	Isolated PowerShell script containing the complete `$global:MetadataFileParser` object definition. This is the source of truth for sidecar metadata file discovery and classification — including all regex identification patterns, per-type behavioral attributes, exiftool exclusion lists, extension group classifications, and indexer include/exclude patterns. The regex patterns in this object (particularly the BCP 47 language code alternation for subtitle detection) have been carefully crafted and MUST be ported to Python with deliberate attention to preserving their exact matching semantics. See §7.3 for the full porting guidance.

Dependency Catalogs

Each dependency catalog documents a single function from the original pslib library that `MakeIndex` depends on — its parameters, internal sub-functions, external calls, and behavioral contract.

Document	Path	Original Function
Base64DecodeString	`./docs/porting-reference/Base64DecodeString_DependencyCatalog.md`	Decodes Base64-encoded and URL-encoded strings. Eliminated — exiftool arguments are passed directly.
Date2UnixTime	`./docs/porting-reference/Date2UnixTime_DependencyCatalog.md`	Converts formatted date strings to Unix timestamps in milliseconds. Eliminated — timestamps are derived directly from stat results.
DirectoryId	`./docs/porting-reference/DirectoryId_DependencyCatalog.md`	Computes hash-based identity for directories using the two-layer `hash(hash(name) + hash(parentName))` scheme.
FileId	`./docs/porting-reference/FileId_DependencyCatalog.md`	Computes hash-based identity for files from content hashes (or name hashes for symlinks).
MakeIndex	`./docs/porting-reference/MakeIndex_DependencyCatalog.md`	The top-level function being ported. Orchestrates traversal, identity generation, metadata extraction, and output routing.
MetaFileRead	`./docs/porting-reference/MetaFileRead_DependencyCatalog.md`	Reads and parses sidecar metadata files with format-specific handling (JSON, text, binary, subtitles, hash files, URL/LNK shortcuts).
TempOpen	`./docs/porting-reference/TempOpen_DependencyCatalog.md`	Creates temporary files with UUID-based naming. Eliminated — replaced by `tempfile` if needed at all.
TempClose	`./docs/porting-reference/TempClose_DependencyCatalog.md`	Deletes temporary files by path. Eliminated — replaced by context manager cleanup.
Vbs	`./docs/porting-reference/Vbs_DependencyCatalog.md`	Structured logging with severity levels, caller identification, session IDs, and colorized console output. Replaced by Python's `logging` framework.

External References

Document	URL	Description
RFC 2119	https://www.rfc-editor.org/rfc/rfc2119	Defines the requirement level keywords (MUST, SHOULD, MAY, etc.) used throughout this specification.
JSON Schema Draft-07	https://json-schema.org/draft-07/schema#	The JSON Schema dialect used by both the v1 and v2 output schemas.
ExifTool	https://exiftool.org/	The external binary dependency for embedded metadata extraction.
shruggie-feedtools	https://github.com/shruggietech/shruggie-feedtools	A sibling ShruggieTech project whose specification, repository layout, and GUI design serve as the primary architectural and visual reference for `shruggie-indexer`. The `shruggie-indexer` GUI is modeled directly after the `shruggie-feedtools` GUI (CustomTkinter, two-panel layout, dark theme, shared font stack and appearance conventions).

2. Project Overview

2.1. Project Identity

Field	Value
Project name	`shruggie-indexer`
Package name (PyPI)	`shruggie-indexer`
Import name (Python)	`shruggie_indexer`
Repository	shruggietech/shruggie-indexer
Organization	ShruggieTech LLC
License	Apache 2.0 (full text)
License file	`LICENSE` (full Apache 2.0 text, obtained from https://www.apache.org/licenses/LICENSE-2.0.txt)
Build system	`hatchling`
Current version	0.1.0 (MVP)
CLI entry point	`shruggie-indexer = "shruggie_indexer.cli.main:main"`
Module entry point	`python -m shruggie_indexer` (via `__main__.py`)

`shruggie-indexer` is a standalone project within the ShruggieTech tool family. It shares no code with `shruggie-feedtools` or any other ShruggieTech project at runtime, but it follows the same repository conventions, packaging patterns, build tooling, and GUI design language established by `shruggie-feedtools` (see §1.5, External References). Where this specification does not explicitly define a convention — such as `pyproject.toml` field ordering, ruff configuration, or venv setup scripting — the `shruggie-feedtools` repository serves as the normative reference for project scaffolding.

This project is not published to PyPI. End users download pre-built executables from GitHub Releases. The `pip install -e` workflow is for contributors setting up a local development environment only.

2.2. Relationship to the Original Implementation

`shruggie-indexer` originated as a ground-up Python reimplementation of the `MakeIndex` function from the PowerShell-based pslib library (`main.ps1`). The relationship is behavioral, not structural — the tool targets the same logical outcomes (filesystem indexing with hash-based identity, metadata extraction, and structured JSON output) but uses its own code organization, naming patterns, and internal architecture.

The tool carries forward the original's core behavioral contract: given a target path (file or directory), produce a JSON index entry containing hash-based identities, filesystem metadata, timestamps, embedded EXIF data, and sidecar metadata — all structured according to a defined output schema. Everything else — the language, the architecture, the output schema version, the dependency set, the configuration model, and the platform scope — is designed for the Python ecosystem and cross-platform execution.

The original PowerShell source code is closed-source and is not included in the repository. All behavioral knowledge required for implementation and maintenance is captured in this specification and the reference documents listed in §1.5. See §1.2 (Out of Scope) for the full policy on original source handling.

> **Historical note:** The original `MakeIndex` is one function among hundreds in `main.ps1`, a monolithic 17,000+ line PowerShell script that serves as a general-purpose utility library. `MakeIndex` itself spans approximately 1,500 lines with 20+ nested sub-functions. It depends on eight additional top-level pslib functions (`Base64DecodeString`, `Date2UnixTime`, `DirectoryId`, `FileId`, `MetaFileRead`, `TempOpen`, `TempClose`, `Vbs`) and two external binaries (`exiftool`, `jq`). The combined dependency tree encompasses roughly 60 discrete code units.

2.3. Design Goals and Non-Goals

Design Goals

G1 — Behavioral fidelity. For any input path, the tool MUST produce a v2 index entry whose semantic content is correct and complete — the same file always produces the same hash-based identity, the same timestamp values (within platform precision limits), the same embedded metadata extraction results, and the same sidecar metadata discovery and parsing outcomes. "Semantic equivalence" means deterministic, reproducible output. It does NOT mean byte-identical JSON across runs; timestamp access times and output formatting may vary.

> **Historical note (G1):** Behavioral fidelity was originally defined relative to the PowerShell `MakeIndex` function's v1 output, accounting for the documented schema restructuring (§5) and intentional deviations (§2.6).

G2 — Cross-platform portability. The tool MUST run on Windows, Linux, and macOS without platform-specific code branches in the core indexing engine. Platform-specific behavior (such as creation time availability or symlink semantics) is handled through documented abstractions with explicit fallback strategies, not through conditional imports or OS-detection switches in the hot path. The only external binary dependency — `exiftool` — is itself cross-platform.

> **Historical note (G2):** The original ran only on Windows due to reliance on PowerShell, .NET Framework types, `certutil`, Windows-specific path handling, and hardcoded Windows filesystem assumptions. Cross-platform portability was a primary design goal for the Python rewrite.

G3 — Three delivery surfaces from a single codebase. The tool ships as a CLI utility, a Python library, and a standalone GUI application. All three surfaces consume the same core indexing engine. The CLI and GUI are thin presentation layers over the library API. No indexing logic lives in the CLI argument parser

or the GUI event handlers.

> **Historical note (G3):** This is the same architecture used by `shruggie-feedtools`.

G4 — Externalized, user-modifiable configuration. All behavioral parameters — sidecar metadata patterns, exiftool exclusion lists, filesystem exclusion filters, extension validation rules — are externalized into a typed configuration system with sensible defaults, a documented TOML configuration file format, and a merge/override mechanism for user customization. A user SHOULD be able to add a new sidecar metadata pattern, extend the exiftool exclusion list, or modify the filesystem exclusion filters without editing source code.

> **Historical note (G4):** The original hardcoded all configuration in the `$global:MetadataFileParser` PowerShell object and in literal values scattered across the function body. The externalized configuration system is a deliberate architectural improvement.

G5 — Dependency minimization. The tool declares a small, deliberately chosen set of required runtime dependencies — `click` (CLI parsing), `orjson` (JSON serialization performance), `pyexiftool` (batch exiftool invocation with safe Unicode argument passing), and `tqdm` (progress reporting) — alongside `exiftool` as the sole required external binary. These four packages are listed in `[project.dependencies]` in `pyproject.toml` and are installed by a bare `pip install shruggie-indexer`. Each was promoted from optional to required because it replaces functionality that would otherwise need to be built from scratch or re-implemented with inferior characteristics (see DEV-15, DEV-16, and the per-package rationale in §12.3). The GUI package (`customtkinter`) remains optional and is declared as an extra. Development and testing tools (`rich`, `pytest`, `ruff`, etc.) are also declared as extras. The core indexing engine has no other third-party runtime dependencies beyond the four listed above.

G6 — AI-agent implementability. Every section of this specification provides sufficient detail for an AI implementation agent to produce correct, complete code for the described component within a single context window, without interactive clarification. Cross-references between sections are explicit. Ambiguous behavioral questions are resolved in the specification text rather than left to implementer judgment. This is the document's primary design constraint and the reason for its level of explicitness.

G7 — Structured, v2-native output. The tool targets the v2 output schema exclusively. The v2 schema consolidates related fields into logical sub-objects (`NameObject`, `HashSet`, `SizeObject`, `TimestampPair`, `TimestampsObject`, `ParentObject`, `MetadataEntry`), adds a `schema_version` discriminator, and provides the structural foundation for MetaMergeDelete reversal. A v1-to-v2 migration utility is a planned post-MVP deliverable (see §1.2, Out of Scope).

> **Historical note (G7):** The v2 schema is a ground-up restructuring of the original v1 schema, which used a flat field layout with separate top-level keys for each hash variant, timestamp format, and parent attribute. The tool does not produce v1 output.

Non-Goals

NG1 — Full pslib port. Only the `MakeIndex` function and its dependency tree informed this project. The hundreds of other functions in the original pslib library are unrelated.

NG2 — Backward-compatible v1 JSON output. The tool does not produce output conforming to the v1 schema (`MakeIndex_OutputSchema.json`). Consumers of existing v1 index assets will need the planned v1-to-v2 migration utility (post-MVP) or must adapt their parsers to the v2 schema.

NG3 — Drop-in PowerShell replacement. The tool has its own CLI, API, and configuration system designed for the Python ecosystem. It does not replicate the original PowerShell parameter interface or integration with pslib global state.

NG4 — Real-time or watch-mode indexing. The tool processes a target path and produces output. It does not monitor the filesystem for changes or re-index automatically. File-watching functionality is a potential future enhancement (see §18) but is not part of the MVP.

NG5 — Database or server backend. Index output is written to JSON files or stdout. The tool does not write to databases, expose an HTTP API, or provide query functionality over indexed data. Downstream consumers ingest the JSON output using their own storage and query infrastructure.

NG6 — Metadata editing or file transformation. The tool reads and indexes filesystem content. It does not modify file contents, edit EXIF tags, transcode media, or perform any write operation on the indexed files themselves. The rename operation (§6.10) renames files to their `storage_name` but does not alter their content. The MetaMergeDelete operation deletes sidecar files after merging but does not modify the parent file.

2.4. Platform and Runtime Requirements

Target Platforms

Platform	Status	Notes
Windows 10/11 x64	Primary	Primary target for standalone `.exe` builds via PyInstaller.
Linux x64 (Ubuntu 22.04+, Fedora 38+)	Supported	Full feature parity. CI test matrix includes Ubuntu.
macOS x64 / ARM64 (13 Ventura+)	Supported	Full feature parity. `st_birthtime` available for true creation time. CI test matrix includes macOS.

"Primary" means this platform receives standalone executable artifacts in every release and is the first target for manual testing and user-facing documentation. "Supported" means full feature parity, inclusion in the CI test matrix, and bug fixes for platform-specific issues, but standalone executables MAY lag behind the primary platform in release cadence.

Required External Binary

Binary	Version	Purpose	Installation
`exiftool`	≥ 12.0	Embedded EXIF/XMP/IPTC metadata extraction	Must be installed separately and available on the system `PATH`. See https://exiftool.org/ for platform-specific installation instructions.

`exiftool` is the only external binary dependency. The tool MUST verify `exiftool` availability at startup and produce a clear, actionable error message if it is not found. If `exiftool` is missing, operations that require it (embedded metadata extraction) MUST fail gracefully with a warning, while operations that do not require it (hashing, timestamp extraction, sidecar metadata reading) MUST continue to function normally.

> **Historical note:** The original also required `jq` for JSON processing and `certutil` for Base64 encoding; both are eliminated in the Python tool (see §2.6).

Runtime Environment

The tool does not require administrator/root privileges for any operation. It operates entirely within the permissions of the invoking user. Filesystem paths that the user cannot read produce per-item warnings and are skipped rather than causing the entire indexing operation to fail.

2.5. Python Version Requirements

Requirement	Value	Rationale
Minimum Python version	`>=3.12`	Matches the `shruggie-feedtools` baseline. Provides `tomllib` in the standard library (3.11+), improved `pathlib` semantics, enhanced `dataclasses` features, and current `typing` module capabilities.
Target Python version for development	3.12	The `ruff` target version, CI primary version, and PyInstaller build version.
Maximum Python version	No upper bound	The tool SHOULD work on 3.13+ without modification. The `pyproject.toml` specifies `requires-python = ">=3.12"` with no upper constraint.

The `>=3.12` floor is a deliberate alignment with the ShruggieTech project family. Key standard library features that justify this floor include `tomllib` for TOML configuration parsing without a third-party dependency, `pathlib.Path.walk()` (3.12+) as a modern alternative to `os.walk()`, improved error messages in `argparse` and `dataclasses`, and PEP 695 type parameter syntax support.

The implementation MUST NOT use features introduced after Python 3.12 without a documented compatibility fallback, to ensure the tool works on the minimum supported version.

2.6. Intentional Deviations from the Original

This section catalogs the architectural and behavioral decisions where the tool deliberately diverges from the original PowerShell implementation. Each deviation is identified by a short code (e.g., `DEV-01`) for cross-referencing from other sections of this specification. The full technical details of each deviation are developed in the referenced sections; this subsection provides the summary rationale and serves as a navigational index.

DEV-01 — Unified hashing module

A single hashing utility module provides `hash_file()` and `hash_string()` functions consumed by all callers, eliminating code duplication. See §6.3.

> **Historical note (DEV-01):** The original implemented hashing logic independently in four separate locations — `FileId` (8 sub-functions), `DirectoryId` (5 sub-functions), `ReadMetaFile` (2 sub-functions), and `MetaFileRead` (2+ sub-functions) — each repeating the same `Create() → ComputeHash() → ToString() → replace('-','')` pattern with no shared code. The unified module eliminates approximately 17 redundant sub-functions.

DEV-02 — Multi-algorithm single-pass hashing

MD5 and SHA256 are always computed. SHA512 is computed when explicitly enabled in the configuration (`config.compute_sha512 = True`). SHA1 is not computed — it serves no unique purpose in the identity system and adds overhead without benefit (see §5.2.1 for the rationale). All active algorithms are computed in a single file read using `hashlib`'s ability to feed the same byte chunks to multiple hash objects simultaneously, halving the I/O for the default two-algorithm case. See §6.3.

> **Historical note (DEV-02):** The original computed only MD5 and SHA256 at runtime despite the output schema defining fields for SHA1 and SHA512 (left `$null` in practice). Each algorithm was computed in a separate file-read pass.

DEV-03 — Unified filesystem traversal

A single traversal function parameterized by a `recursive: bool` flag handles both modes. `Path.rglob('*')` or `os.walk()` handles the recursive case; `Path.iterdir()` handles the non-recursive case. Both paths feed into the same object-construction pipeline. See §6.1.

> **Historical note (DEV-03):** The original implemented recursive and non-recursive directory traversal as two entirely separate code paths (`MakeDirectoryIndexRecursiveLogic` and `MakeDirectoryIndexLogic`) with near-complete code duplication. Both paths manually assembled an `ArrayList` from separate `Get-ChildItem` calls for files and directories.

DEV-04 — Unified path resolution

A single `resolve_path()` utility function handles all path resolution needs. See §6.2.

> **Historical note (DEV-04):** The original independently implemented the "resolve path" operation in three locations: `ResolvePath` in `MakeIndex`, `FileId-ResolvePath` in `FileId`, and `DirectoryId-ResolvePath` in `DirectoryId`. All three performed the same `Resolve-Path` → `GetFullPath()` fallback logic.

DEV-05 — Elimination of the Base64 argument encoding pipeline

Exiftool arguments are defined as plain Python string lists and communicated to exiftool directly. The primary interface is `pyexiftool` (DEV-16), which uses exiftool's `-stay_open` mode via stdin/stdout pipes — arguments are never written to disk. The fallback interface passes arguments via a `subprocess.run()` call with a `-@` argfile written through Python's `tempfile` module, using `-charset filename=utf8` for Unicode safety. No Base64 encoding is involved. See §6.6.

> **Historical note (DEV-05):** The original stored exiftool arguments as Base64-encoded strings, decoded them at runtime via `Base64DecodeString` (which called `certutil` on Windows and handled URL-encoding as a separate opcode), wrote them to a temporary file via `TempOpen`, passed them to exiftool via its `-@` argfile switch, and cleaned up via `TempClose`. The entire Base64 pipeline and its four dependencies (`Base64DecodeString`, `certutil`, `TempOpen`, `TempClose`) are eliminated.

DEV-06 — Elimination of jq

Exiftool JSON output is parsed directly with `json.loads()`. Unwanted keys are removed with a dict comprehension against the configured exclusion set. See §6.6.

> **Historical note (DEV-06):** The original piped exiftool's JSON output through `jq` for two purposes: compacting the JSON and deleting unwanted keys. The `jq` binary dependency is eliminated entirely.

DEV-07 — Direct timestamp derivation (Date2UnixTime elimination)

Unix timestamps are derived directly from `os.stat()` float values: `int(stat_result.st_mtime * 1000)`. ISO 8601 strings are produced separately from the same `datetime` object. No round-trip parsing. See §6.5.

> **Historical note (DEV-07):** The original read timestamps from `Get-Item` as .NET `DateTime` objects, formatted them to strings via `.ToString($DateFormat)`, then passed those strings to the external `Date2UnixTime` function which parsed them back into `DateTimeOffset` objects to call `.ToUnixTimeMilliseconds()` — a needless format-parse-format round-trip. `Date2UnixTime` and its own internal dependency chain (`Date2FormatCode`, `Date2UnixTimeSquash`, `Date2UnixTimeCountDigits`, `Date2UnixTimeFormatCode`) are all eliminated.

DEV-08 — Python logging replaces Vbs

Python's `logging` standard library module provides structured logging with severity levels, formatters for console and file output, rotation handlers, and built-in caller information. The tool uses a logger hierarchy rooted at `shruggie_indexer` with per-module child loggers. See §11.

> **Historical note (DEV-08):** The original used `Vbs`, a custom structured logging function and the most widely-called function in the pslib library. `Vbs` implemented its own severity normalization, colorized console output via `Write-Host`, call-stack compression (`A:A:A` → `A(3)`), session ID embedding, monthly log file rotation, and log directory bootstrapping — all manually. Python's `logging` module provides all of these capabilities natively or through standard handlers.

DEV-09 — Computed null-hash constants

Null-hash constants (the hash of an empty string) are computed once at module load time: `hashlib.md5(b'').hexdigest().upper()`, etc. This is self-documenting, eliminates the risk of copy-paste errors in long hex strings, and automatically produces correct values if additional algorithms are added in the future. See §6.3.

> **Historical note (DEV-09):** The original hardcoded null-hash constants as literal hex strings in multiple locations across `DirectoryId` and `FileId` sub-functions (e.g., `D41D8CD98F00B204E9800998ECF8427E` for MD5).

DEV-10 — Externalized filesystem exclusion filters

The exclusion list is externalized into the configuration system with a cross-platform default set that covers `$RECYCLE.BIN`, `System Volume Information`, `.DS_Store`, `.Spotlight-V100`, `.Trashes`, `.fseventsd`, and similar platform artifacts. Users can extend or override the list via configuration. See §6.1 and §7.

> **Historical note (DEV-10):** The original hardcoded the exclusion of `$RECYCLE.BIN` and `System Volume Information` as inline `Where-Object` filters in the traversal logic. These are Windows-specific system directories.

DEV-11 — v2 output schema

Output conforms to the v2 schema, which consolidates related fields into typed sub-objects (`NameObject`, `HashSet`, `SizeObject`, `TimestampPair`, `TimestampsObject`, `ParentObject`), adds a `schema_version` discriminator, adds filesystem provenance fields to `MetadataEntry` for MetaMergeDelete

reversal, and eliminates the `Encoding` field (see DEV-12). The v2 schema is defined at schemas.shruggie.tech/data/shruggie-indexer-v2.schema.json. See §5.

> **Historical note (DEV-11):** The original's output conformed to the v1 schema (`MakeIndex_OutputSchema.json`), which used a flat field layout with separate top-level keys for each hash variant, timestamp format, and parent attribute — resulting in significant structural redundancy.

DEV-12 — Encoding field dropped

The `Encoding` field is intentionally absent from the v2 schema. The encoding information it provided (code page identifiers, `IsBrowserDisplay`, `IsMailNewsSave`, etc.) is deeply coupled to the .NET `System.Text.Encoding` type hierarchy and has limited utility outside of .NET consumers. See §5.11.

> **Historical note (DEV-12):** The v1 output included an `Encoding` key containing detailed file encoding properties (BOM detection, code page, encoder/decoder fallback objects) derived from the `GetFileEncoding` sub-function, which invoked a custom C# class (`EncodingDetector.cs`) loaded at pslib initialization. The `GetFileEncoding` sub-function and all related logic (`EncodingDetector.cs`, `GetFileEncoding-Squash`) are not carried forward.

DEV-13 — Dead code removal

The following original functions are not carried forward: `ValidateIsLink` (listed as a dependency but never actually called — `FileId` and `DirectoryId` perform symlink detection inline), `UpdateFunctionStack`, and `VariableStringify` (internal utilities whose purposes are absorbed by Python built-ins). See §12.4.

DEV-14 — Configurable extension validation

The extension validation pattern is externalized into the configuration system so users can adjust it for edge cases (e.g., `.numbers`, `.download`, or other legitimate long extensions) without editing source code. The default pattern preserves the original's intent. See §7.

> **Historical note (DEV-14):** The original hardcoded the extension validation regex `^(([a-z0-9]){1,2}|([a-z0-9]){1}([a-z0-9\-]){1,12}([a-z0-9]){1})$` in `MakeObject`. It rejected extensions longer than 14 characters or those containing non-alphanumeric characters (beyond hyphens).

DEV-15 — Unconditional NFC normalization before string hashing

`hash_string()` applies `unicodedata.normalize('NFC', value)` unconditionally on all platforms before encoding to UTF-8 and hashing. This ensures that a file named `café.txt` produces identical identity hashes regardless of whether the filesystem returned the name in NFC (`é` = U+00E9) or NFD (`e` + U+0301). The normalization is unconditional rather than platform-conditional because APFS (macOS) preserves whichever form was used at creation, so NFD filenames can appear on any macOS volume — not just HFS+. Unconditional NFC is the simplest invariant: every string is NFC-normalized before hashing, period.

The same logical filename MUST produce the same identity on all supported platforms. See §6.3 and §15.3.

> **Historical note (DEV-15):** The original hashed string values as-is, using whatever byte representation the OS returned. Because macOS HFS+ stores filenames in NFD (decomposed) form while Windows NTFS and most Linux filesystems store them in NFC (composed) form, the same logical filename would produce different hash digests depending on the platform. The original ran exclusively on Windows (NFC), so this discrepancy was unobservable. The unconditional NFC normalization is a deliberate break from the original's "hash what the filesystem returns" approach to ensure cross-platform hash determinism.

DEV-16 — PyExifTool batch mode as primary exiftool strategy

The `pyexiftool` package (`>=0.5`) is a required runtime dependency. It uses exiftool's `-stay_open` mode to keep a single persistent Perl process alive for the duration of the indexing run. File paths and arguments are written to the process's stdin pipe one per line — this is semantically equivalent to a continuously open argfile and inherits the same Unicode safety guarantees documented in exiftool's FAQ §18 (`-charset filename=utf8`). Per-file exiftool cost drops from 200–500 ms (process startup dominated) to 20–50 ms (metadata extraction only). A `subprocess.run()` + argfile fallback is retained for environments where `pyexiftool` cannot maintain a stable connection. See §6.6 and §17.5.

> **Historical note (DEV-16):** The original invoked exiftool once per file through a Base64-decoded argument file pipeline (`Base64DecodeString` → `TempOpen` → exiftool `-@` argfile → `TempClose`). Every invocation spawned a new Perl process and a new `certutil` process for argument decoding.

3. Repository Structure

This section defines the complete file and directory layout of the `shruggie-indexer` repository. The structure follows the conventions established by `shruggie-feedtools` (see §1.5, External References) — specifically the `src`-layout packaging pattern, the `scripts/` directory for platform-paired build and setup scripts, and the separation of reference documentation from project documentation. Where this section does not explicitly define a convention, the `shruggie-feedtools` repository is the normative reference.

All paths in this section are relative to the repository root unless otherwise noted.

3.1. Top-Level Layout

```
shruggie-indexer/
├── .archive/
```

```
├── .github/
│   ├── copilot-instructions.md
│   └── workflows/
│       ├── docs.yml
│       └── release.yml
├── docs/
│   ├── assets/
│   │   └── images/
│   │       └── gui/
│   ├── getting-started/
│   ├── porting-reference/
│   ├── schema/
│   └── user-guide/
├── scripts/
├── src/
│   └── shruggie_indexer/
├── tests/
├── .gitignore
├── .python-version
├── CHANGELOG.md
├── LICENSE
├── mkdocs.yml
├── pyproject.toml
├── README.md
├── shruggie-indexer-cli.spec
├── shruggie-indexer-gui.spec
├── shruggie-indexer-spec.html
├── shruggie-indexer-spec.md
├── shruggie-indexer-spec.pdf
└── shruggie-indexer.code-workspace
```

Path	Type	Description
`.archive/`	Directory	Human-only workspace for project notes, prompt files, and historical planning artifacts. AI agents should not parse or modify contents of this directory unless explicitly instructed. Not tracked in source control.
`.github/`	Directory	GitHub-specific repository configuration. Contains `copilot-instructions.md` (project-level AI coding guidelines) and `workflows/` with CI/CD pipeline definitions: `release.yml` for the release build pipeline (see §13) and `docs.yml` for automated documentation site deployment to GitHub Pages (see §3.7).
`docs/`	Directory	All project documentation beyond the top-level specification files. Subdivided into `getting-started/` (installation and quick-start guides), `schema/` (canonical v2 JSON Schema and validation examples), `porting-reference/` (historical reference materials from the original implementation), `user-guide/` (end-user documentation including CLI, GUI, configuration, and API reference), and `assets/` (images and media for documentation). See §3.6 and §3.7.
`scripts/`	Directory	Platform-paired shell scripts for development environment setup, build automation, and test execution. See §3.5.
`src/shruggie_indexer/`	Directory	The Python source package. All importable code lives here. See §3.2.
`tests/`	Directory	All test code. Mirrors the source package structure. See §3.4.
`.gitignore`	File	Standard Python `.gitignore` covering `__pycache__/`, `*.pyc`, `.venv/`, `dist/`, `build/`, `*.egg-info/`, `site/`, IDE/editor files, OS artifacts, and PyInstaller working directories.
`.python-version`	File	Contains the string `3.12` (no minor patch). Used by `pyenv` and similar version managers to auto-select the correct interpreter.
`CHANGELOG.md`	File	Project changelog following Keep a Changelog format. Documents all notable changes organized by release version.
`LICENSE`	File	Full Apache 2.0 license text, obtained from https://www.apache.org/licenses/LICENSE-2.0.txt.
`mkdocs.yml`	File	MkDocs configuration for the documentation site. Defines navigation structure, Material for MkDocs theme settings, and plugin configuration. See §3.7.
`pyproject.toml`	File	Centralized project metadata, build system configuration, dependency declarations, entry points, and tool settings (`ruff`, `pytest`, `pyinstaller`). See §13.2.
`README.md`	File	Project overview, installation instructions, quick-start usage examples, and links to full documentation.
`shruggie-indexer-cli.spec`	File	PyInstaller build specification for the CLI executable. Defines packaging configuration, runtime hooks, and binary bundling rules for producing the standalone `shruggie-indexer` command-line binary. See §13.

Path	Type	Description
shruggie-indexer-gui.spec	File	PyInstaller build specification for the GUI executable. Defines packaging configuration, runtime hooks, and binary bundling rules for producing the standalone desktop application binary. See §13.
shruggie-indexer-spec.html	File	HTML rendering of this technical specification, generated via VS Code plugin (Markdown Preview Enhanced or similar). Generated artifact — not manually edited.
shruggie-indexer-spec.md	File	This technical specification. Lives at the repository root for top-level visibility.
shruggie-indexer-spec.pdf	File	PDF rendering of this technical specification, generated via VS Code plugin. Generated artifact — not manually edited.
shruggie-indexer.code-workspace	File	VS Code multi-root workspace configuration. Defines workspace-level editor settings, recommended extensions, and folder mappings for development.

The src-layout (source code under src/shruggie_indexer/ rather than a bare shruggie_indexer/ at the root) is a deliberate choice inherited from shruggie-feedtools. It prevents accidental imports of the development source tree during testing — import shruggie_indexer in tests always resolves to the installed package, not the working directory. This is the layout recommended by the Python Packaging Authority and enforced by hatchling by default.

3.2. Source Package Layout

```
src/shruggie_indexer/
├── __init__.py
├── __main__.py
├── _version.py
├── exceptions.py
├── core/
│   ├── __init__.py
│   ├── traversal.py
│   ├── paths.py
│   ├── hashing.py
│   ├── timestamps.py
│   ├── exif.py
│   ├── sidecar.py
│   ├── entry.py
│   ├── serializer.py
│   └── rename.py
├── models/
│   ├── __init__.py
│   └── schema.py
├── config/
│   ├── __init__.py
│   ├── types.py
│   ├── defaults.py
│   └── loader.py
├── cli/
│   ├── __init__.py
│   └── main.py
└── gui/
    ├── __init__.py
    └── app.py
```

Top-Level Package Files

File	Purpose
__init__.py	Public API surface. Exports the primary programmatic entry points (e.g., index_path(), index_file(), index_directory()) and the configuration constructor. Consumers who import shruggie_indexer interact through this module. See §9.1.
__main__.py	Enables python -m shruggie_indexer invocation. Contains only an import and call to cli.main.main(). No logic beyond the entry-point dispatch.
_version.py	Single source of truth for the package version string: __version__ = "0.1.0". Read by pyproject.toml (via hatchling's version plugin), by __init__.py for the public __version__ attribute, and by the CLI --version flag. This is the same version management pattern used by shruggie-feedtools.
exceptions.py	Defines the exception hierarchy used throughout the package: IndexerError (base), IndexerConfigError, IndexerTargetError, IndexerRuntimeError, RenameError, and IndexerCancellationError. See §9.4, Exception hierarchy.

`core/` — Indexing Engine

The `core/` subpackage contains all indexing logic. Every module in `core/` corresponds to one or more operation categories from the Operations Catalog (§1.5). The CLI and GUI are thin presentation layers that call into `core/` — no indexing logic lives outside this subpackage.

Module	Operations Catalog Categories	Responsibility
`traversal.py`	Cat 1 (Filesystem Traversal & Discovery)	Enumerates files and directories within a target path. Supports recursive and non-recursive modes via a single parameterized function (DEV-03). Applies configurable filesystem exclusion filters. Classifies items as files or directories. Yields items to the entry-construction pipeline.
`paths.py`	Cat 2 (Path Resolution & Manipulation)	The single `resolve_path()` utility (DEV-04) plus path component extraction (parent, name, stem, suffix) and extension validation against the configurable regex pattern (DEV-14). All callers — traversal, hashing, entry construction — use this one module for path operations.
`hashing.py`	Cat 3 (Hashing & Identity Generation)	Provides `hash_file()` (content hashing) and `hash_string()` (name hashing) functions that compute MD5 and SHA256 by default (with SHA512 opt-in) in a single pass (DEV-01, DEV-02). SHA1 is not computed. Applies unconditional NFC Unicode normalization before string hashing (DEV-15). Computes null-hash constants at module load time (DEV-09). Constructs `HashSet` objects. Implements the directory two-layer identity scheme (`hash(hash(name) + hash(parentName))`). Prefixes identities with `y` (file), `x` (directory), or `z` (generated metadata).
`timestamps.py`	Cat 5 (Filesystem Timestamps & Date Conversion)	Derives Unix timestamps (milliseconds) and ISO 8601 strings directly from `os.stat()` results (DEV-07). Handles creation-time portability: `st_birthtime` on macOS, `st_ctime` on Windows, documented fallback on Linux. Constructs `TimestampPair` and `TimestampsObject` model instances.
`exif.py`	Cat 6 (EXIF / Embedded Metadata Extraction)	Invokes `exiftool` using `pyexiftool`'s `-stay_open` batch mode as the primary backend (DEV-05, DEV-16), with a `subprocess.run()` + argfile fallback. Parses JSON output directly with `json.loads()` (DEV-06). Filters unwanted keys via dict comprehension. Respects the configurable exiftool file-type exclusion list. Handles `exiftool` absence gracefully (warning, not fatal).
`sidecar.py`	Cat 7 (Sidecar Metadata File Handling)	Discovers sidecar metadata files by matching filenames against the configurable regex identification patterns from the sidecar configuration. Classifies sidecars by type (Description, JsonMetadata, Hash, Link, Subtitles, Thumbnail, etc.). Reads and parses sidecar content with format-specific handlers (JSON, plain text, hash files, URL/LNK shortcuts). Constructs `MetadataEntry` model instances.
`entry.py`	Cat 8 (Output Object Construction & Schema)	Orchestrates the construction of a single `IndexEntry` from a filesystem path. Calls into `paths`, `hashing`, `timestamps`, `exif`, and `sidecar` to gather all components, then assembles the final v2 schema object.
`serializer.py`	Cat 9 (JSON Serialization & Output Routing)	Converts `IndexEntry` model instances to JSON. Routes output to stdout, a single aggregate file, or per-item in-place sidecar files (`_meta2.json` / `_directorymeta2.json`), depending on the active output mode. Handles pretty-printing vs. compact output. Uses `orjson` as the primary serializer with a `json.dumps()` stdlib fallback for resilience.
`rename.py`	Cat 10 (File Rename & In-Place Write)	Implements the `StorageName` rename operation: renames files and directories from their original names to their hash-based `storage_name` values. Handles collision detection, dry-run mode, and rollback on partial failure.

The `core/__init__.py` file SHOULD re-export the primary orchestration functions (e.g., `index_path`, `index_file`, `index_directory`) so that internal callers can write `from shruggie_indexer.core import index_path` without reaching into individual modules. The individual modules remain importable for callers who need fine-grained access (e.g., `from shruggie_indexer.core.hashing import hash_file`).

`models/` — Data Structures

Module	Responsibility
`schema.py`	Defines the v2 output schema as Python data structures — `IndexEntry`, `NameObject`, `HashSet`, `SizeObject`, `TimestampPair`, `TimestampsObject`, `ParentObject`, `MetadataEntry`, and any supporting types. Implemented as `dataclasses` (the stdlib primitive, chosen for zero-dependency portability of the model layer), with optional Pydantic models behind an import guard for consumers who want runtime schema validation. Includes serialization helpers (`to_dict()`, `to_json()`) that produce schema-compliant output.

Rationale for `models/` as a separate subpackage: In `shruggie-feedtools`, the Pydantic schema models live inside `core/schema.py`. That works for feedtools because its model layer is relatively flat (one `FeedOutput` model with nested item objects). The indexer's model layer is more complex — the v2 schema defines seven distinct sub-object types, the `IndexEntry` itself is recursive (the `items` field contains child `IndexEntry` objects), and the configuration system introduces a parallel set of typed structures (see `config/types.py` below). Separating `models/` from `core/` avoids circular import risks between the

configuration types that `core/` modules consume and the schema types that `core/` modules produce. It also provides a clean import path for external consumers who need the types without pulling in the engine: `from shruggie_indexer.models import IndexEntry`.

> **Historical note:** This is a structural departure from the `shruggie-feedtools` convention of keeping schema models inside `core/`. The departure is justified by the added complexity of the indexer's type hierarchy and the circular-import risk it introduces. If during implementation the `models/` subpackage proves to contain only `schema.py` with no future growth path, collapsing it back into `core/schema.py` is an acceptable simplification — but the separation SHOULD be the starting point.

`config/` — Configuration System

Module	Responsibility
`types.py`	Defines the typed configuration dataclasses: the top-level `IndexerConfig` and any nested structures for metadata file parser settings, exiftool exclusion lists, filesystem exclusion filters, extension validation patterns, and sidecar suffix patterns. See §7.1.
`defaults.py`	Contains the hardcoded default values for every configuration field. This is the baseline configuration that applies when no user configuration file is present. The defaults provide sensible behavior for cross-platform file indexing with comprehensive sidecar pattern coverage. See §7.2.
`loader.py`	Reads TOML configuration files via `tomllib` (stdlib, Python 3.11+), validates their structure against the `types.py` dataclasses, and merges user-provided values over the defaults. Implements the override/merge strategy defined in §7.7. Provides the `load_config()` function consumed by the CLI, GUI, and public API.

The `config/__init__.py` file SHOULD export `IndexerConfig`, `load_config()`, and `get_default_config()` as the public configuration API.

`cli/` — Command-Line Interface

Module	Responsibility
`main.py`	Defines the CLI entry point using `click`. Parses command-line arguments, constructs an `IndexerConfig`, calls into `core/` to perform the requested operation, and routes output via `serializer`. Contains no indexing logic — it is a pure presentation layer. Registered as the `shruggie-indexer` console script entry point in `pyproject.toml`. See §8.

The `cli/` subpackage is intentionally minimal for the MVP. If the CLI grows to support subcommands in future versions, additional modules (e.g., `cli/commands/`) can be added without restructuring.

`gui/` — Graphical User Interface

Module	Responsibility
`app.py`	The standalone desktop GUI application built with CustomTkinter. Uses a two-panel layout with a dark theme. Provides a visual frontend to the same `core/` library code used by the CLI. Shipped as a separate PyInstaller-built executable artifact. See the GUI specification section of this document.

The `gui/` subpackage is isolated from the rest of the package — it imports from `core/`, `models/`, and `config/`, but nothing outside `gui/` imports from it. The `customtkinter` dependency is declared as an optional extra (`pip install shruggie-indexer[gui]`) and is only imported inside `gui/`. This ensures that the CLI and library surfaces function without any GUI dependencies installed.

If the GUI grows in complexity (custom widgets, asset files, multiple views), additional modules and an `assets/` subdirectory can be added under `gui/` without restructuring.

3.3. Configuration File Locations

The indexer's configuration system uses a layered resolution strategy. The following locations are checked in order, with later sources overriding earlier ones:

Priority	Location	Description
1 (lowest)	Compiled defaults	The values in `config/defaults.py`. Always present.
2	User config directory	`~/.config/shruggie-indexer/config.toml` on Linux/macOS, `%APPDATA%\shruggie-indexer\config.toml` on Windows. Per-user persistent configuration.
3	Project/working directory	`./shruggie-indexer.toml` in the current working directory. Per-project overrides.
4 (highest)	CLI flags	Command-line arguments override all file-based configuration.

The user config directory path is resolved using Python's `platformdirs` conventions (or a manual equivalent using `os.environ` lookups for `XDG_CONFIG_HOME` / `APPDATA`). The implementation MUST NOT hardcode platform-specific paths — the resolution logic must work correctly on all three target platforms.

Configuration files are TOML format, parsed by `tomllib` (stdlib). See §7.6 for the file format specification and §7.7 for the merge/override behavior.

No configuration file is required. The tool MUST operate correctly using only compiled defaults. If no configuration files are found at any of the checked locations, the tool proceeds silently with default configuration — it does NOT produce a warning or error about missing configuration files.

3.4. Test Directory Layout

```
tests/
├── conftest.py
├── fixtures/
│   ├── sample_files/
│   ├── sample_trees/
│   ├── sidecar_samples/
│   ├── exiftool_responses/
│   └── config_files/
├── unit/
│   ├── __init__.py
│   ├── test_traversal.py
│   ├── test_paths.py
│   ├── test_hashing.py
│   ├── test_timestamps.py
│   ├── test_exif.py
│   ├── test_sidecar.py
│   ├── test_entry.py
│   ├── test_serializer.py
│   ├── test_rename.py
│   ├── test_schema.py
│   └── test_config.py
├── integration/
│   ├── __init__.py
│   ├── test_single_file.py
│   ├── test_directory_flat.py
│   ├── test_directory_recursive.py
│   ├── test_output_modes.py
│   └── test_cli.py
├── conformance/
│   ├── __init__.py
│   └── test_v2_schema.py
└── platform/
    ├── __init__.py
    ├── test_timestamps_platform.py
    └── test_symlinks_platform.py
```

Directory	Purpose
`conftest.py`	Shared pytest fixtures, temporary directory setup, `exiftool` mock/skip markers, and common test utilities.
`fixtures/`	Static test data files consumed by tests. `sample_files/` contains individual files of various types for single-file indexing tests. `sample_trees/` contains pre-built directory hierarchies for traversal and recursive indexing tests. `sidecar_samples/` contains sidecar metadata files of each supported type. `exiftool_responses/` contains captured JSON outputs for mocking exiftool in unit tests. `config_files/` contains valid, invalid, and partial TOML configuration files for config-loading tests.
`unit/`	Unit tests. Each `test_*.py` file corresponds to the `core/`, `models/`, or `config/` module it exercises. Tests in this directory mock external dependencies (exiftool, filesystem) and validate individual function behavior in isolation.
`integration/`	Integration tests. Exercise the full indexing pipeline end-to-end — from a real filesystem path to a validated JSON output — without mocking the core engine. `exiftool` may still be mocked or skipped (via pytest markers) in CI environments where it is not installed.
`conformance/`	Schema conformance tests. Validate that the JSON output produced by the tool conforms to the v2 JSON Schema definition. These tests load the canonical v2 schema from its published URL (or a local copy) and run `jsonschema` validation against actual indexer output.
`platform/`	Platform-specific tests. Exercise behaviors that vary by operating system — creation-time availability, symlink semantics, case-sensitivity, path-length limits. These tests use pytest markers to conditionally skip on platforms where the tested behavior is not applicable.

The test directory does NOT mirror the `src/shruggie_indexer/` package hierarchy directory-for-directory. Instead, test files are grouped by test type (unit, integration, conformance, platform) with a flat file layout within each group. This is a deliberate choice: the test type grouping is more useful for CI matrix configuration (run unit tests everywhere, run platform tests conditionally) than a structural mirror would be.

Test files MUST be runnable with a bare `pytest` invocation from the repository root. The `pyproject.toml` `[tool.pytest.ini_options]` section configures `testpaths = ["tests"]` and registers custom markers for platform-conditional and exiftool-dependent tests.

3.5. Scripts and Build Tooling

```
scripts/
├── venv-setup.ps1
├── venv-setup.sh
├── build.ps1
├── build.sh
├── test.ps1
└── test.sh
```

Scripts are provided in platform-paired sets: `.ps1` (PowerShell, for Windows) and `.sh` (Bash, for Linux/macOS). This is the same convention used by `shruggie-feedtools`.

Script Pair	Purpose
`venv-setup.ps1` / `venv-setup.sh`	Creates a Python virtual environment (`.venv/`), activates it, installs the package in editable mode (`pip install -e ".[dev,gui]"`), and verifies that the `shruggie-indexer` console script is available. Checks for the correct Python version before proceeding. Idempotent — safe to re-run.
`build.ps1` / `build.sh`	Runs the PyInstaller build to produce standalone executables. Builds both the CLI executable and the GUI executable as separate artifacts. Outputs to `dist/`. See §13.4.
`test.ps1` / `test.sh`	Runs the full test suite via `pytest`. Accepts optional arguments to control scope (e.g., `./test.sh unit` to run only unit tests). Sets up any required environment variables and ensures the virtual environment is active.

Scripts MUST be executable without arguments for the default behavior. Optional arguments (e.g., test scope, build target) are documented in a comment block at the top of each script.

All scripts assume they are invoked from the repository root. They MUST NOT `cd` into subdirectories as part of their operation — all paths within the scripts are relative to the repository root.

The documentation site build configuration (`mkdocs.yml`) lives at the repository root rather than in `scripts/` — it is consumed directly by `mkdocs build` and `mkdocs serve`. See §3.7 for full documentation site details.

3.6. Documentation Artifacts

```
docs/
├── index.md
├── schema/
│   ├── shruggie-indexer-v2.schema.json
│   └── examples/
│       └── flashplayer.exe_meta2.json
├── porting-reference/
│   ├── index.md
│   ├── MakeIndex_DependencyCatalog.md
│   ├── Base64DecodeString_DependencyCatalog.md
│   ├── Date2UnixTime_DependencyCatalog.md
│   ├── DirectoryId_DependencyCatalog.md
│   ├── FileId_DependencyCatalog.md
│   ├── MetaFileRead_DependencyCatalog.md
│   ├── TempOpen_DependencyCatalog.md
│   ├── TempClose_DependencyCatalog.md
│   ├── Vbs_DependencyCatalog.md
│   ├── MakeIndex_OperationsCatalog.md
│   ├── MakeIndex_OutputSchema.json
│   ├── MakeIndex(MetadataFileParser).ps1
│   └── v1-examples/
│       ├── apktool.jar_meta.json
│       ├── exiftool.exe_meta.json
│       ├── flashplayer.exe_meta.json
│       └── SearchMyFiles.chm_meta.json
└── user/
    ├── index.md
    ├── installation.md
    ├── quickstart.md
    ├── configuration.md
    └── changelog.md
```

Path	Purpose
`index.md`	Documentation site landing page. Provides a project overview and links to the three documentation sections (schema reference, porting reference, user guide). Rendered as the site home page by MkDocs. See §3.7.
`schema/`	Canonical v2 JSON Schema definition (`shruggie-indexer-v2.schema.json`) and validation examples (`examples/`). The schema file is the local copy of the canonical schema hosted at schemas.shruggie.tech. The `examples/` subdirectory contains real-world v2-compliant output files used for manual validation reference and documentation.
`porting-reference/`	Reference materials derived from the original PowerShell implementation. These documents inform the tool's design but are not part of the runtime codebase. They include dependency catalogs for each of the eight original pslib functions, the operations catalog mapping original logic to Python modules, the v1 output schema (for reference only — the tool does not target v1), the v1 output examples (`v1-examples/`), and the isolated `MetadataFileParser` object definition. An `index.md` provides a navigable overview for the documentation site. These files are committed to the repository for traceability and historical reference. They are read-only reference artifacts — they are never modified after initial commit unless an error in the documentation is discovered.
`porting-reference/v1-examples/`	Real-world v1 output examples from the original `MakeIndex` function. These files demonstrate the v1 schema structure as produced by the original implementation and serve as backward-compatibility validation fixtures for the eventual v1-to-v2 migration utility.
`user/`	End-user documentation: installation guide (`installation.md`), quick-start tutorial (`quickstart.md`), configuration reference (`configuration.md`), and changelog (`changelog.md`). An `index.md` provides the user guide landing page. Pages are populated incrementally as features stabilize.

Important constraint (reiterated from §1.2): The original PowerShell source code for the `MakeIndex` function — including its complete function body, parameter block, and all nested sub-functions — SHALL NOT be included in the repository in any form. The `MakeIndex(MetadataFileParser).ps1` file in `porting-reference/` is permitted because it contains only the configuration data object, not the function's implementation logic. The dependency catalogs and operations catalog describe behavior in prose, not source code.

3.7. Documentation Site

The project documentation is published as a static site built with MkDocs using the Material for MkDocs theme. MkDocs is a Python-native, Markdown-first static site generator that reads directly from the `docs/` directory — making it the natural choice for a Python project whose documentation artifacts are already authored in Markdown.

3.7.1. Site Configuration

The site is configured by `mkdocs.yml` at the repository root. Key configuration settings:

Setting	Value	Purpose
`site_name`	`shruggie-indexer`	Displayed in the site header and browser title.
`site_description`	Project tagline for SEO and social metadata.	
`site_url`	The GitHub Pages URL for the project.	Base URL for canonical links and sitemap generation.
`docs_dir`	`docs`	MkDocs reads all documentation source from the `docs/` directory.
`theme.name`	`material`	Activates the Material for MkDocs theme.
`theme.features`	Navigation tabs, instant loading, search highlighting, content tabs.	Provides a polished, responsive documentation experience.

The `nav` key in `mkdocs.yml` defines the sidebar navigation structure explicitly rather than relying on directory auto-discovery. This ensures predictable ordering and human-readable section labels:

> **Updated 2026-02-23:** Navigation structure updated to reflect the current `mkdocs.yml`. The "Getting Started" section was added to separate onboarding from the User Guide. The "Desktop Application" page was added to the User Guide. The Changelog was promoted to a top-level nav item. Paths were corrected to match the actual `docs/` directory structure.

```
nav:
  - Home: index.md
  - Getting Started:
      - Installation: getting-started/installation.md
      - Quick Start: getting-started/quickstart.md
      - ExifTool Setup: getting-started/exiftool.md
  - User Guide:
      - Overview: user-guide/index.md
      - Desktop Application: user-guide/gui.md
      - CLI Reference: user-guide/cli-reference.md
      - Configuration: user-guide/configuration.md
```

```
      - Python API: user-guide/python-api.md
      - Platform Notes: user-guide/platform-notes.md
  - Schema Reference:
      - Overview: schema/index.md
  - Porting Reference:
      - Overview: porting-reference/index.md
      - Operations Catalog: porting-reference/MakeIndex_OperationsCatalog.md
      - Dependency Catalogs:
          - MakeIndex: porting-reference/MakeIndex_DependencyCatalog.md
          - Base64DecodeString: porting-reference/Base64DecodeString_DependencyCatalog.md
          - Date2UnixTime: porting-reference/Date2UnixTime_DependencyCatalog.md
          - DirectoryId: porting-reference/DirectoryId_DependencyCatalog.md
          - FileId: porting-reference/FileId_DependencyCatalog.md
          - MetaFileRead: porting-reference/MetaFileRead_DependencyCatalog.md
          - TempOpen: porting-reference/TempOpen_DependencyCatalog.md
          - TempClose: porting-reference/TempClose_DependencyCatalog.md
          - Vbs: porting-reference/Vbs_DependencyCatalog.md
  - Changelog: changelog.md
```

3.7.2. Non-Markdown Asset Handling

The `docs/` directory contains non-Markdown files (`.json`, `.ps1`) that are reference artifacts rather than documentation pages. MkDocs copies these files to the built site as static assets. They are linked from their parent section's `index.md` page as downloadable files rather than rendered as documentation content. The navigation entries for `.json` and `.ps1` files produce direct download links in the built site.

3.7.3. Build and Preview

Command	Purpose
`mkdocs serve`	Starts a local development server with live reload. Used during documentation authoring to preview changes. Serves at `http://127.0.0.1:8000/` by default.
`mkdocs build`	Produces the static site in the `site/` directory. The `site/` directory is listed in `.gitignore` and is never committed.

3.7.4. Deployment

The documentation site is deployed to GitHub Pages via a dedicated GitHub Actions workflow (`.github/workflows/docs.yml`). The workflow:

- **Triggers** on push to `main` when files in `docs/` or `mkdocs.yml` change.
- **Builds** the site using `mkdocs build --strict` (strict mode fails the build on warnings such as broken links or missing pages).
- **Deploys** the built `site/` directory to the `gh-pages` branch using `mkdocs gh-deploy --force`.

The `--strict` flag ensures that documentation quality is enforced in CI — broken internal links, missing navigation targets, and unreferenced pages cause build failures rather than silent degradation.

3.7.5. Dependencies

`mkdocs` and `mkdocs-material` are added as optional development dependencies in `pyproject.toml` under a `[project.optional-dependencies]` docs group:

```toml
[project.optional-dependencies]
docs = [
    "mkdocs>=1.6",
    "mkdocs-material>=9.5",
]
```

These packages are NOT required for using, developing, or testing `shruggie-indexer` itself. They are required only for building or previewing the documentation site. The `venv-setup` scripts (§3.5) do not install the `docs` extra by default — documentation authors install it explicitly with `pip install -e ".[docs]"`.

4. Architecture

This section defines the high-level architecture of `shruggie-indexer` — the processing pipeline, module decomposition, data flow, state management, error handling strategy, and entry point routing. Where design decisions have historical context rooted in the project's origins, that lineage is noted in callout blocks.

The module-level detail here complements the source package layout in §3.2 but focuses on **behavioral relationships** between modules — who calls whom, what data crosses each boundary, and what invariants each layer is responsible for maintaining. §6 (Core Operations) provides the per-operation behavioral

contracts; this section provides the structural skeleton that those operations hang on.

4.1. High-Level Processing Pipeline

Every invocation of `shruggie-indexer` — whether from the CLI, the GUI, or the Python API — passes through the same linear pipeline. The pipeline has six stages, executed in strict order. No stage begins until its predecessor completes for the current item, though the pipeline as a whole operates on one item at a time within a traversal loop.

Stage 1 — Configuration Resolution. Load compiled defaults, merge any user configuration file (§7), and apply CLI/API overrides. Produce a fully-resolved, immutable `IndexerConfig` object. This happens exactly once per invocation.

Stage 2 — Target Resolution and Classification. Resolve the input target path to an absolute canonical form. Classify the target as one of three types: single file, single directory (flat), or directory tree (recursive). This classification determines which traversal strategy is used in Stage 3. If the path does not exist or is not accessible, the pipeline terminates with an error at this stage.

Stage 3 — Traversal and Discovery. Enumerate the items to be indexed. For a single file, the "traversal" is trivial — the item set contains only the target. For a directory (flat or recursive), the traversal yields files and subdirectories according to the recursion mode, filtering out excluded paths per the configuration. The traversal produces an ordered sequence of filesystem paths to process.

Stage 4 — Entry Construction. For each path yielded by Stage 3, build a complete `IndexEntry` (the v2 schema object defined in §5). This is the core of the indexing engine and involves:

- Path component extraction (name, stem, suffix, parent).
- Symlink detection.
- Hash computation (content hashes for files, name hashes for directories, name hashes for both).
- Identity generation (selecting the `id` from the chosen algorithm, applying the `y`/`x`/`z` prefix).
- Timestamp extraction (accessed, created, modified — in both Unix-millisecond and ISO 8601 forms).
- EXIF/embedded metadata extraction (via `exiftool`, if applicable and available).
- Sidecar metadata discovery, parsing, and optional merging.
- Parent identity computation.
- Assembly of the final `IndexEntry` model instance.

For directory entries in recursive mode, Stage 4 recurses into child items and attaches their completed `IndexEntry` objects to the parent's `items` field before the parent entry is considered complete.

Stage 5 — Output Routing. Route the completed entry (or entry tree) to one or more output destinations based on the configured output mode: stdout, a single aggregate file, and/or per-item in-place sidecar files. Serialization to JSON occurs at this stage. In-place sidecar writes happen during traversal (Stage 3–4 loop) so that partial results survive interruption; stdout and aggregate file writes happen after the full entry tree is assembled.

Stage 6 — Post-Processing. Execute deferred operations that must occur after all indexing is complete: MetaMergeDelete file removal (if active), elapsed-time logging, and final status reporting.

> **Historical note:** The original's post-processing stage also performed global variable cleanup (`Remove-Variable` on each promoted `$global:` variable). This is unnecessary in the Python implementation due to the no-global-state design (see §4.4).

The stages map to the tool's module structure as follows:

Stage	Module(s)
1 — Configuration	`config/loader.py`
2 — Target Resolution	`core/paths.py`, `cli/main.py`
3 — Traversal	`core/traversal.py`
4 — Entry Construction	`core/entry.py` (orchestrator), `core/hashing.py`, `core/timestamps.py`, `core/exif.py`, `core/sidecar.py`, `core/paths.py`
5 — Output Routing	`core/serializer.py`, `core/rename.py`
6 — Post-Processing	`core/serializer.py` (finalization), top-level orchestrator

Pipeline linearity

Traversal (Stage 3), entry construction (Stage 4), and output routing (Stage 5) are separated at the module boundary level. Entry construction knows nothing about where its output goes. Output routing knows nothing about how entries were built. The one deliberate exception is in-place sidecar writes, which occur within the traversal loop so that partial results survive interruption — but even there, the serializer module is called as a service rather than being inlined into the traversal logic.

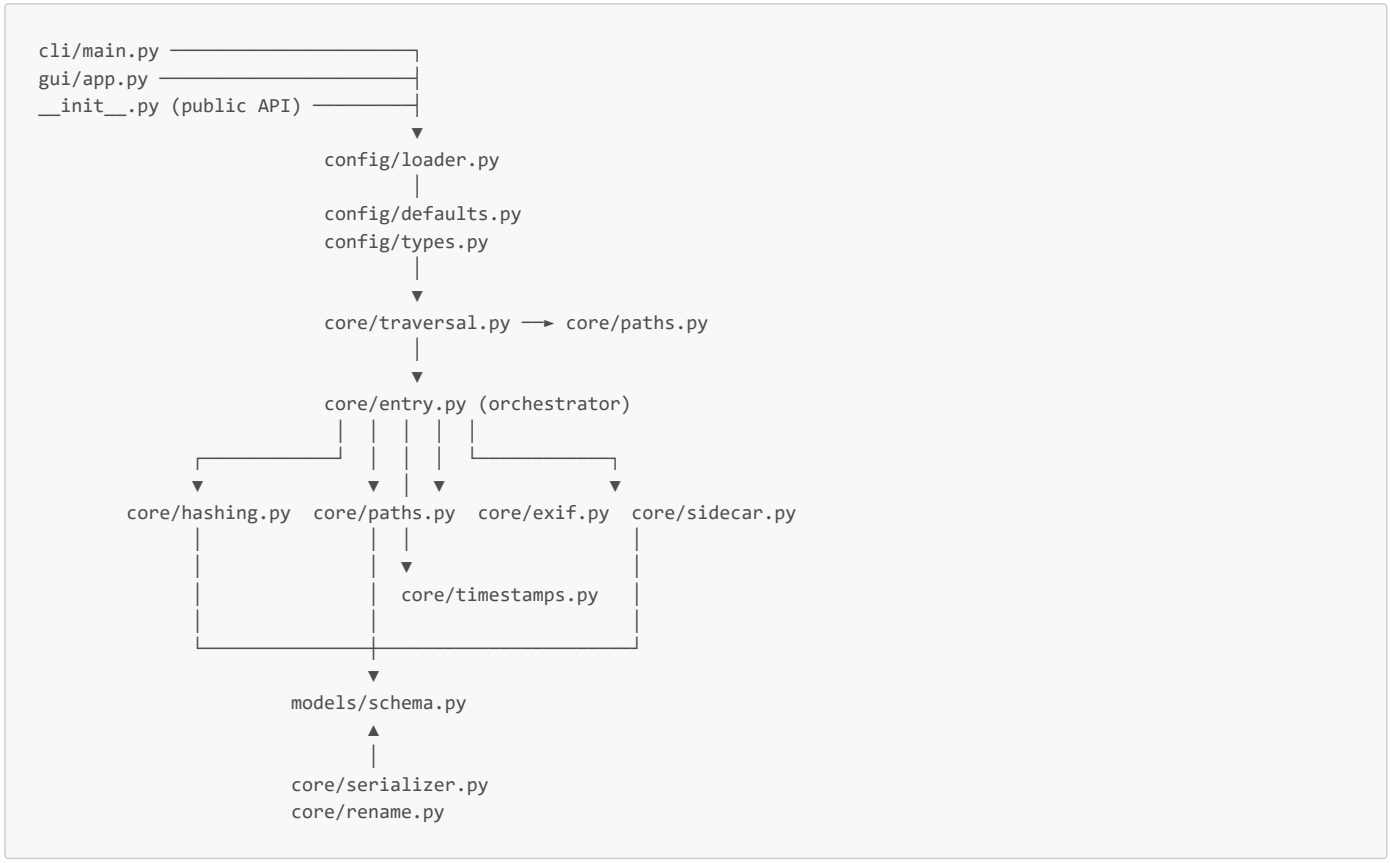
> **Historical note:** The original interleaved traversal, entry construction, output writing, and rename operations within single functions (`MakeDirectoryIndexLogic` and `MakeDirectoryIndexRecursiveLogic`) — a single function handled discovery, construction, serialization, and file mutation in a tightly coupled loop. The clean separation of concerns in the current architecture eliminates this coupling.

4.2. Module Decomposition

This subsection describes the dependency relationships between modules. §3.2 defines each module's responsibility in isolation; this section defines who calls whom and why.

Dependency Graph

The following graph shows the runtime call relationships between the tool's Python modules. Arrows point from caller to callee. Modules are grouped by subpackage.



Key structural rules:

Rule 1 — Presentation layers are thin. `cli/main.py`, `gui/app.py`, and the public API (`__init__.py`) contain no indexing logic. They perform argument parsing or UI event handling, construct an `IndexerConfig`, call into `core/`, and format the result for their respective output medium. This is design goal G3 from §2.3.

Rule 2 — `core/entry.py` is the sole orchestrator. All coordination between the component modules (hashing, timestamps, exif, sidecar, paths) happens inside `entry.py`. No component module calls another component module directly — `hashing.py` does not call `paths.py`, `exif.py` does not call `hashing.py`. Each component module receives its inputs as function arguments and returns its outputs as return values. `entry.py` wires them together.

> **Historical note:** The original's `MakeObject` function called `FileId`, `DirectoryId`, `Date2UnixTime`, `GetFileExif`, `GetFileMetaSiblings`, and `ReadMetaFile` directly, which in turn each internally called their own copies of path resolution and hashing logic. This created a web of implicit dependencies and the code duplication cataloged in §2.6 (DEV-01 through DEV-04). The hub-and-spoke model through `entry.py` eliminates this duplication by making all shared operations explicit dependencies injected by the orchestrator.

Rule 3 — `models/schema.py` is a leaf dependency. The schema model types (`IndexEntry`, `HashSet`, `TimestampPair`, etc.) are pure data structures with no business logic and no imports from `core/` or `config/`. Every `core/` module that produces or consumes schema objects imports from `models/`, but `models/` never imports from `core/`. This prevents circular imports and keeps the data model independently testable.

Rule 4 — `config/` is consumed, not called back into. Configuration flows one direction: from `config/loader.py` into the calling code, and then down through the `core/` module call chain as function parameters. No `core/` module reaches back into `config/loader.py` to re-read configuration at runtime. The `IndexerConfig` object is constructed once and threaded through as an argument.

Rule 5 — `gui/` is isolated. The `gui/` subpackage imports from `core/`, `models/`, and `config/`, but nothing outside `gui/` imports from it. The `customtkinter` dependency is only imported inside `gui/`. Removing the `gui/` subpackage entirely has zero effect on the CLI or library surfaces.

Module count

The tool comprises 10 `core/` modules, 1 `models/` module, and 3 `config/` modules. This represents a roughly 4:1 consolidation over the equivalent scope in the original PowerShell implementation, achieved primarily by eliminating the hashing duplication (DEV-01), path resolution duplication (DEV-04), and the five eliminated dependencies (DEV-05 through DEV-08, DEV-13).

> **Historical note:** The original codebase contained approximately 60 discrete code units across the `MakeIndex` function body, its 20+ inline sub-functions, the 8 external pslib dependencies, and their own internal sub-functions.

4.3. Data Flow

This subsection traces the data that flows through the pipeline for the most common operation — indexing a directory recursively — and identifies the types that cross module boundaries.

Primary data types at module boundaries

Boundary	Data Crossing	Type
CLI/API → `config/loader`	Raw CLI arguments or API keyword arguments	`dict` / keyword args
`config/loader` → caller	Fully-resolved configuration	`IndexerConfig` (frozen dataclass)
CLI/API → `core/traversal`	Target path + config + recursion flag	`Path`, `IndexerConfig`, `bool`
`core/traversal` → `core/entry`	Individual filesystem path to index	`Path` (yielded one at a time)
`core/entry` → `core/paths`	Raw `Path` for component extraction	`Path` → `str` (name, stem, suffix, parent name)
`core/entry` → `core/hashing`	File path (for content hashing) or string (for name hashing)	`Path` or `str` → `HashSet`
`core/entry` → `core/timestamps`	`os.stat_result`	`os.stat_result` → `TimestampsObject`
`core/entry` → `core/exif`	File path + config (exclusion list)	`Path`, `IndexerConfig` → `dict` or `None`
`core/entry` → `core/sidecar`	File path + parent directory listing + config (patterns)	`Path`, `list[Path]`, `IndexerConfig` → `list[MetadataEntry]`
`core/entry` → caller	Completed index entry	`IndexEntry`
Caller → `core/serializer`	Completed entry tree + output mode config	`IndexEntry`, `IndexerConfig` → JSON `str` or file writes
Caller → `core/rename`	Completed entry + original path	`IndexEntry`, `Path` → renamed `Path`

Recursive directory data flow (detailed)

The following walkthrough traces data through the system for a recursive directory indexing operation — the most complex scenario — in sufficient detail for an implementer to understand the wiring.

1. The caller (CLI, GUI, or API) invokes `index_path(target, config)` where `target` is a `Path` to a directory and `config` is the resolved `IndexerConfig` with `recursive=True`.
2. `index_path()` calls `core/paths.resolve_path(target)` to canonicalize the target. It then calls `target.stat()` and `target.is_dir()` to classify the target as a directory.
3. `index_path()` calls `core/entry.build_directory_entry(target, config, recursive=True)`.
4. Inside `build_directory_entry()`:

   a. Path components are extracted via `core/paths`: `target.name`, `target.parent.name`.

   b. Symlink status is checked: `target.is_symlink()`.

   c. Directory identity is computed via `core/hashing.hash_directory(name, parent_name)`, which implements the two-layer `hash(hash(name) + hash(parent_name))` scheme and returns a `HashSet`.

   d. An `id` is selected from the `HashSet` based on `config.id_algorithm` and prefixed with `x`.

   e. Timestamps are extracted via `core/timestamps.extract_timestamps(target.stat())`, returning a `TimestampsObject`.

   f. Parent identity is computed via `core/hashing.hash_directory(parent_name, grandparent_name)`, returning a `ParentObject`.

   g. `core/traversal.list_children(target, config)` yields child paths, separated into files and subdirectories. The files-first, directories-second ordering from the original is preserved.

   h. For each child file: `core/entry.build_file_entry(child_path, config)` is called, producing a child `IndexEntry`. This call internally invokes `hashing.hash_file()` for content hashing, `exif.extract_exif()` for embedded metadata, and `sidecar.discover_and_parse()` for sidecar metadata.

i. For each child subdirectory: `core/entry.build_directory_entry(child_path, config, recursive=True)` is called recursively, producing a child `IndexEntry` with its own `items` list.

j. The child `IndexEntry` objects are collected into the parent's `items` list. The directory's `size` is computed as the sum of all child sizes.

k. If in-place output mode is active, `core/serializer.write_inplace(entry, target)` writes the current entry's sidecar file before returning.

l. The completed `IndexEntry` for the directory is returned.

5. Back in `index_path()`, the completed root `IndexEntry` (containing the full recursive tree) is passed to `core/serializer` for output routing.
6. If MetaMergeDelete is active, the accumulated delete queue (a `list[Path]` built up during sidecar discovery in step 4h) is iterated and files are removed.

Data flow invariant

Every `core/` module function is a **pure data transformation** with respect to the index entry being built — it receives input arguments and returns output values without modifying global state, writing to the filesystem (except `serializer` and `rename`), or communicating with other modules via side channels. The only exceptions are:

- `core/exif.py` invokes an external subprocess (`exiftool`). This is an I/O side effect but does not mutate program state.
- `core/serializer.py` writes JSON to the filesystem or stdout. This is the intended terminal side effect.
- `core/rename.py` renames files on the filesystem. This is the intended terminal side effect.
- `core/sidecar.py` reads sidecar file contents from the filesystem. This is a read-only I/O operation.

This property makes the core indexing logic straightforward to unit-test: mock the filesystem interactions (stat, read, subprocess) and every module becomes a deterministic function from inputs to outputs.

4.4. State Management

Design principle: no mutable global state

`shruggie-indexer` uses no mutable global state. All shared data flows through one of two mechanisms:

1. **Function parameters.** Configuration, file paths, and intermediate results are passed as explicit arguments down the call chain. Every function's data dependencies are visible in its signature.
2. **Return values.** Results flow upward through return values and are collected by the orchestrator (`entry.py`) or the top-level caller.

There are no module-level mutable variables, no singleton objects with hidden state, and no cleanup blocks needed at the end of an invocation. The module decomposition (§4.2) makes explicit parameter passing sufficient for all shared data flow.

> **Historical note:** The original `MakeIndex` relied heavily on mutable global state. The `$global:MetadataFileParser` object, `$global:ExiftoolRejectList`, `$global:MetaSuffixInclude`, `$global:MetaSuffixExclude`, `$global:MetaSuffixIncludeString`, `$global:MetaSuffixExcludeString`, `$global:DeleteQueue`, and `$LibSessionID` were all script-level or explicitly `$global:`-promoted variables that were read and written by deeply nested sub-functions across the call tree. At the end of `MakeIndex`, a cleanup block called `Remove-Variable` on each promoted global to prevent state leakage between invocations within the same PowerShell session. Python's import system and explicit parameter passing eliminate the need for this pattern entirely.

State objects and their lifetimes

Object	Created	Lifetime	Mutability	Scope
`IndexerConfig`	Stage 1 (configuration resolution)	Entire invocation	Immutable (frozen dataclass)	Passed as argument to every `core/` function that needs configuration
`IndexEntry`	Stage 4 (entry construction), one per item	Until serialized and output in Stage 5	Built incrementally during Stage 4, immutable after construction completes	Returned by `build_file_entry()` / `build_directory_entry()`, collected by parent entries or the top-level caller
Delete queue	Built during Stage 4 sidecar processing	Until Stage 6 post-processing	Append-only `list[Path]`	Created by the top-level orchestrator function and passed into sidecar processing; iterated during Stage 6
Session ID	Generated at startup	Entire invocation	Immutable (`str`)	Injected into the logging system via a `logging.Filter`; not passed through the indexing call chain
Logger instances	Created at import time (module-level)	Process lifetime	Mutable (log level can be reconfigured)	Per-module via `logging.getLogger(__name__)`

The delete queue

The delete queue warrants specific attention because it is the one piece of cross-cutting mutable state preserved from the original design. When MetaMergeDelete is active, sidecar files that have been successfully merged into their parent entry's `metadata` array are queued for deletion. The deletion itself is deferred to Stage 6 (post-processing) rather than happening inline during traversal, for safety: if the process is interrupted mid-traversal, no sidecar files have been deleted, and the partially-written index entries still reference the sidecar paths.

The delete queue is implemented as a plain `list[Path]` owned by the top-level orchestrator function (not a global variable). It is passed into `core/sidecar.discover_and_parse()` as a parameter, and that function appends paths to it. After the traversal loop completes and all output has been written, the orchestrator iterates the queue and calls `Path.unlink()` on each entry. Errors during deletion are logged as warnings, not raised as exceptions — a failure to delete one sidecar file does not abort the deletion of others.

> **Historical note:** The original's `$global:DeleteQueue` was a `$global:`-scoped `ArrayList` that `ReadMetaFile` appended to directly. The current design makes the delete queue an explicit parameter rather than a global, consistent with the no-global-state principle. The behavioral contract is identical — accumulate during traversal, drain after completion — but the ownership is explicit rather than ambient.

4.5. Error Handling Strategy

Design principle: fail per-item, not per-invocation

When indexing a directory tree that may contain thousands of items, a single unreadable file, a permission error, or a corrupt metadata file MUST NOT abort the entire operation. The error handling follows a **per-item isolation** strategy: errors encountered while processing a single file or directory are caught, logged, and result in that item being either skipped or partially populated, while the traversal continues with the next item.

> **Historical note:** The original follows this same principle in practice — most errors within `MakeObject` are caught, logged via `Vbs`, and result in `$null` values for the affected fields. The current implementation formalizes this into an explicit, tiered strategy.

Error severity tiers

Tier	Behavior	Examples	Historical Equivalent
Fatal	Abort the entire invocation. Exit with a non-zero code.	Target path does not exist. Target path is not readable. Invalid configuration file syntax.	Implicit — the original did not cleanly distinguish these, but equivalent conditions caused PowerShell terminating errors.
Item-level	Log a warning. Skip the affected item entirely (exclude it from the output). Continue processing remaining items.	Permission denied reading a file. Filesystem error during `stat()`. Symlink target does not exist (and fallback hashing also fails).	Corresponds to `Vbs -Status e` messages within `MakeObject` followed by returning `$null` for the item.
Field-level	Log a warning. Populate the affected field with `null` (or its type-appropriate absence value). Include the item in the output with the affected field empty. Continue processing the current item.	`exiftool` not installed (EXIF metadata will be `null`). `exiftool` returns an error for a specific file. A sidecar metadata file exists but contains malformed JSON. Timestamp extraction fails for one timestamp type.	Corresponds to `Vbs -Status w` or `Vbs -Status e` messages within sub-functions, with the field set to `$null`.
Diagnostic	Log at debug level. No effect on output.	A file matched the exiftool exclusion list and was skipped. A directory matched the filesystem exclusion filter and was skipped. A sidecar pattern regex matched but the file had no parseable content.	Corresponds to `Vbs -Status d` or `Vbs -Status i` messages.

Implementation pattern

Every `core/` module function that performs I/O or processes untrusted input (file content, exiftool output, sidecar file content) follows this pattern:

```
# Illustrative — not the exact implementation.
def extract_exif(path: Path, config: IndexerConfig) -> dict | None:
    """Extract EXIF metadata. Returns None if extraction fails or is skipped."""
    if path.suffix.lower() in config.exiftool_exclude_extensions:
        logger.debug("Skipping exiftool for excluded extension: %s", path.suffix)
        return None
    try:
        result = subprocess.run(
            ["exiftool", "-json", "-n", str(path)],
            capture_output=True, text=True, timeout=30,
        )
        if result.returncode != 0:
            logger.warning("exiftool returned non-zero for %s: %s", path, result.stderr)
```

```
            return None
        data = json.loads(result.stdout)
        # ... filter unwanted keys ...
        return filtered_data
    except FileNotFoundError:
        logger.warning("exiftool not found on PATH; EXIF extraction disabled")
        return None
    except subprocess.TimeoutExpired:
        logger.warning("exiftool timed out for %s", path)
        return None
    except (json.JSONDecodeError, KeyError, IndexError) as exc:
        logger.warning("Failed to parse exiftool output for %s: %s", path, exc)
        return None
```

The orchestrator (`entry.py`) checks the return value and populates the corresponding `IndexEntry` field with `None` (which serializes to `null` in JSON) when a component returns a failure signal. The orchestrator does not catch exceptions from component modules — the components are responsible for catching their own anticipated failure modes and returning clean failure values. Unanticipated exceptions (bugs) propagate upward to the top-level caller, where a final catch-all logs the error and, depending on the invocation mode, either skips the item (directory traversal) or terminates with a non-zero exit code (single-file mode).

Exiftool availability

`exiftool` occupies a unique position in the error model: it is the only external binary dependency, and its absence is a **configuration-time condition**, not a per-item error. The tool checks for `exiftool` availability once during Stage 1 or at the first attempted invocation, caches the result, and uses it to gate all subsequent exiftool calls. If `exiftool` is not found, a single warning is emitted (not per-file) and all EXIF metadata fields are populated with `null` for the entire invocation. This avoids the performance cost and log noise of repeatedly spawning a doomed subprocess.

> **Historical note:** The original invoked `exiftool` via `GetFileExifRun` for every eligible file without first checking whether the binary exists. If `exiftool` was missing, each invocation failed independently, producing a per-file error. The probe-once approach is both more efficient and more user-friendly.

4.6. Entry Point Routing

Three input scenarios determine how the pipeline executes. The classification is performed once, during Stage 2 (target resolution), and dictates the traversal strategy for Stage 3.

Input classification

Scenario	Condition	Target Type Code	Pipeline Behavior
Single file	Target path exists and is a file (or a symlink to a file)	`file`	No traversal. `build_file_entry()` is called once on the target. Output is a single `IndexEntry` (no `items` field).
Directory (flat)	Target path exists and is a directory; recursive mode is not requested	`directory_flat`	`list_children()` enumerates immediate children. `build_directory_entry()` constructs the parent entry with a single level of child entries in its `items` list. No descent into subdirectories.
Directory (recursive)	Target path exists and is a directory; recursive mode is requested	`directory_recursive`	`build_directory_entry()` recurses depth-first into all subdirectories. The result is a fully nested tree of `IndexEntry` objects mirroring the filesystem hierarchy.

The three-way dispatch is implemented with two functions that compose naturally:

- `build_file_entry(path, config) → IndexEntry` — handles a single file.
- `build_directory_entry(path, config, recursive) → IndexEntry` — handles a directory. When `recursive=True`, it calls itself for child subdirectories and calls `build_file_entry()` for child files. When `recursive=False`, it does the same for immediate children only, without descending.

Both functions delegate to the same component modules (hashing, timestamps, exif, sidecar). There is no duplicated wiring between the file path and the directory path — only the presence or absence of the `items` assembly loop.

Routing decision tree

The following pseudocode shows the complete routing logic performed by the top-level `index_path()` function. This is the single entry point consumed by the CLI, GUI, and public API.

```
# Illustrative — not the exact implementation.
def index_path(target: Path, config: IndexerConfig) -> IndexEntry:
    resolved = resolve_path(target)
```

```
    if not resolved.exists():
        raise IndexerError(f"Target does not exist: {resolved}")

    if resolved.is_file() or (resolved.is_symlink() and not resolved.is_dir()):
        return build_file_entry(resolved, config)

    if resolved.is_dir():
        return build_directory_entry(resolved, config, recursive=config.recursive)

    raise IndexerError(f"Target is neither a file nor a directory: {resolved}")
```

The `config.recursive` flag is the sole control that distinguishes the flat-directory and recursive-directory scenarios. The CLI SHOULD still accept `--file` and `--directory` flags for explicit disambiguation (e.g., when indexing a symlink whose target type is ambiguous), but these flags refine the classification rather than selecting between separate code paths.

> **Historical note:** The original's `Recursive`, `Directory`, and `File` switches were three separate parameters requiring mutual-exclusion validation. Its `TargetTyp` routing selected between three essentially-independent code paths (`MakeDirectoryIndexRecursive`, `MakeFileIndex`, `MakeDirectoryIndex`), each of which was a wrapper that called `MakeObject` differently and contained its own traversal, output-writing, and rename logic. The current routing selects between two functions that share all component modules and differ only in whether a traversal loop is present, eliminating the structural duplication without changing the logical behavior.

Symlink routing edge case

When the target path is a symlink, the classification follows the symlink's target type (file or directory) for routing purposes, but the `is_link` flag on the resulting `IndexEntry` is set to `True`. Content hashing falls back to name hashing for symlinked files (because the link target may be inaccessible or on a different filesystem), and exiftool is skipped for symlinks.

> **Historical note:** The original's `FileId` and `DirectoryId` both check for the reparse-point attribute and switch to name-based hashing when it is present.

If the symlink target does not exist (a dangling symlink), the item is treated as an item-level error: a warning is logged, and the item is either skipped or included with degraded fields (null hashes, null timestamps), depending on what information can be recovered from `os.lstat()` (which reads the symlink itself, not its target).

> **Historical note:** The original did not explicitly handle dangling symlinks — the PowerShell `Get-Item` call would fail, and the error would propagate in a platform-dependent way. The explicit handling here is a minor robustness improvement.

5. Output Schema

This section defines the complete v2 output schema for `shruggie-indexer` — the structure, field definitions, type constraints, nullability rules, and behavioral guidance for every field in an `IndexEntry`. The canonical machine-readable schema is the JSON Schema document at schemas.shruggie.tech/data/shruggie-indexer-v2.schema.json. This section interprets and extends that schema with implementation guidance, v1-to-v2 mapping context, and behavioral notes that a JSON Schema cannot express — but does not supersede the canonical schema for structural or type-level definitions. Where a conflict exists between this section and the canonical schema, the canonical schema governs field names, types, and `required` constraints; this section governs behavioral semantics and implementation strategy.

The v2 schema is a ground-up restructuring of the earlier MakeIndex v1 output format (`MakeIndex_OutputSchema.json`). `shruggie-indexer` targets v2 exclusively and does not produce v1 output. A v1-to-v2 migration utility for converting existing v1 index assets is a planned post-MVP deliverable (see §1.2, Out of Scope).

5.1. Schema Overview

Design principles

The v2 schema is governed by five design principles that drove the restructuring from v1.

P1 — Logical grouping. Related fields are consolidated into typed sub-objects rather than scattered across the top level. In v1, a file's name and its name hashes are separate top-level keys (`Name`, `NameHashes`); in v2, they are a single `NameObject` with `text` and `hashes` properties. The same consolidation applies to timestamps (`TimestampPair` pairs an ISO string with a Unix integer), sizes (`SizeObject` pairs a human-readable string with a byte count), and parent relationships (`ParentObject` groups the parent's ID and name). This eliminates the implicit coupling between field pairs that existed in v1 and makes the schema self-documenting: every sub-object is a complete, independently meaningful unit.

P2 — Single discriminator for item type. v1 uses a dual-boolean pattern (`IsDirectory: true/false` and an implied file/directory distinction from field presence) that is ambiguous in edge cases and requires consumers to perform boolean logic. v2 replaces this with a single `type` enum (`"file"` or `"directory"`). Combined with the `schema_version` discriminator (absent in v1), consumers can route parsing logic unambiguously from the first two fields of any entry.

P3 — Provenance tracking for metadata entries. v1's `Metadata` array entries carry `Source`, `Type`, `Name`, `NameHashes`, and `Data` — enough to describe the metadata content but not enough to reconstruct the original sidecar file that the content came from. v2's `MetadataEntry` adds `origin` (generated vs. sidecar), `file_system` (relative path of the original sidecar file), `size`, `timestamps`, and an `attributes` sub-object (type classification, format, transforms).

This makes the `metadata` array a complete manifest for MetaMergeDelete reversal: every sidecar entry carries enough information to reconstruct the original file on disk.

P4 — Elimination of redundancy and platform coupling. v1 includes fields that are structurally redundant (`BaseName` duplicates the stem of `Name`, `Ids` and `ContentHashes` carry overlapping hash values for files), platform-specific (`Encoding` is a .NET `System.Text.Encoding` serialization), or algorithmically redundant (`SHA1` is carried alongside MD5 and SHA256 despite serving no unique purpose in the identity scheme). v2 drops these fields and normalizes what remains. See §5.11 for the full inventory.

P5 — Explicit algorithm selection. v1's `_id` field is derived from one of the hash algorithms (MD5 or SHA256) but the schema does not record which algorithm was used — consumers must reverse-match the `_id` value against the `Ids` object to determine the algorithm. v2 adds `id_algorithm` as an explicit top-level field, making the identity derivation fully self-describing.

Canonical schema location

The canonical v2 JSON Schema is hosted at:

```
https://schemas.shruggie.tech/data/shruggie-indexer-v2.schema.json
```

This document uses JSON Schema Draft-07 (`https://json-schema.org/draft-07/schema#`). The schema `$id` is set to the canonical URL. The schema `title` is `IndexEntry`.

Schema version

Every v2 output document includes a `schema_version` field with the integer value `2`. This field is the first property in the serialized JSON output (by convention, not by requirement — JSON objects are unordered, but the serializer SHOULD place `schema_version` first for readability). The value is a `const` constraint in the JSON Schema: `{ "type": "integer", "const": 2 }`. Consumers SHOULD check this field before attempting to parse the remainder of the document and SHOULD reject documents with an unrecognized schema version.

The v1 schema has no version discriminator. The absence of a `schema_version` field (or the presence of the v1-specific `_id` field with its `y`/`x` prefix) is sufficient to identify a v1 document, but consumers SHOULD NOT rely on field absence for version detection — the eventual v1-to-v2 migration utility will handle schema identification as part of its conversion logic.

5.2. Reusable Type Definitions

The v2 schema defines six reusable type definitions in the `definitions` block of the JSON Schema. These are the building blocks from which the top-level `IndexEntry` properties and the nested `MetadataEntry` objects are composed. Each definition is referenced via `$ref` wherever it appears.

The Python implementation SHOULD model these definitions as individual `dataclass` types (or Pydantic models behind an import guard) in `models/schema.py`. This mirrors the schema's compositional structure and gives every sub-object a named type for static analysis, IDE support, and independent testability. See §3.2 for the module location rationale.

5.2.1. HashSet

A `HashSet` is a collection of cryptographic hash digests for a given input. All hash values are uppercase hexadecimal strings (characters `0–9`, `A–F`).

Property	Type	Required	Description
`md5`	`string`	Yes	MD5 digest. 32 hex characters. Pattern: `^[0-9A-F]{32}$`.
`sha256`	`string`	Yes	SHA-256 digest. 64 hex characters. Pattern: `^[0-9A-F]{64}$`.
`sha512`	`string`	No	SHA-512 digest. 128 hex characters. Pattern: `^[0-9A-F]{128}$`. Included when configured or when additional verification strength is warranted.

`additionalProperties` is `false` — no extra keys are permitted.

v1 comparison: v1 defines hash fields as separate top-level objects (`Ids`, `NameHashes`, `ContentHashes`, `ParentIds`, `ParentNameHashes`) each with their own `MD5`, `SHA1`, `SHA256`, `SHA512` properties. v2 replaces all of these with `HashSet` references.

SHA1 removal: v1 includes `SHA1` as a required field in most hash objects. v2 drops SHA1 entirely. SHA1 served no unique purpose in the identity scheme (it is not used for `_id` derivation) and adds computational overhead for each hashed input. MD5 provides the legacy default identity algorithm; SHA256 provides the cryptographically strong alternative; SHA512 provides an optional high-strength digest. SHA1 occupies an awkward middle ground where it is neither the fastest, the strongest, nor the default. See §5.11 for the full drop rationale.

Uppercase convention: All hex strings are uppercase, produced via `hashlib.hexdigest().upper()` in the Python implementation.

> **Historical note:** The original PowerShell implementation also uses uppercase hex throughout (`FileId` and `DirectoryId` both call `.ToUpper()` on their output). The convention is preserved for backward compatibility with existing v1 identity values.

> **Implementation note:** The Python `HashSet` dataclass SHOULD have `sha512` as an `Optional[str]` field defaulting to `None`. The serialization helper MUST omit the `sha512` key entirely from the JSON output when its value is `None`, rather than emitting `"sha512": null`. This matches the JSON Schema's `required` constraint (only `md5` and `sha256` are required) and avoids bloating the output with null optional fields.

5.2.2. NameObject

A `NameObject` pairs a text string with its associated hash digests. Used for file names, directory names, parent directory names, and metadata sidecar file names.

Property	Type	Required	Description
`text`	`string` or `null`	Yes	The text value of the name. Includes the extension for files. Null when the named entity has no meaningful name (e.g., generated metadata entries).
`hashes`	`HashSet` or `null`	Yes	Hash digests of the UTF-8 byte representation of the `text` string. Null when `text` is null.

The `text` and `hashes` fields have a co-nullability invariant: when `text` is `null`, `hashes` MUST also be `null`. When `text` is a non-empty string, `hashes` MUST be a populated `HashSet`. The implementation SHOULD enforce this invariant at construction time.

v1 comparison: v1 uses separate top-level field pairs — `Name` / `NameHashes`, `ParentName` / `ParentNameHashes` — where the relationship between the text and its hashes is implicit. v2's `NameObject` makes the relationship explicit and eliminates the possibility of a name being present without its hashes or vice versa.

Hash input encoding: The hashes in a `NameObject` are computed from the UTF-8 encoded bytes of the `text` string, i.e., `hashlib.md5(name.encode('utf-8'))`.

> **Historical note:** The original uses the same encoding: PowerShell's `[System.Text.Encoding]::UTF8.GetBytes($Name)` produces identical UTF-8 bytes.

5.2.3. SizeObject

A `SizeObject` expresses a file size in both human-readable and machine-readable forms.

Property	Type	Required	Description
`text`	`string`	Yes	Human-readable size string with appropriate unit suffix (e.g., `"15.28 MB"`, `"135 B"`, `"2.04 GB"`). Units follow the decimal SI convention: B, KB, MB, GB, TB.
`bytes`	`integer`	Yes	Exact size in bytes. Minimum value: `0`.

`additionalProperties` is `false`.

v1 comparison: v1 has a single `Size` field of type `number` (bytes only). v2 adds the human-readable string for consumer convenience. The `bytes` field preserves the exact integer value for programmatic use.

Human-readable formatting rules: The `text` string SHOULD use two decimal places for values ≥ 1 KB and no decimal places for values in bytes. The unit thresholds are: < 1,000 B → `"N B"`, < 1,000,000 B → `"N.NN KB"`, < 1,000,000,000 B → `"N.NN MB"`, < 1,000,000,000,000 B → `"N.NN GB"`, otherwise `"N.NN TB"`. These thresholds use decimal SI (powers of 1,000), not binary (powers of 1,024). A 1,048,576-byte file is reported as `"1.05 MB"`, not `"1.00 MiB"`.

> **New in v2.** This type has no v1 equivalent. The original stores only a raw byte count.

5.2.4. TimestampPair

A `TimestampPair` expresses a single timestamp in both ISO 8601 local-time and Unix epoch millisecond formats.

Property	Type	Required	Description
`iso`	`string`	Yes	ISO 8601 timestamp with fractional seconds and timezone offset. Format: `yyyy-MM-ddTHH:mm:ss.fffffffzzz` (up to 7 fractional digits; Python implementations will typically produce 6).
`unix`	`integer`	Yes	Unix timestamp in milliseconds since epoch (`1970-01-01T00:00:00Z`). Integer precision. Timezone-independent.

`additionalProperties` is `false`.

v1 comparison: v1 uses separate top-level field pairs — `TimeAccessed` / `UnixTimeAccessed`, `TimeCreated` / `UnixTimeCreated`, `TimeModified` / `UnixTimeModified`. v2's `TimestampPair` consolidates each pair into a single object and nests them inside a `TimestampsObject` (see §5.2.5).

Fractional seconds precision: Python's `datetime.isoformat()` produces 6 fractional digits by default (microsecond precision). The ISO string format SHOULD include all available fractional digits without artificial truncation or zero-padding to a specific width. See §6.5 for the timestamp derivation logic and §15.5 for cross-platform precision considerations.

> **Historical note:** The original's `.ToString($DateFormat)` uses the `fffffff` format specifier, which produces 7 fractional digits (100-nanosecond precision, the resolution of .NET's `DateTime` type). The 7th digit is almost always `0` in practice because filesystem timestamps rarely carry sub-microsecond precision. The 6-digit Python output is an acceptable deviation.

Millisecond Unix timestamps: The `unix` value is in milliseconds, not seconds, computed as `int(stat_result.st_mtime * 1000)`. See DEV-07 in §2.6.

> **Historical note:** This matches the original's `[DateTimeOffset]::ToUnixTimeMilliseconds()` output.

5.2.5. TimestampsObject

A `TimestampsObject` groups the three standard filesystem timestamps.

Property	Type	Required	Description
`created`	`TimestampPair`	Yes	When the item was created on the filesystem.
`modified`	`TimestampPair`	Yes	When the item's content was last modified.
`accessed`	`TimestampPair`	Yes	When the item was last accessed (read). See platform caveat below.

`additionalProperties` is `false`.

Access time caveat: Filesystem access-time tracking varies by OS and mount options. Linux systems mounted with `noatime` or `relatime` (the default on most distributions) may report stale or approximate access times. The indexer reports whatever the filesystem provides via `os.stat()` without attempting to validate accuracy. Consumers SHOULD NOT rely on `accessed` timestamps for precise behavioral analysis.

Creation time portability: On Windows, `os.stat().st_birthtime` (Python 3.12+) or `st_ctime` provides the file creation time. On Linux, `st_birthtime` is available on Python 3.12+ for filesystems that support it (ext4 with kernel 4.11+); on older kernels or unsupported filesystems, it is unavailable. On macOS, `st_birthtime` is generally available. When creation time is not available, the implementation MUST fall back to `st_ctime` (which on Linux represents the inode change time, not the creation time) and SHOULD log a diagnostic message on the first occurrence per invocation. See §15.5 for the full cross-platform discussion.

5.2.6. ParentObject

A `ParentObject` provides identity and naming information for the parent directory of an indexed item.

Property	Type	Required	Description
`id`	`string`	Yes	The unique identifier of the parent directory. Uses the `x` prefix (directory ID namespace). Pattern: `^x[0-9A-F]+$`.
`name`	`NameObject`	Yes	The name of the parent directory.

`additionalProperties` is `false`.

The parent `id` is computed using the same two-layer directory hashing scheme as directory entries themselves: `hash(hash(parent_name) + hash(grandparent_name))`, prefixed with `x`. For items at the root of the indexed tree, the parent directory exists on the filesystem (it is the target directory's own parent) but will not have its own `IndexEntry` in the output. The parent ID is still computed and populated.

v1 comparison: v1 spreads parent identity across four top-level fields — `ParentId`, `ParentIds`, `ParentName`, `ParentNameHashes`. v2 collapses these into a single `ParentObject` with two properties. The v1 `ParentId` (a single string selected from `ParentIds` by the chosen algorithm) maps to v2's `parent.id`. The v1 `ParentIds` (the full hash set) is not directly carried into `ParentObject` — the parent's `name.hashes` provides the name hashes, and the parent's full identity hash set can be recomputed from those if needed. This is a deliberate simplification: carrying the full `ParentIds` hash set was redundant given that the parent ID's derivation from name hashes is deterministic and documented.

> **Deviation from v1 field cardinality:** v1's `ParentIds` provides the parent's full hash-based ID set (MD5, SHA256, optionally SHA1 and SHA512) as a separate object. v2 provides only the single selected `parent.id` string and the parent's `name` (with its name hashes). If a consumer needs the parent's alternative algorithm ID, they can recompute it from the parent name hashes using the documented directory ID scheme. This reduces per-entry size and eliminates a field whose values were derivable from other present fields.

5.3. Top-Level IndexEntry Fields

An `IndexEntry` is a JSON object conforming to the root schema. It describes a single indexed file or directory. The following table lists all top-level properties in the order they appear in the canonical schema. Detailed behavioral guidance for each field follows in §5.4 through §5.10.

Property	Type	Required	Section
`schema_version`	`integer` (const `2`)	Yes	§5.4
`id`	`string`	Yes	§5.4
`id_algorithm`	`string` (enum)	Yes	§5.4
`type`	`string` (enum)	Yes	§5.4

Property	Type	Required	Section
`name`	`NameObject`	Yes	§5.5
`extension`	`string` or `null`	Yes	§5.5
`mime_type`	`string` or `null`	No	§5.5
`size`	`SizeObject`	Yes	§5.5
`hashes`	`HashSet` or `null`	Yes	§5.5
`file_system`	`object`	Yes	§5.6
`timestamps`	`TimestampsObject`	Yes	§5.7
`attributes`	`object`	Yes	§5.8
`items`	`array` of `IndexEntry` or `null`	No	§5.9
`metadata`	`array` of `MetadataEntry` or `null`	No	§5.10

`additionalProperties` is `false` at the root level — no extra keys are permitted. The `required` array in the canonical schema is:

```
[
  "schema_version", "id", "id_algorithm", "type", "name",
  "extension", "size", "hashes", "file_system", "timestamps", "attributes"
]
```

The `items` and `metadata` fields are not in the `required` array. They MAY be omitted entirely (not just set to `null`) when not applicable. However, the implementation SHOULD include them with explicit `null` values for consistency — an `IndexEntry` for a file emits `"items": null`, `"metadata": null` rather than omitting the keys. This makes every entry structurally uniform, which simplifies consumer parsing. The `mime_type` field is also not required and follows the same convention.

5.4. Identity Fields

These fields establish the item's unique identity and schema context.

`schema_version`

Attribute	Value
Type	`integer`
Constraint	`const: 2`
Required	Yes
v1 equivalent	None (v1 has no version discriminator)

Always the integer `2`. The serializer SHOULD place this field first in the serialized JSON output for readability, though JSON objects are unordered and consumers MUST NOT depend on field order.

`id`

Attribute	Value
Type	`string`
Pattern	`^[xy][0-9A-F]+$`
Required	Yes
v1 equivalent	`_id`

The primary unique identifier for the indexed item. The first character is a type prefix that encodes the item's namespace:

- `y` — File. The hash portion is derived from the file's content bytes (or from the file's name string if the file is a symbolic link).
- `x` — Directory. The hash portion is derived from the two-layer name hashing scheme: `hash(hash(directory_name) + hash(parent_directory_name))`.

The remaining characters are the uppercase hexadecimal hash digest selected by `id_algorithm`. For the default `id_algorithm` of `md5`, a file ID is 33 characters total (1 prefix + 32 hex). For `sha256`, it is 65 characters (1 prefix + 64 hex).

v1 comparison: Identical semantics. The field name changes from `_id` to `id` (the leading underscore was a legacy convention from the original's MongoDB-influenced naming). The `z` prefix used for generated metadata entry IDs appears in `MetadataEntry.id` (§5.10), not at the `IndexEntry` top level.

Derivation details: The hash computation and prefix application logic are defined in §6.3. The `id` value MUST be one of the hash digests present in the `hashes` field (for files) or derivable from the `name.hashes` field (for directories). The `id_algorithm` field identifies which one.

`id_algorithm`

Attribute	Value
Type	`string`
Enum	`["md5", "sha256"]`
Required	Yes
v1 equivalent	None (implicit in v1)

The hash algorithm used to generate the `id` field. The value is always lowercase. The default is `"md5"`. The `"sha256"` option exists for workflows that require cryptographically stronger identifiers.

> **Historical note:** The original PowerShell implementation also used MD5 as the default identity algorithm.

This field determines:

- Which digest from the `hashes` `HashSet` is used as the basis for `id`.
- Which digest is used to construct `attributes.storage_name`.
- Which digest is used for `file_system.parent.id` (the parent directory ID is always computed with the same algorithm as the child's ID).

> **New in v2.** v1 provides no mechanism to determine which algorithm produced the `_id` value. Consumers must reverse-match `_id` against the `Ids` object. v2 makes this explicit.

`type`

Attribute	Value
Type	`string`
Enum	`["file", "directory"]`
Required	Yes
v1 equivalent	`IsDirectory` (boolean)

The fundamental filesystem type of the indexed item. This is the primary structural discriminator — it determines which other fields are populated vs. null, whether `items` is meaningful, and how `hashes` should be interpreted.

`type` value	`hashes`	`extension`	`items`	`metadata`
`"file"`	Content hash `HashSet` (or name hash if symlink)	File extension string	`null`	Array of `MetadataEntry` or `null`
`"directory"`	`null`	`null`	Array of child `IndexEntry` or `null`	`null`

v1 comparison: v1 uses `IsDirectory: true/false`. v2 replaces this with a string enum for three reasons. First, it eliminates the implicit "IsFile" concept (v1 has no explicit `IsFile` field — it is inferred from `IsDirectory: false`). Second, it avoids the boolean ambiguity where `false` carries semantic meaning that must be negated to interpret ("not a directory" → "a file"). Third, string enums are extensible if future schema versions need to add item types (e.g., `"symlink"` as a distinct type rather than a flag).

5.5. Naming and Content Fields

These fields describe the item's name, extension, content type, size, and content hashes.

`name`

Attribute	Value
Type	`NameObject`
Required	Yes
v1 equivalent	`Name` + `NameHashes`

The name of the indexed item. For files, `name.text` includes the extension (e.g., `"report.pdf"`). For directories, `name.text` is the directory name (e.g., `"photos"`). Does not include any path components.

The `name.hashes` field contains the hash digests of the UTF-8 encoded bytes of `name.text`. These name hashes are used in directory identity computation (see §6.3) and are included for both files and directories.

v1 comparison: v1's `Name` (a plain string) and `NameHashes` (a hash object) are consolidated into a single `NameObject`. v1's `BaseName` (the filename without extension) is dropped — see §5.11.

`extension`

Attribute	Value
Type	`string` or `null`
Required	Yes
v1 equivalent	`Extension`

The file extension without the leading period (e.g., `"exe"`, `"json"`, `"tar.gz"`). Null for directories. Also null for files that have no extension.

The extension value is derived from the filesystem name. For multi-part extensions like `.tar.gz`, the implementation SHOULD use the full compound extension. Extension validation is governed by the configurable pattern described in §7 (default: the legacy regex externalized per DEV-14 in §2.6). Extensions that fail validation are still recorded in this field — validation failures affect only whether the extension is considered "recognized" for purposes like exiftool processing, not whether it is stored.

The extension is stored in lowercase.

> **Historical note:** The original's `MakeObject` also converts extensions to lowercase before storage. This convention is preserved.

`mime_type`

Attribute	Value
Type	`string` or `null`
Required	No
v1 equivalent	None

The MIME type of the file as detected by the indexer (e.g., `"application/octet-stream"`, `"text/plain"`, `"image/png"`). Null for directories.

Detection is based on the file extension using Python's `mimetypes.guess_type()` from the standard library. If `exiftool` is available and returns a `MIMEType` field, the exiftool-reported MIME type takes precedence over the extension-based guess when the two disagree. If neither method produces a result, the field is set to `null`.

> **New in v2.** This field has no v1 equivalent. The original does not perform MIME type detection. This is a low-cost addition that provides significant utility for downstream consumers who need to filter or route index entries by content type.

`size`

Attribute	Value
Type	`SizeObject`
Required	Yes
v1 equivalent	`Size` (number, bytes only)

The size of the item. For files, `size.bytes` is the file size as reported by `os.stat().st_size`. For directories, `size.bytes` is the total combined size of all contained files and subdirectories — computed as the sum of all child `size.bytes` values during the traversal loop (see §4.3, step 4j). The `size.text` field provides the human-readable representation per the formatting rules in §5.2.3.

v1 comparison: v1's `Size` is a bare `number` (bytes only). v2 wraps it in a `SizeObject` that adds the human-readable `text` field.

`hashes`

Attribute	Value
Type	`HashSet` or `null`
Required	Yes
v1 equivalent	`ContentHashes` (partially; see below)

Hash digests of the item's content.

For **files**: the `HashSet` contains digests computed over the file's byte content in a single streaming pass (see §6.3 and §17.1). If the file is a symbolic link, the hashes are computed over the UTF-8 encoded bytes of the file's name string instead, ensuring deterministic IDs without requiring the link target to be accessible.

For **directories**: `null`. Directory identity is derived from name hashing (the two-layer scheme), not content hashing. The directory's name hashes are in `name.hashes`.

v1 comparison: v1 has two separate hash objects — `Ids` (which for files contains content-derived values and for directories contains name-derived values) and `ContentHashes` (which for files duplicates `Ids` and for directories is `null`). v2 eliminates this redundancy. The `hashes` field corresponds to content hashes only; name hashes live in `name.hashes`. For files, `hashes` is the single source of content hash digests. For directories, `hashes` is `null` and identity derivation uses `name.hashes`.

> **Deviation from v1 `Ids` semantics:** v1's `Ids` object serves double duty — it is both "the hashes used for identity derivation" and "the content/name hashes." v2 separates these concepts. The `id` field is the identity. The `hashes` field is the content hashes (files only). The `name.hashes` field is the name hashes (both files and directories). The `id` value is derived from one of these hash sets depending on the item type, as documented by `id_algorithm`. This separation is clearer and eliminates the confusion of having an `Ids` object that means different things for files vs. directories.

5.6. Filesystem Location and Hierarchy Fields

The `file_system` top-level property groups filesystem location and hierarchy information.

`file_system`

Attribute	Value
Type	`object`
Required	Yes
v1 equivalent	Partially: `ParentId` + `ParentIds` + `ParentName` + `ParentNameHashes`

Property	Type	Required	Description
`relative`	`string`	Yes	The relative path from the index root to this item. Uses forward-slash separators regardless of the source platform.
`parent`	`ParentObject` or `null`	Yes	Identity and naming information for this item's parent directory.

`additionalProperties` is `false`.

`file_system.relative` is the relative path from the root of the index operation to the current item. For the root item itself (the target of the indexing invocation), this is `"."`. For a file `photos/vacation/beach.jpg` within a directory being indexed recursively, the relative path is `"photos/vacation/beach.jpg"`. Path separators are always forward slashes, even when the indexer runs on Windows. This ensures that index output is portable across platforms.

> **New in v2.** v1 has no relative path field. The original's output embeds items in a recursive tree structure but does not record the relative path for any individual entry. The relative path is new in v2 and provides significant utility for consumers who need to locate or reconstruct the filesystem layout from a flat iteration of the entry tree.

`file_system.parent` is a `ParentObject` (§5.2.6) containing the parent directory's computed ID and name. Null for the root item of a single-file index operation where the parent directory's identity is not meaningful. For all other items — including the root item of a directory index operation — the parent is populated.

v1 comparison: v1's `ParentId`, `ParentIds`, `ParentName`, and `ParentNameHashes` are consolidated into `file_system.parent`. The v1 `ParentId` value of `"x"` (used when the item is at the root of the system) is not preserved — `parent` is `null` in that scenario instead. See §5.2.6 for the `ParentObject` field mapping.

5.7. Timestamp Fields

`timestamps`

Attribute	Value
Type	`TimestampsObject`
Required	Yes
v1 equivalent	`TimeAccessed` + `UnixTimeAccessed` + `TimeCreated` + `UnixTimeCreated` + `TimeModified` + `UnixTimeModified`

The three standard filesystem timestamps, each as a `TimestampPair` (§5.2.4) providing both ISO 8601 and Unix millisecond representations.

v1 comparison: v1 uses six separate top-level fields for three timestamps (two formats each). v2 nests them in a single `TimestampsObject` containing three `TimestampPair` values. The semantic content is identical; the structural organization is consolidated.

Derivation: All timestamps are derived from `os.stat()` / `os.lstat()` results. The ISO string is produced from a `datetime` object constructed from the stat float, with the local timezone offset attached. The Unix millisecond integer is computed directly from the stat float. See §6.5 for implementation details and §15.5 for cross-platform behavior.

5.8. Attribute Fields

`attributes`

Attribute	Value
Type	`object`
Required	Yes
v1 equivalent	`IsLink` + `StorageName`

Property	Type	Required	Description
`is_link`	`boolean`	Yes	Whether the item is a symbolic link (symlink).
`storage_name`	`string`	Yes	The deterministic name for renamed/storage mode.

`additionalProperties` is `false`.

`attributes.is_link` is `true` when the item is a symbolic link, `false` otherwise. When `true`, the item's `hashes` are computed from the file name string rather than the file content (for files) or the hashing falls back to name-only (for directory symlinks), since the link target may not be accessible. See §6.4 for symlink detection logic.

v1 comparison: Direct mapping from v1's `IsLink` boolean. The semantics are identical.

`attributes.storage_name` is the deterministic filename used when the indexer's rename operation is active. For files: the `id` followed by a period and the extension (e.g., `"yA8A8C089A6A8583B24C85F5A4A41F5AC.exe"`). For files without an extension: identical to the `id`. For directories: identical to the `id` (e.g., `"x3B4F479E9F880E438882FC34B67D352C"`).

v1 comparison: Direct mapping from v1's `StorageName`. The construction logic is identical.

5.9. Recursive Items Field

`items`

Attribute	Value
Type	`array` of `IndexEntry` or `null`
Required	No
v1 equivalent	`Items`

Child items contained within a directory. Each element is a complete `IndexEntry` conforming to the same root schema (a recursive `$ref` to the root). Present only when the indexed item is a directory and the indexer is operating in recursive or flat-directory mode.

Scenario	`items` value
Item is a file	`null`
Item is a directory, flat mode	Array of immediate child `IndexEntry` objects
Item is a directory, recursive mode	Array of child `IndexEntry` objects, where child directories themselves have populated `items` (recursive nesting)
Item is a directory, single-file mode (not applicable)	N/A — a directory in single-file mode is not a valid scenario

The children in the `items` array SHOULD be ordered files-first, then directories. Within each group (files, directories), entries SHOULD be ordered by name (lexicographic, case-insensitive). This ordering is a convention for human readability, not a schema constraint — consumers MUST NOT depend on any particular ordering of the `items` array.

> **Historical note:** The original also used files-first, directories-second traversal order.

v1 comparison: v1's `Items` has the same recursive structure. The v1 schema defines `Items` with `anyOf` permitting `null` elements within the array — the v2 schema tightens this to require that every element in the `items` array is a valid `IndexEntry` (no null entries). If an item cannot be processed (permission error, etc.), it is excluded from the array entirely rather than represented as a null placeholder. This is a stricter contract that simplifies consumer code.

5.10. Metadata Array and MetadataEntry Fields

`metadata`

Attribute	Value
Type	`array` of `MetadataEntry` or `null`
Required	No
v1 equivalent	`Metadata`

An array of metadata records associated with the indexed item. For files, this typically includes an exiftool-generated entry (when the `-Meta` flag is active and the file type is not in the exclusion list) and any sidecar metadata files discovered alongside the item (when MetaMerge is active). For directories, this is `null`. Each element is a self-contained `MetadataEntry` object.

The `metadata` array MAY be empty (an empty array `[]`) when metadata processing is active but no metadata sources are found for the item. The distinction between `null` (metadata processing not applicable or not requested) and `[]` (metadata processing was performed but yielded no results) is semantically meaningful and SHOULD be preserved by the implementation.

MetadataEntry

A `MetadataEntry` is a self-contained record describing a single metadata source associated with the parent `IndexEntry`. The v2 `MetadataEntry` is significantly richer than its v1 counterpart, carrying enough information to support MetaMergeDelete reversal operations.

Top-level properties of `MetadataEntry`:

Property	Type	Required	Description
`id`	`string`	Yes	Unique identifier. Prefix `z` for generated, `y` for sidecar. Pattern: `^[yz][0-9A-F]+$`.
`origin`	`string` (enum)	Yes	`"generated"` or `"sidecar"`. Primary discriminator.
`name`	`NameObject`	Yes	Source name. For sidecar: original filename. For generated: `text` and `hashes` are both `null`.
`hashes`	`HashSet`	Yes	Content hashes. For sidecar: hashes of the original file bytes. For generated: hashes of the serialized output.
`file_system`	`object` or absent	No	Sidecar only. Contains `relative` (relative path to original sidecar file).
`size`	`SizeObject` or absent	No	Sidecar only. Size of the original sidecar file.
`timestamps`	`TimestampsObject` or absent	No	Sidecar only. Timestamps of the original sidecar file.
`attributes`	`object`	Yes	Classification, format, and transform info. See below.
`data`	`null`, `string`, `object`, or `array`	Yes	The metadata content.

The `required` array for `MetadataEntry` is: `["id", "origin", "name", "hashes", "attributes", "data"]`.

MetadataEntry.origin

The `origin` field is the primary structural discriminator for a `MetadataEntry`. It determines which optional fields are present and how the entry should be interpreted.

`origin`	ID prefix	`file_system`	`size`	`timestamps`	Description
`"generated"`	`z`	Absent	Absent	Absent	Created by a tool during indexing (e.g., exiftool output). Never existed as a standalone file.
`"sidecar"`	`y`	Present	Present	Present	Absorbed from an external metadata file discovered alongside the indexed item. Carries full filesystem provenance for MetaMergeDelete reversal.

v1 comparison: v1's `MetadataEntry` has `Source` and `Type` fields that partially encode provenance, but does not distinguish generated from sidecar metadata structurally. v1 has no filesystem provenance fields for sidecar entries — the original sidecar file's path, size, and timestamps are lost after MetaMerge. v2's explicit `origin` discriminator and the sidecar-only provenance fields are the structural foundation for reversible MetaMergeDelete operations.

MetadataEntry.attributes

The `attributes` sub-object classifies the metadata entry's content type, serialization format, and any transformations applied to the source data before storage.

Property	Type	Required	Description
`type`	`string`	Yes	Semantic classification. See type values below.
`format`	`string` (enum)	Yes	Serialization format of the `data` field: `"json"`, `"text"`, `"base64"`, or `"lines"`.
`source_media_type`	`string`	No	MIME type of the original source data when the stored format differs from the original (e.g., `"image/png"` for a Base64-encoded screenshot).
`transforms`	`array` of `string`	Yes	Ordered list of transformations applied. Empty array means no transforms.

The `required` array for `attributes` is: `["type", "format", "transforms"]`.

`attributes.type` values use hierarchical dot-notation where a prefix identifies the generating tool for generated metadata:

For generated metadata (prefix identifies the tool):

- `"exiftool.json_metadata"` — EXIF/XMP/IPTC metadata extracted by exiftool, delivered as a JSON object.

For sidecar metadata (no prefix — `origin` already indicates sidecar provenance):

- `"description"` — Text description file (e.g., youtube-dl output).
- `"desktop_ini"` — Windows `desktop.ini` file.
- `"generic_metadata"` — Generic metadata/config file (`.cfg`, `.conf`, `.yaml`, `.meta`, etc.).
- `"hash"` — File containing hash/checksum values (`.md5`, `.sha256`, `.crc32`, etc.).
- `"json_metadata"` — JSON-format metadata file (`.info.json`, `.meta.json`, etc.).
- `"link"` — URL shortcut (`.url`), filesystem shortcut (`.lnk`), or pointer file.
- `"screenshot"` — Screen capture image associated with the indexed item.
- `"subtitles"` — Subtitle/caption track (`.srt`, `.sub`, `.vtt`, `.lrc`, etc.).
- `"thumbnail"` — Thumbnail/cover image (`.cover`, `.thumb`, `thumbs.db`).
- `"torrent"` — Torrent/magnet link file.
- `"error"` — The entry could not be read or classified. `data` may be `null` or partial.

These type values are derived from the `MetadataFileParser.Identify` key names in the original configuration, lowercased and converted from PascalCase to snake_case (e.g., `JsonMetadata` → `json_metadata`, `DesktopIni` → `desktop_ini`). The generated metadata type uses dot-notation (`exiftool.json_metadata`) to namespace it separately from the sidecar types.

v1 comparison: v1's `Source` and `Type` fields map roughly to `origin` and `attributes.type` respectively. v1's `Source` carries free-text values like `"internal"` or `"external"`; v2's `origin` is a strict two-value enum. v1's `Type` carries the PascalCase type name; v2's `attributes.type` uses lowercase snake_case with tool-prefixed dot-notation for generated entries.

`attributes.format` values:

Format	`data` type	Description
`"json"`	`object` or `array`	Parsed JSON. Stored as a native JSON structure, not a string.
`"text"`	`string`	UTF-8 text content.
`"base64"`	`string`	Base64-encoded binary content. Decode to recover original bytes.
`"lines"`	`array` of `string`	Line-oriented content (hash files, subtitle files).

`attributes.transforms` is an ordered list of transformations applied to the source data before storing it in `data`. The list is ordered from first-applied to last-applied. To reverse storage and recover the original data, apply the inverse of each transform in reverse order.

Defined transform identifiers:

- `"base64_encode"` — Source bytes were Base64-encoded for JSON-safe storage. Inverse: Base64-decode.
- `"json_compact"` — Source JSON was compacted (whitespace removed). Inverse: none needed.
- `"line_split"` — Source text was split into an array of lines (empty lines filtered). Inverse: join with newline.
- `"key_filter"` — Specific keys were removed from a JSON object (e.g., exiftool system keys). Inverse: not reversible.

An empty array means the data is stored as-is with no transformations.

MetadataEntry.data

The `data` field contains the actual metadata content. Its structure depends on `attributes.format`:

- When `format` is `"json"`: a JSON object or array.
- When `format` is `"text"` or `"base64"`: a string.
- When `format` is `"lines"`: an array of strings.
- May be `null` if the metadata could not be read (when `attributes.type` is `"error"`).

v1 comparison: v1's `Data` field has the same polymorphic nature (`"type": ["null", "string", "object", "array"]`). The difference is that v2's `attributes.format` explicitly declares how to interpret the `data` value, whereas v1 consumers must infer the format from context.

5.11. Dropped and Restructured Fields

This section documents every v1 field that is absent from v2 and every v1 field whose v2 representation differs structurally. This serves as a complete mapping reference for the eventual v1-to-v2 migration utility and for consumers adapting existing v1 parsing code.

Dropped fields

`Encoding` — Dropped (DEV-12 in §2.6). The v1 `Encoding` field contains a serialization of the .NET `System.Text.Encoding` object produced by the `GetFileEncoding` sub-function. This includes properties like `IsSingleByte`, `Preamble`, `BodyName`, `EncodingName`, `HeaderName`, `WebName`, `WindowsCodePage`, `IsBrowserDisplay`, `IsBrowserSave`, `IsMailNewsDisplay`, `IsMailNewsSave`, `EncoderFallback`, `DecoderFallback`, `IsReadOnly`, and `CodePage`. The field is deeply coupled to the .NET type system and has limited utility outside .NET consumers. Python has no standard library facility that produces the same output structure. BOM detection can be performed via `chardet` or manual byte inspection, but the full .NET encoding profile is not reproducible. The field is dropped without replacement. If encoding detection becomes a requirement in a future version, a new field with a Python-native structure would be added.

`BaseName` — Dropped. The v1 `BaseName` field contains the filename without its extension (the "stem"). This value is trivially derivable from `name.text` by stripping the `extension` — e.g., for `name.text = "report.pdf"` and `extension = "pdf"`, the base name is `"report"`. Storing a derivable value as a separate field adds no information and inflates the output. Consumers who need the base name can compute it: `name.text.rsplit('.', 1)[0]` (or simply `name.text` when `extension` is `null`).

> **Improvement over v1:** The original includes `BaseName` as a top-level required field because PowerShell's `Get-Item` object exposes `.BaseName` as a property and it was inexpensive to include. In a schema designed for clarity and minimalism, derivable fields are omitted.

`SHA1` (within hash objects) — Dropped. All v1 hash objects (`Ids`, `NameHashes`, `ContentHashes`, `ParentIds`, `ParentNameHashes`) include a `SHA1` property. v2's `HashSet` drops SHA1 entirely. SHA1 is not used for identity derivation (only MD5 and SHA256 are `id_algorithm` options), it provides no unique value that MD5 or SHA256 does not already provide in the indexer's use case, and computing it adds overhead for every hashed input. SHA1's cryptographic weaknesses (demonstrated collision attacks since 2017) make it unsuitable as a security-relevant digest, and its 160-bit length occupies an awkward middle ground between MD5 (128-bit, fast, legacy default) and SHA256 (256-bit, strong, recommended). `shruggie-indexer` does not compute SHA1 digests.

Restructured fields (v1 → v2 mapping)

v1 Field	v2 Location	Structural Change
`_id`	`id`	Renamed. Leading underscore removed.
`Ids`	`hashes` (files), `name.hashes` (directories)	Split. For files, `Ids` content → `hashes`. For directories, `Ids` content was derived from name hashes → `name.hashes`. SHA1 dropped.
`Name`	`name.text`	Nested into `NameObject`.
`NameHashes`	`name.hashes`	Nested into `NameObject`. SHA1 dropped.
`ContentHashes`	`hashes`	Renamed and promoted. Null for directories. SHA1 dropped.
`Extension`	`extension`	Renamed (lowercase).
`BaseName`	Dropped	Derivable from `name.text` and `extension`.
`StorageName`	`attributes.storage_name`	Nested into `attributes` object.
`Encoding`	Dropped	.NET-specific. No replacement.
`Size`	`size.bytes`	Nested into `SizeObject`. `size.text` added.
`IsDirectory`	`type`	Replaced by string enum.
`IsLink`	`attributes.is_link`	Nested into `attributes` object.
`ParentId`	`file_system.parent.id`	Nested into `file_system.parent`.
`ParentIds`	Dropped	Derivable from `file_system.parent.name.hashes`.
`ParentName`	`file_system.parent.name.text`	Nested into `file_system.parent.name`.
`ParentNameHashes`	`file_system.parent.name.hashes`	Nested into `file_system.parent.name`. SHA1 dropped.
`UnixTimeAccessed`	`timestamps.accessed.unix`	Nested into `TimestampsObject` → `TimestampPair`.

v1 Field	v2 Location	Structural Change
`TimeAccessed`	`timestamps.accessed.iso`	Nested into `TimestampsObject` → `TimestampPair`.
`UnixTimeCreated`	`timestamps.created.unix`	Nested.
`TimeCreated`	`timestamps.created.iso`	Nested.
`UnixTimeModified`	`timestamps.modified.unix`	Nested.
`TimeModified`	`timestamps.modified.iso`	Nested.
`Items`	`items`	Renamed (lowercase). Null entries disallowed.
`Metadata`	`metadata`	Renamed (lowercase). `MetadataEntry` structure significantly enriched.
(Metadata) `Source`	`metadata[].origin`	Replaced by strict enum.
(Metadata) `Type`	`metadata[].attributes.type`	Nested. Snake_case. Dot-notation for generated entries.
(Metadata) `Name`	`metadata[].name.text`	Nested into `NameObject`.
(Metadata) `NameHashes`	`metadata[].name.hashes`	Nested into `NameObject`.
(Metadata) `Data`	`metadata[].data`	Same polymorphic type. Format now explicit via `attributes.format`.

New v2 fields with no v1 equivalent

v2 Field	Description
`schema_version`	Version discriminator (always `2`).
`id_algorithm`	Explicit algorithm identifier for `id` derivation.
`type`	String enum replacing `IsDirectory` boolean.
`mime_type`	MIME type detection (extension and/or exiftool).
`size.text`	Human-readable size string.
`file_system.relative`	Relative path from index root.
`metadata[].origin`	Generated vs. sidecar discriminator.
`metadata[].file_system`	Sidecar file relative path (for MetaMergeDelete reversal).
`metadata[].size`	Sidecar file size (for MetaMergeDelete reversal).
`metadata[].timestamps`	Sidecar file timestamps (for MetaMergeDelete reversal).
`metadata[].hashes`	Content hashes of metadata (integrity verification).
`metadata[].attributes.format`	Explicit data format declaration.
`metadata[].attributes.transforms`	Applied transformation chain (for data reversal).
`metadata[].attributes.source_media_type`	Original MIME type of binary sidecar data.

5.12. Schema Validation and Enforcement

Build-time validation

The canonical v2 JSON Schema (`shruggie-indexer-v2.schema.json`) MUST be used as the validation target for output conformance testing (see §14.4). The test suite SHOULD include a schema conformance test that:

1. Generates index entries for a representative set of inputs (files of various types, directories, symlinks, items with sidecar metadata, items without metadata).
2. Validates each generated entry against the canonical JSON Schema using a Draft-07-compliant validator (e.g., `jsonschema` Python package).
3. Fails the test suite if any entry violates the schema.

This ensures that the implementation's serialization logic stays in sync with the schema definition.

Runtime validation

The core indexing engine (`core/entry.py`) does NOT perform JSON Schema validation at runtime — this would add unacceptable overhead for large directory trees. Instead, the implementation relies on **structural correctness by construction**: the `IndexEntry`, `HashSet`, `NameObject`, `SizeObject`, `TimestampPair`, `TimestampsObject`, `ParentObject`, and `MetadataEntry` dataclasses enforce type constraints and required fields through their

constructors. If a field is required by the schema, the corresponding dataclass field has no default value (forcing the caller to provide it). If a field is nullable, the corresponding dataclass field is typed `Optional[T]` with a default of `None`.

For consumers who want runtime validation (e.g., when ingesting index output from untrusted sources), optional Pydantic models are available behind an import guard in `models/schema.py`. These models mirror the dataclass definitions but add Pydantic's runtime type checking, pattern validation, and `model_validate_json()` for schema-validating a JSON string on ingestion. The Pydantic models are not used by the core engine. See §3.2 for the module layout.

Serialization invariants

The serializer (`core/serializer.py`) MUST enforce the following invariants when converting an `IndexEntry` to JSON:

1. **No additional properties.** Every JSON object in the output corresponds to a schema type with `additionalProperties: false`. The serializer MUST NOT include keys that are not defined in the schema.
2. **Required fields are always present.** Every field in the schema's `required` array MUST appear in the output, even when its value is `null`.
3. **Optional fields with `null` values.** For top-level `IndexEntry` properties that are not in `required` (`items`, `metadata`, `mime_type`), the serializer SHOULD include them with `null` values for structural uniformity. Consumers SHOULD be prepared for these fields to be absent or `null`.
4. **Optional `HashSet` fields.** `HashSet.sha512` MUST be omitted from the JSON output (not emitted as `null`) when it was not computed. This matches the JSON Schema expectation that `sha512` is simply not present rather than present-but-null.
5. **Sidecar-only MetadataEntry fields.** `MetadataEntry.file_system`, `MetadataEntry.size`, and `MetadataEntry.timestamps` MUST be present for sidecar entries (`origin: "sidecar"`) and MUST be absent (not `null`) for generated entries (`origin: "generated"`). This is a structural invariant enforced by the `origin` discriminator.

5.13. Backward Compatibility Considerations

The v2 schema is a breaking change from v1. Consumers of existing v1 index assets cannot parse v2 output without modification, and vice versa.

Migration path

The planned v1-to-v2 migration utility (post-MVP, see §1.2) will convert existing `_meta.json` and `_directorymeta.json` v1 files to the v2 format. The migration is lossy in one direction: v1 fields that are dropped in v2 (`Encoding`, `BaseName`, `SHA1` hashes) are discarded. The migration is enriching in the other direction: v2 fields that have no v1 equivalent (`schema_version`, `id_algorithm`, `type`, `mime_type`, `size.text`, `file_system.relative`, and all `MetadataEntry` provenance fields) are populated with computed or default values where possible and `null` where not.

Filename convention

v1 index sidecar files use the suffixes `_meta.json` (for files) and `_directorymeta.json` (for directories). v2 index sidecar files use the suffixes `_meta2.json` and `_directorymeta2.json`. The `2` in the v2 suffixes prevents collision with existing v1 files and allows both versions to coexist on disk during a migration period. This convention is enforced by the serializer when writing in-place output files.

Consumer guidance

Consumers adapting from v1 to v2 parsing should:

1. Check for the presence of `schema_version`. If present and equal to `2`, parse as v2. If absent, parse as v1.
2. Replace all PascalCase field accessors with snake_case equivalents (e.g., `entry["IsDirectory"]` → `entry["type"] == "directory"`).
3. Navigate nested sub-objects for fields that were previously top-level (e.g., `entry["Size"]` → `entry["size"]["bytes"]`).
4. Handle the absence of `Encoding`, `BaseName`, and `SHA1` fields.
5. For `MetadataEntry` processing, switch from `Source`/`Type` string matching to `origin` enum checking and `attributes.type` matching.

6. Core Operations

This section defines the behavioral contract for every operation in the indexing engine — the inputs each operation accepts, the outputs it produces, the invariants it maintains, and the error conditions it handles. Each subsection corresponds to one operation category from the Operations Catalog (§1.5) and to one or more `core/` modules from the source package layout (§3.2). Together, these subsections constitute the complete specification of what the `core/` subpackage does and how it does it.

The operations are presented in dependency order: foundational operations (traversal, path manipulation, hashing) before the operations that depend on them (entry construction, serialization). An implementer working through these subsections top-to-bottom will build the leaf modules first and the orchestrator last, with each module's dependencies already specified by the time it is reached.

§4 (Architecture) defines the structural relationships between modules — who calls whom. This section defines the behavioral detail within each module — what each function does when called. §5 (Output Schema) defines the data structures that these operations produce. The three sections are complementary: §4 provides the wiring diagram, §6 provides the logic, and §5 provides the output contract.

6.1. Filesystem Traversal and Discovery

Module: `core/traversal.py` **Operations Catalog:** Category 1 **Original functions:** `MakeDirectoryIndexLogic`, `MakeDirectoryIndexRecursiveLogic`, `MakeFileIndex`

Purpose

Enumerates the set of filesystem items (files and subdirectories) to be indexed within a target path. The traversal module is Stage 3 of the processing pipeline (§4.1) — it sits between target resolution (Stage 2) and entry construction (Stage 4). It does not build index entries; it produces an ordered sequence of `Path` objects that the entry builder will process.

Public interface

The traversal module exposes one primary function:

```
def list_children(
    directory: Path,
    config: IndexerConfig,
) -> tuple[list[Path], list[Path]]:
    """Enumerate immediate children of a directory.

    Returns (files, directories) as two separate sorted lists.
    Items matching the configured exclusion filters are omitted.
    """
```

The caller — `core/entry.build_directory_entry()` — invokes `list_children()` once per directory being indexed. For recursive mode, the caller recurses into each returned subdirectory; for flat mode, the caller processes only the immediate children. The traversal module itself does not recurse — recursion is controlled by the entry builder (§6.8), consistent with the separation of traversal from construction described in §4.1.

> **Historical note (DEV-03):** The original has two near-identical traversal code paths: `MakeDirectoryIndexRecursiveLogic` (which calls itself for child directories) and `MakeDirectoryIndexLogic` (which does not recurse). Both paths enumerate children, separate files from directories, filter exclusions, and feed items to `MakeObject` — with almost completely duplicated logic. The single `list_children()` function replaces both, with the recursive/flat distinction handled by the caller rather than the traversal module.

Enumeration strategy

`list_children()` enumerates directory contents using `os.scandir()` in a single pass. Each `DirEntry` object returned by `os.scandir()` is classified as a file or directory using `DirEntry.is_file(follow_symlinks=False)` and `DirEntry.is_dir(follow_symlinks=False)`. The `follow_symlinks=False` argument ensures that symlinks are classified based on the link itself, not the link target — a symlink to a directory appears in the files list (or the directories list if it is a directory symlink), and its symlink status is resolved later during entry construction (§6.4).

`os.scandir()` is preferred over `Path.iterdir()` because `DirEntry` objects cache the results of `is_file()` and `is_dir()` from the underlying `readdir` call on platforms that support it, avoiding redundant `stat()` calls. For large directories (tens of thousands of entries), this caching produces a measurable performance improvement.

> **Historical note:** The original performs two separate `Get-ChildItem` calls — one with `-File` and one with `-Directory` — to separate files from directories. This iterates the directory twice. `os.scandir()` classifies entries in a single pass.

Ordering

The returned file list and directory list are each sorted lexicographically by name (case-insensitive). Files are processed before directories by convention — the caller iterates the file list first, then the directory list. This produces output where file entries precede subdirectory entries within any `items` array.

> **Historical note:** The original also uses files-first, directories-second traversal order.

The sort is performed via `sorted(entries, key=lambda e: e.name.lower())`. The case-insensitive comparison ensures consistent ordering across platforms (Windows is case-insensitive by default; Linux is case-sensitive).

Filesystem exclusion filters

Before returning, `list_children()` removes entries whose names match the configured exclusion set. The exclusion set is defined in `config.filesystem_excludes` — a set of case-insensitive name patterns. The default exclusion set covers cross-platform system artifacts:

Platform	Default exclusions
Windows	`$RECYCLE.BIN`, `System Volume Information`, `desktop.ini`, `Thumbs.db`
macOS	`.DS_Store`, `.Spotlight-V100`, `.Trashes`, `.fseventsd`, `.TemporaryItems`, `.DocumentRevisions-V100`
Linux	`.Trash-*` (glob pattern)
All	`.git` (optional, configurable)

The default set includes all platform-specific entries regardless of the current platform. Filtering a macOS entry on Windows is a no-op (the entry will not exist), but including it in the default list ensures that indexes produced from cross-platform network shares or external drives are clean. See §7 for the configuration schema and override mechanism.

> **Historical note (DEV-10):** The original hardcodes only `$RECYCLE.BIN` and `System Volume Information` as inline `Where-Object` filters. The exclusion list is externalized into configuration and expanded to cover all three target platforms.

Exclusion matching is performed by checking `entry.name.lower()` against the set of lowercased exclusion names. For glob-pattern exclusions (e.g., `.Trash-*`), `fnmatch.fnmatch()` is used. Simple string exclusions use direct set membership for O(1) matching. The exclusion check runs once per entry and is a negligible fraction of the traversal cost.

Error handling

If `os.scandir()` raises `PermissionError` or `OSError` for the directory itself, the error is a **fatal** condition for that directory — the directory cannot be enumerated. The error propagates to the caller (`build_directory_entry`), which handles it according to the item-level error tier (§4.5): the directory is either skipped or included with an empty `items` list and a warning logged.

If an individual `DirEntry` raises an exception during `.is_file()` or `.is_dir()` (rare but possible on network filesystems or corrupted directories), that single entry is skipped with a warning. The remaining entries are still returned.

Single-file scenario

When the target is a single file (§4.6), the traversal module is not called. The entry builder processes the file directly without enumeration. The `list_children()` function is only invoked for directory targets.

6.2. Path Resolution and Manipulation

Module: `core/paths.py` **Operations Catalog:** Category 2 **Original functions:** `ResolvePath`, `FileId-ResolvePath`, `DirectoryId-ResolvePath`, `GetParentPath`, path manipulation in `MakeObject`

Purpose

Provides all path-related operations used by the rest of the indexing engine: resolving paths to canonical absolute form, extracting path components (name, stem, suffix, parent), validating file extensions, and constructing derived paths for output files. This is the single source of truth for path handling — no other module performs its own path manipulation.

Public interface

```
def resolve_path(path: Path) -> Path:
    """Resolve a path to its canonical absolute form.

    Resolves symlinks, normalizes separators, and collapses
    '..' and '.' components. Raises IndexerError if the path
    does not exist and cannot be resolved.
    """

def extract_components(path: Path) -> PathComponents:
    """Extract all path components needed by the entry builder.

    Returns a PathComponents object containing:
      name: str         - Full filename including extension (Path.name)
      stem: str         - Filename without extension (Path.stem)
      suffix: str|None  - Extension without leading dot, or None
      parent_name: str  - Name of the parent directory (Path.parent.name)
      parent_path: Path - Absolute path of the parent directory
    """

def validate_extension(suffix: str, config: IndexerConfig) -> str | None:
    """Validate a file extension against the configured regex pattern.

    Returns the validated extension string (lowercase, no leading dot)
    if valid; returns None if the extension is empty or fails validation.
    """

def build_sidecar_path(item_path: Path, item_type: str) -> Path:
    """Construct the path for an in-place sidecar output file.

    For files: <item_path>_meta2.json
    For directories: <item_path>/_directorymeta2.json
    """
```

```
def build_storage_path(
    item_path: Path, storage_name: str
) -> Path:
    """Construct the target path for a rename operation.

    Returns item_path.parent / storage_name.
    """
```

> **Historical note (DEV-04):** The original contains three independent copies of path-resolution logic: `ResolvePath` inside `MakeIndex`, `FileId-ResolvePath` inside `FileId`, and `DirectoryId-ResolvePath` inside `DirectoryId`. All three do the same thing — call `Resolve-Path` with a `GetFullPath()` fallback for non-existent paths. A single `resolve_path()` function replaces all three. The original also uses `GetParentPath` (a manual `Split-Path` wrapper) and direct `[System.IO.Path]` calls for component extraction; `extract_components()` consolidates all of these using `pathlib` properties.

Path resolution behavior

`resolve_path()` calls `Path.resolve(strict=True)` for paths that exist on the filesystem. This resolves symlinks, normalizes directory separators, and produces an absolute path. If the path does not exist, `resolve_path()` falls back to `Path.resolve(strict=False)`, which normalizes the path without verifying existence. If neither resolution produces a usable path, an `IndexerError` is raised.

The `strict=True` → `strict=False` fallback is functionally equivalent to the original's `Resolve-Path` → `GetFullPath()` fallback pattern, but `pathlib` handles the dispatch cleanly without requiring a try/except around the initial resolution.

Component extraction

`extract_components()` derives all components from `pathlib.Path` properties rather than string manipulation:

Component	`pathlib` property	Notes
`name`	`path.name`	Full filename including extension.
`stem`	`path.stem`	Filename without the final extension.
`suffix`	`path.suffix`	The final extension, including the leading dot. Converted to lowercase and stripped of the leading dot before returning. Empty string → `None`.
`parent_name`	`path.parent.name`	The leaf name of the parent directory. For root-level items (e.g., `C:\file.txt`), this is an empty string — the caller handles this as the "no parent" case.
`parent_path`	`path.parent`	The absolute path to the parent directory.

The suffix is always lowercased for consistency (`path.suffix.lower()`).

> **Historical note:** The original also lowercases extensions via `.ToLower()` in `MakeObject`.

Extension validation

`validate_extension()` matches the extracted suffix against a configurable regex pattern. The default pattern reproduces the intent of the legacy hardcoded regex:

```
Original: ^(([a-z0-9]){1,2}|([a-z0-9]){1}([a-z0-9\-]){1,12}([a-z0-9]){1})$
```

This pattern accepts extensions that are 1–2 alphanumeric characters, or 3–14 characters where the first and last are alphanumeric and interior characters may include hyphens. The purpose is to reject malformed or suspiciously long extensions that might indicate corrupted filenames or path components misidentified as extensions.

The pattern is applied via `re.fullmatch()` with the regex compiled once from `config.extension_validation_pattern`. This is the same regex content as the legacy default but applied via Python's `re` module rather than PowerShell's `-match` operator.

> **Historical note (DEV-14):** The extension validation regex is externalized into the configuration system rather than hardcoded. Users who encounter legitimate long extensions (e.g., `.numbers`, `.download`, `.crdownload`) can adjust the pattern without editing source code.

When validation fails, the extension is treated as absent — the entry's `extension` field is set to `null` and the `storage_name` is constructed from the `id` alone (no extension appended). A debug-level log message is emitted noting the rejected extension.

Path construction

`build_sidecar_path()` constructs the in-place output path using the v2 naming convention:

Item type	Sidecar path pattern	Example
File	`{parent_dir}/{filename}_meta2.json`	`photos/sunset.jpg` → `photos/sunset.jpg_meta2.json`
Directory	`{directory}/_directorymeta2.json`	`photos/vacation/` → `photos/vacation/_directorymeta2.json`

The `2` suffix in `_meta2.json` and `_directorymeta2.json` prevents collision with existing v1 sidecar files (`_meta.json`, `_directorymeta.json`) during a migration period. See §5.13.

`build_storage_path()` constructs the rename-target path by joining the item's parent directory with its `storage_name`. No separator management is needed — `pathlib`'s `/` operator handles platform-correct path construction.

> **Historical note:** The original constructs renamed paths by concatenating strings with the `$Sep` global variable. `pathlib` path arithmetic eliminates manual separator handling entirely.

6.3. Hashing and Identity Generation

Module: `core/hashing.py` **Operations Catalog:** Category 3 **Original functions:** `FileId` (and 8 sub-functions), `DirectoryId` (and 7 sub-functions), `ReadMetaFile-GetNameHashMD5`, `ReadMetaFile-GetNameHashSHA256`, `MetaFileRead-Sha256-File`, `MetaFileRead-Sha256-String`

Purpose

Computes cryptographic hash digests of file contents and name strings, and from those digests produces the deterministic unique identifiers (`id` field) that are the foundation of the indexing system. This is the most dependency-consolidated module in the tool — it replaces four separate locations in the original where hashing logic was independently implemented.

> **Historical note (DEV-01):** The original has no fewer than four independent implementations of the same hashing logic: `FileId` (8 sub-functions for file content and name hashing), `DirectoryId` (4 sub-functions for directory name hashing), `ReadMetaFile-GetNameHashMD5` / `-SHA256` (sidecar file name hashing inside MakeIndex), and `MetaFileRead-Sha256-File` / `-Sha256-String` (content and name hashing inside MetaFileRead). A single hashing module with reusable functions replaces all four.

Public interface

```
def hash_file(path: Path, algorithms: tuple[str, ...] = ("md5", "sha256")) -> HashSet:
    """Compute content hashes of a file.

    Reads the file in chunks and feeds each chunk to all requested
    hash algorithms simultaneously. Returns a HashSet with the
    computed digests.
    """

def hash_string(value: str, algorithms: tuple[str, ...] = ("md5", "sha256")) -> HashSet:
    """Compute hashes of a NFC-normalized, UTF-8 encoded string.

    Applies unicodedata.normalize('NFC', value) before encoding to
    UTF-8 bytes (DEV-15). Computes the requested digests. Returns a
    HashSet.
    """

def hash_directory_id(
    name: str,
    parent_name: str,
    algorithms: tuple[str, ...] = ("md5", "sha256"),
) -> HashSet:
    """Compute directory identity using the two-layer hashing scheme.

    Algorithm:
      1. hash(name)       → name_digest
      2. hash(parent_name) → parent_digest
      3. hash(name_digest + parent_digest) → final_digest

    Returns a HashSet containing the final digests.
    """

def select_id(
    hashes: HashSet,
    algorithm: str,
    prefix: str,
) -> str:
    """Select and prefix an identity value from a HashSet.
```

```
    Returns prefix + hashes[algorithm], e.g., "yA8A8C089...".
    """

# Module-level constants (computed once at import time)
NULL_HASHES: HashSet  # Hash of empty string b'' for each algorithm
```

Algorithms

The tool computes MD5 and SHA256 by default for all hash operations. SHA512 is computed when explicitly enabled in the configuration (`config.compute_sha512 = True`). SHA1 is not computed.

> **Historical note:** The original's `FileId` and `DirectoryId` accept an `IncludeHashTypes` parameter defaulting to `@('md5', 'sha256')`, and the output schema defines fields for SHA1 and SHA512 that remain empty at runtime. SHA1 is dropped entirely (see §5.2.1 for the rationale — SHA1 serves no unique purpose and adds overhead). SHA512 is available as an opt-in for consumers who want high-strength digests. MD5 and SHA256 are always computed because they serve the identity system: MD5 is the legacy default `id_algorithm`, and SHA256 is the recommended alternative.

Multi-algorithm single-pass hashing

`hash_file()` reads the file in fixed-size chunks and feeds each chunk to all active hash objects simultaneously. This is the core performance optimization described in §17.1:

```
# Illustrative — not the exact implementation.
def hash_file(path: Path, algorithms: tuple[str, ...] = ("md5", "sha256")) -> HashSet:
    hashers = {alg: hashlib.new(alg) for alg in algorithms}
    with open(path, "rb") as f:
        while chunk := f.read(CHUNK_SIZE):
            for h in hashers.values():
                h.update(chunk)
    return HashSet(
        md5=hashers["md5"].hexdigest().upper(),
        sha256=hashers["sha256"].hexdigest().upper(),
        sha512=hashers.get("sha512", _Absent).hexdigest().upper() if "sha512" in hashers else None,
    )
```

The chunk size (`CHUNK_SIZE`) defaults to 65,536 bytes (64 KB). This value balances memory usage against read-call overhead — Python's `hashlib` documentation recommends chunk sizes between 4 KB and 128 KB for stream hashing. The chunk size is not configurable; it is an implementation detail of the hashing module.

> **Historical note (DEV-02):** The original computes each hash algorithm in a separate pass, opening and reading the file once per algorithm. For a file hashed with two algorithms, this means two complete reads. The tool reads the file once and updates all hash objects from each chunk, halving the I/O for the default two-algorithm case.

String hashing

`hash_string()` first applies `unicodedata.normalize('NFC', value)` to the input string, then encodes the NFC-normalized result to UTF-8 bytes via `.encode("utf-8")`, and computes the requested digests in a single pass. The NFC normalization step (DEV-15) ensures that the same logical string produces identical hash digests regardless of whether the source filesystem returned the name in NFC (Windows, most Linux) or NFD (macOS HFS+). This is an intentional deviation from the original's `[System.Text.Encoding]::UTF8.GetBytes()` conversion, which hashed the raw bytes without normalization — an approach that was safe only because the original ran exclusively on Windows.

For `None` or empty-string inputs, `hash_string()` returns the `NULL_HASHES` constant rather than computing the hash of an empty byte sequence. The result is identical (the hash of `b""` is the null-hash constant for each algorithm), but returning the precomputed constant avoids unnecessary hash construction.

Null-hash constants

The `NULL_HASHES` module-level constant is computed once at import time:

```
NULL_HASHES = HashSet(
    md5=hashlib.md5(b"").hexdigest().upper(),
    sha256=hashlib.sha256(b"").hexdigest().upper(),
    sha512=hashlib.sha512(b"").hexdigest().upper(),
)
```

> **Historical note (DEV-09):** The original hardcodes null-hash constants as literal hex strings in multiple locations (e.g., `D41D8CD98F00B204E9800998ECF8427E` for MD5). The tool computes them once from `hashlib`, which is self-documenting, immune to copy-paste errors, and automatically correct if the algorithm set changes.

Directory identity scheme

`hash_directory_id()` implements the original's two-layer directory identity algorithm:

1. Compute `hash(directory_name)` → `name_digest` (a hex string).
2. Compute `hash(parent_directory_name)` → `parent_digest` (a hex string).
3. Concatenate the two hex strings: `combined = name_digest + parent_digest`.
4. Compute `hash(combined)` → `final_digest`.

This is performed independently for each active hash algorithm. Step 3 concatenates the uppercase hex representations of the digests (not the raw bytes), matching the original's `[BitConverter]::ToString()` output concatenation. The concatenation order is name-first, parent-second.

When the parent name is an empty string (the directory is at a filesystem root), step 2 produces the null-hash constant. This matches the original's behavior — `DirectoryId-HashString` returns the hardcoded null-hash for empty inputs.

Identity prefix convention

Item type	Prefix	Example
File	`y`	`yA8A8C089A6A8583B24C85F5A4A41F5AC`
Directory	`x`	`x3B4F479E9F880E438882FC34B67D352C`
Generated metadata	`z`	`z7E240DE74FB1ED08FA08D38063F6A6A9`

The `select_id()` function applies the appropriate prefix to the digest selected by `config.id_algorithm` (either `"md5"` or `"sha256"`). The prefix is prepended as a literal character — `f"{prefix}{digest}"` — producing the `id` field value for the output schema.

The `z` prefix for generated metadata entries (exiftool output) is a v2 addition. In v1, generated metadata entries use the `y` prefix (same as files) because their identity is derived from content hashing. In v2, the `z` prefix provides a namespace that distinguishes generated entries from sidecar entries without inspecting the `origin` field.

Uppercase convention

All hex digest strings are uppercased via `hexdigest().upper()`. The original uses `.ToUpper()` on `[BitConverter]::ToString()` output (which produces hyphen-separated uppercase hex, with the hyphens stripped via `.Replace('-', '')`). Python's `hashlib.hexdigest()` produces lowercase hex without separators; the `.upper()` call normalizes to the convention established by the original.

Symlink hashing fallback

When a file is a symlink, content hashing is replaced by name hashing — `hash_file()` is not called, and `hash_string(filename)` is used instead to produce the `hashes` field. The symlink detection itself is handled by the entry builder (§6.8) or by a dedicated check described in §6.4; the hashing module does not detect symlinks. It simply provides `hash_file()` for content and `hash_string()` for names, and the caller chooses which to invoke.

This matches the original's behavior in `FileId`, where the `ReparsePoint` attribute check gates whether the content hash or name hash is used.

6.4. Symlink Detection

Module: `core/entry.py` (inline within entry construction), also consulted in `core/hashing.py` call decisions **Operations Catalog:** Category 4 **Original functions:** `FileId` (ReparsePoint attribute check), `DirectoryId` (hardcoded `IsLink = $false`), `MakeObject` (reads `.IsLink` from identity result)

Purpose

Determines whether a file or directory is a symbolic link. This determination controls two downstream behaviors: the hashing strategy (content hashing vs. name hashing fallback for files) and the `attributes.is_link` field in the output schema.

Detection mechanism

Symlink detection uses a single call: `path.is_symlink()`. This works cross-platform:

Platform	Underlying mechanism
Windows	Detects NTFS reparse points (symlinks, junctions). Equivalent to the original's `Attributes -band ReparsePoint` check.
Linux/macOS	Detects POSIX symbolic links via `lstat()`.

The call is performed on the original path (before symlink resolution) using `os.lstat()` semantics — `Path.is_symlink()` does not follow the link.

> **Historical note:** The original uses `(Get-Item).Attributes -band [System.IO.FileAttributes]::ReparsePoint`, a Windows-specific bitwise attribute check. `Path.is_symlink()` is the correct cross-platform equivalent and is strictly simpler.

Behavioral effects

When `is_symlink` is `True`:

Operation	Normal behavior	Symlink behavior
File content hashing (§6.3)	`hash_file(path)` — hash file bytes	`hash_string(filename)` — hash name string instead
EXIF extraction (§6.6)	Invoke exiftool	Skip entirely — return `None`
Timestamp extraction (§6.5)	`os.stat()` (follows symlink)	`os.lstat()` (reads symlink metadata itself)
`hashes` field in output	Content hash digest set	Name hash digest set (same values as `name.hashes`)

For directory symlinks, the identity scheme is unchanged — directory identity is always based on name hashing (the two-layer scheme in §6.3), so no fallback is needed.

Dead code: `ValidateIsLink`

The original lists `ValidateIsLink` as a dependency of `MakeIndex` but never calls it. The `is_symlink` check is performed inline within `FileId` and `DirectoryId`. `ValidateIsLink` is not carried forward (DEV-13).

Dangling symlinks

A dangling symlink (one whose target does not exist) is treated as an item-level warning. `Path.is_symlink()` returns `True` for dangling symlinks (it reads the link itself, not its target). The entry builder proceeds with degraded fields: `hashes` are computed from the name, `size` is `null`, timestamps come from `os.lstat()` (the symlink's own metadata), and exif/sidecar operations are skipped. A warning is logged identifying the dangling link.

> **Historical note:** The original does not explicitly handle dangling symlinks. A missing link target causes `Get-Item` to fail, producing platform-dependent error behavior. The explicit handling here is a robustness improvement.

6.5. Filesystem Timestamps and Date Conversion

Module: `core/timestamps.py` **Operations Catalog:** Category 5 **Original functions:** `MakeObject` (timestamp reading and formatting), `Date2UnixTime` (string-to-Unix conversion)

Purpose

Extracts filesystem timestamps from stat results and produces both Unix-millisecond integers and ISO 8601 formatted strings for each of the three standard timestamp types: accessed, created, and modified. The output populates the `timestamps` field of the v2 schema (`TimestampsObject` containing three `TimestampPair` values — see §5.7).

Public interface

```
def extract_timestamps(
    stat_result: os.stat_result,
    *,
    is_symlink: bool = False,
) -> TimestampsObject:
    """Derive all timestamps from an os.stat_result.

    Returns a TimestampsObject containing accessed, created, and
    modified TimestampPairs, each with both ISO 8601 and Unix
    millisecond representations.

    When is_symlink is True, the stat_result is expected to come
    from os.lstat() (symlink metadata) rather than os.stat()
    (target metadata).
    """
```

The stat result is obtained by the entry builder (§6.8) before calling this function. The entry builder chooses between `os.stat()` (for regular files/directories) and `os.lstat()` (for symlinks) and passes the result.

Derivation — Unix timestamps

Unix timestamps are derived directly from the stat result's floating-point values, converted to milliseconds:

Timestamp	Stat attribute	Conversion
`accessed.unix`	`st_atime`	`int(stat_result.st_atime * 1000)`

Timestamp	Stat attribute	Conversion
`modified.unix`	`st_mtime`	`int(stat_result.st_mtime * 1000)`
`created.unix`	See below	See below

> **Historical note (DEV-07):** The original performs an unnecessary round-trip: it formats a datetime to a string via `.ToString($DateFormat)`, then passes that string to `Date2UnixTime`, which parses it back into a `DateTimeOffset` to call `.ToUnixTimeMilliseconds()`. The tool derives Unix timestamps directly from the stat float values — no intermediate string representation, no round-trip parsing, and `Date2UnixTime` (along with its entire internal dependency chain: `Date2FormatCode`, `Date2UnixTimeSquash`, `Date2UnixTimeCountDigits`, `Date2UnixTimeFormatCode`) is eliminated.

Creation time portability

Creation time handling varies by platform and is one of the primary portability concerns for the timestamp module (see §15.5 for the full platform analysis). The resolution strategy is:

1. Attempt `stat_result.st_birthtime`. This is the true file creation time and is available on macOS (all filesystems), Windows (NTFS), and some Linux configurations (kernel 4.11+ with `statx` support on ext4/XFS/Btrfs). If the attribute exists, use it.
2. Fall back to `stat_result.st_ctime`. On Windows, `st_ctime` is the creation time (Python's Windows `stat()` maps it correctly). On Linux and macOS, `st_ctime` is the inode change time (metadata modification), not the creation time — but it is the best available approximation.

The implementation wraps the `st_birthtime` access in a try/except for `AttributeError`, since the attribute is absent on platforms that do not support it. This is not a per-file error condition — it is a platform characteristic discovered once. The timestamps module MAY cache the platform's creation-time capability after the first successful or failed `st_birthtime` access to avoid redundant exception handling on subsequent calls.

```
# Illustrative — not the exact implementation.
def _get_creation_time(stat_result: os.stat_result) -> float:
    try:
        return stat_result.st_birthtime
    except AttributeError:
        return stat_result.st_ctime
```

No warning is emitted for the `st_ctime` fallback — it is expected behavior on most Linux systems and would produce noise without actionable information.

Derivation — ISO 8601 strings

ISO strings are produced from `datetime` objects constructed from the same stat float values:

```
# Illustrative — not the exact implementation.
from datetime import datetime, timezone

def _stat_to_iso(timestamp_float: float) -> str:
    dt = datetime.fromtimestamp(timestamp_float, tz=timezone.utc).astimezone()
    return dt.isoformat(timespec="microseconds")
```

The `datetime.fromtimestamp()` call interprets the float as seconds since the Unix epoch and attaches the UTC timezone. The `.astimezone()` call converts to the local timezone, producing an ISO 8601 string with the local timezone offset (e.g., `2024-03-15T14:30:22.123456-04:00`).

The `timespec="microseconds"` argument produces 6-digit fractional seconds. The original's .NET format string `yyyy-MM-ddTHH:mm:ss.fffffffzzz` produces 7-digit fractional seconds. Python's `datetime` provides microsecond precision (6 digits); .NET provides tick precision (7 digits). This is a minor, acceptable deviation — the 7th digit is always zero in practice for filesystem timestamps, which have at most microsecond resolution.

The `TimestampPair` assembly

For each of the three timestamp types, the module constructs a `TimestampPair` (§5.2.4):

```
TimestampPair(
    iso=_stat_to_iso(timestamp_float),
    unix=int(timestamp_float * 1000),
)
```

These are assembled into a `TimestampsObject` (§5.2.5):

```
TimestampsObject(
    accessed=TimestampPair(...),
    created=TimestampPair(...),
    modified=TimestampPair(...),
)
```

The complete `TimestampsObject` is returned to the entry builder, which places it directly into the `IndexEntry.timestamps` field.

6.6. EXIF and Embedded Metadata Extraction

Module: `core/exif.py` **Operations Catalog:** Category 6 **Original functions:** `GetFileExif`, `GetFileExifArgsWrite`, `GetFileExifRun`

Purpose

Invokes the `exiftool` binary to extract embedded EXIF, XMP, and IPTC metadata from a file. The extracted metadata is returned as a Python dictionary (parsed from exiftool's JSON output), with unwanted system keys removed. The result is wrapped into a `MetadataEntry` with `origin: "generated"` and `attributes.type: "exiftool.json_metadata"` during entry construction (§6.8).

Public interface

```
def extract_exif(
    path: Path,
    config: IndexerConfig,
) -> dict | None:
    """Extract embedded metadata from a file using exiftool.

    Returns a dict of metadata key-value pairs, or None if:
      - exiftool is not available
      - the file extension is in the exclusion list
      - exiftool returns an error or no data
      - the item is a symlink
    """
```

Exiftool availability probe

Before the first exiftool invocation, the module checks whether `exiftool` is available on the system PATH. This probe happens once per process lifetime (not once per file) and the result is cached in a module-level variable.

```
# Illustrative — not the exact implementation.
_exiftool_available: bool | None = None  # None = not yet checked

def _check_exiftool() -> bool:
    global _exiftool_available
    if _exiftool_available is None:
        _exiftool_available = shutil.which("exiftool") is not None
        if not _exiftool_available:
            logger.warning(
                "exiftool not found on PATH; embedded metadata extraction disabled"
            )
    return _exiftool_available
```

> **Historical note:** The original invokes exiftool for every eligible file without checking availability first. If exiftool is missing, each invocation fails independently, producing a per-file error. The probe-once approach avoids spawning doomed subprocesses and reduces log noise to a single warning. See §4.5 for the full discussion.

Extension exclusion

Before invoking exiftool, the module checks `path.suffix.lower()` (without the leading dot) against `config.exiftool_exclude_extensions`. The default exclusion set, carried forward from the original's `$global:MetadataFileParser.Exiftool.Exclude`, includes file types where exiftool tends to dump the entire file content into the metadata output rather than extracting meaningful embedded metadata:

`csv`, `htm`, `html`, `json`, `tsv`, `xml`

The exclusion list is configurable (§7.4). When a file is excluded, the function returns `None` and a debug-level message is logged.

Invocation

The exif module supports two invocation backends, selected automatically based on package availability. Both backends use the same exiftool argument set; they differ only in process lifecycle management and argument delivery mechanism.

Primary backend: `pyexiftool` batch mode (DEV-16). When the `pyexiftool` package is installed (it is a required runtime dependency), the module uses `exiftool.ExifToolHelper` to maintain a single persistent exiftool process for the duration of the indexing run. File paths and arguments are written to exiftool's stdin pipe — one per line — using exiftool's `-stay_open` protocol. This is semantically equivalent to a continuously open argfile and inherits the same Unicode safety guarantees documented in exiftool's FAQ §18. Per-file cost drops from 200–500 ms (process startup) to 20–50 ms (metadata extraction only).

```
# Illustrative — not the exact implementation.
import exiftool

EXIFTOOL_COMMON_ARGS = [
    "-json",
    "-n",
    "-extractEmbedded",
    "-scanForXMP",
    "-unknown2",
    "-G3:1",
    "-struct",
    "-ignoreMinorErrors",
    "-charset", "filename=utf8",
    "-api", "requestall=3",
    "-api", "largefilesupport=1",
]

with exiftool.ExifToolHelper(common_args=EXIFTOOL_COMMON_ARGS) as et:
    metadata_list = et.get_metadata(str(path))
```

Error isolation in batch mode is handled by wrapping each per-file `get_metadata()` call in a try/except. When `ExifToolHelper` raises `ExifToolExecuteError` for a non-zero exit code, the handler attempts metadata recovery before discarding the result: if the exception's stdout contains valid JSON with metadata keys beyond `SourceFile`, the module parses and filters the output normally (§17.5). ExifTool returns exit code 1 for informational conditions such as "unknown file type" but still produces usable system-level metadata (file size, timestamps, attributes) — discarding this output would be a behavioral regression against the original MakeIndex.

The persistent exiftool process is **not** reset on a per-file non-zero exit. A non-zero exit code indicates a per-file condition (e.g., unrecognized format), not a process health failure. Only actual process death (broken pipe, timeout, context manager failure) triggers a process reset and re-entry of the `ExifToolHelper` context. This distinction avoids the performance penalty of unnecessary process restarts.

When recovered metadata contains an `ExifTool:Error` field with value `"Unknown file type"`, the event is logged at `INFO` level (not WARNING): `"ExifTool: unknown file type for <filename>; system metadata preserved"`. Other `ExifTool:Error` values are similarly logged at `INFO` with the error text.

The `-quiet` flag is appended to `common_args` when the logging level is above DEBUG.

Fallback backend: `subprocess.run()` with argfile. If `pyexiftool` is unavailable or if the persistent process cannot be maintained (e.g., in constrained environments), the module falls back to a per-file `subprocess.run()` invocation. Arguments are written to a temporary file via `tempfile.NamedTemporaryFile` and passed to exiftool via its `-@` argfile switch, with `-charset filename=utf8` ensuring Unicode filename safety:

```
# Illustrative — not the exact implementation.
import tempfile

args_content = "\n".join(EXIFTOOL_COMMON_ARGS + [str(path)])
with tempfile.NamedTemporaryFile(mode="w", suffix=".args", delete=True, encoding="utf-8") as f:
    f.write(args_content)
    f.flush()
    result = subprocess.run(
        ["exiftool", "-@", f.name],
        capture_output=True,
        text=True,
        timeout=30,
    )
```

> **Historical note (DEV-05, DEV-16):** The original stores exiftool arguments as Base64-encoded strings in the PowerShell source, decodes them at runtime via `Base64DecodeString` (which itself calls `certutil` on Windows), writes them to a temporary file via `TempOpen`, passes the temporary file to exiftool via its `-@` argfile switch, and cleans up via `TempClose`. The Base64 pipeline is eliminated entirely. The primary backend (`pyexiftool`) communicates via stdin pipes with no disk I/O for arguments. The fallback backend writes a temporary argfile using `tempfile`, which is cross-

> platform and handles cleanup automatically. Both backends define arguments as plain Python lists — no Base64 encoding, no `certutil`, no Windows-specific decoding.

The `timeout=30` parameter (fallback backend only — the batch backend enforces timeouts at the `ExifToolHelper` level) prevents a hung exiftool process from blocking the indexer indefinitely. If the timeout is exceeded, the function returns `None` and a warning is logged. The timeout value is not currently configurable but MAY be exposed in a future configuration update if users encounter legitimate long-running extractions.

Output parsing and key filtering

Exiftool's `-json` flag produces a JSON array containing one object per input file. Since the tool processes one file at a time, the output is always a single-element array. The module parses the output via `json.loads()` and extracts the first element.

> **Historical note (DEV-06):** The original pipes exiftool output through `jq` for two purposes: compacting the JSON (`jq -c '.[] | .'`) and deleting unwanted keys via a second `jq` pass (`jq -c 'del(.ExifToolVersion, .FileSequence, ...)'`). `jq` is eliminated entirely. `json.loads()` handles parsing natively, and unwanted keys are removed with a dict comprehension.

The unwanted key set includes exiftool operational metadata and OS-specific filesystem attributes that are either redundant with IndexEntry fields or not embedded metadata from the file. Because exiftool with `-G` flags emits group-prefixed keys (e.g. `System:FileName`), the filter matches by **base key name** — the portion after the last `:` separator. This handles both prefixed and unprefixed key forms.

The complete exclusion set:

Base key name	Category	Reason
`ExifToolVersion`	Operational	ExifTool process version
`FileSequence`	Operational	ExifTool internal counter
`NewGUID`	Operational	ExifTool-generated GUID
`Directory`	Filesystem	Parent directory path
`FileName`	Filesystem	File name (redundant with IndexEntry)
`FilePath`	Filesystem	Full path (redundant/exposes layout)
`BaseName`	Filesystem	Stem without extension
`FilePermissions`	Filesystem	OS-specific permissions string
`SourceFile`	Filesystem	Absolute path as provided to exiftool
`FileSize`	Redundant	Already in IndexEntry `size` object
`FileModifyDate`	Redundant	Already in IndexEntry `timestamps`
`FileAccessDate`	Redundant	Already in IndexEntry `timestamps`
`FileCreateDate`	Redundant	Already in IndexEntry `timestamps`
`FileAttributes`	OS-specific	Filesystem attributes (not embedded)
`FileDeviceNumber`	OS-specific	Device identifier
`FileInodeNumber`	OS-specific	Inode (always 0 on Windows)
`FileHardLinks`	OS-specific	Hard link count
`FileUserID`	OS-specific	UID (always 0 on Windows)
`FileGroupID`	OS-specific	GID (always 0 on Windows)
`FileDeviceID`	OS-specific	Device ID
`FileBlockSize`	OS-specific	Block size (empty on Windows)
`FileBlockCount`	OS-specific	Block count (empty on Windows)
`Now`	Operational	ExifTool processing timestamp
`ProcessingTime`	Operational	ExifTool processing duration

The filtering implementation:

```
def _base_key(key: str) -> str:
    """Extract the base key name, stripping any group prefix."""
    return key.rsplit(":", 1)[-1]

EXIFTOOL_EXCLUDED_KEYS = frozenset({
```

```
    "ExifToolVersion", "FileSequence", "NewGUID", "Directory",
    "FileName", "FilePath", "BaseName", "FilePermissions",
    "SourceFile", "FileSize", "FileModifyDate", "FileAccessDate",
    "FileCreateDate", "FileAttributes", "FileDeviceNumber",
    "FileInodeNumber", "FileHardLinks", "FileUserID", "FileGroupID",
    "FileDeviceID", "FileBlockSize", "FileBlockCount",
    "Now", "ProcessingTime",
})

filtered = {k: v for k, v in raw_data.items() if _base_key(k) not in EXIFTOOL_EXCLUDED_KEYS}
```

The excluded key set is configurable via TOML configuration following the standard collection field conventions (§7.7):

Config key	Type	Behavior
`exiftool.exclude_keys`	list of strings	**Replace** — the specified list becomes the complete exclusion set, overriding all compiled defaults.
`exiftool.exclude_keys_append`	list of strings	**Append** — entries are added to the compiled default set.

If neither key is present, the compiled default set (the table above) applies unchanged. Keys are matched by base name to accommodate the `-G` flag's group prefixes regardless of whether the exclusion set is the default or a user-provided override. See §7.4 for the extension exclusion list, which follows the same replace/append pattern.

Error handling

Condition	Behavior	Severity tier
Exiftool not on PATH	Return `None`. Single warning on first probe.	Field-level
Extension in exclusion list	Return `None`. Debug log.	Diagnostic
Item is a symlink	Return `None`. Debug log.	Diagnostic
Exiftool returns non-zero	Attempt metadata recovery: if the exception or process stdout contains valid JSON with keys beyond `SourceFile`, parse, filter, and return the metadata normally. If `ExifTool:Error` is present, log at INFO. Only return `None` (with WARNING) when no valid metadata can be recovered. Do **not** reset the persistent process.	Field-level
Exiftool output is not valid JSON	Return `None`. Warning with parse error.	Field-level
Exiftool times out	Return `None`. Warning.	Field-level
Exiftool returns empty metadata	Return `None`. Debug log.	Diagnostic

In all cases, the entry builder proceeds with `metadata` containing no exiftool entry (or with the exiftool entry omitted from the metadata array). No exiftool failure is fatal.

Backend selection logic

The exif module selects its invocation backend once at module load time. The selection is logged at `DEBUG` level:

1. If `pyexiftool` is importable and exiftool is on PATH → use batch mode (primary).
2. If `pyexiftool` is not importable but exiftool is on PATH → use subprocess+argfile mode (fallback). Log an `INFO`-level message noting that batch mode is unavailable.
3. If exiftool is not on PATH → disable metadata extraction entirely. Log a single `WARNING`.

The backend selection is cached in a module-level variable and never re-evaluated during the process lifetime.

6.7. Sidecar Metadata File Handling

Module: `core/sidecar.py` **Operations Catalog:** Category 7 **Original functions:** `GetFileMetaSiblings`, `ReadMetaFile`, `MetaFileRead` (and its 20+ sub-functions)

Purpose

Discovers, classifies, reads, and parses sidecar metadata files that live alongside the files they describe. A sidecar file is any file in the same directory as the indexed item whose name matches a known metadata suffix pattern (e.g., `video.mp4.info.json`, `photo.jpg_thumbnail.jpg`, `document.pdf.description`). The module produces `MetadataEntry` objects (§5.10) for each discovered sidecar, carrying the full provenance information needed for MetaMergeDelete reversal.

Public interface

```
def discover_and_parse(
    item_path: Path,
    item_name: str,
    siblings: list[Path],
    config: IndexerConfig,
    delete_queue: list[Path] | None = None,
) -> list[MetadataEntry]:
    """Discover and parse sidecar metadata files for an item.

    Args:
        item_path: Absolute path to the indexed item.
        item_name: The item's filename (used for suffix matching).
        siblings: Pre-enumerated list of all files in the same directory.
        config: Configuration containing identification patterns and
                exclusion rules.
        delete_queue: When MetaMergeDelete is active, sidecar paths are
                      appended here for deferred deletion.

    Returns a list of MetadataEntry objects, one per discovered sidecar.
    Returns an empty list if no sidecars are found.
    """
```

The `siblings` list is provided by the entry builder, which already has the directory listing from `list_children()`. This avoids re-scanning the directory for each file being indexed — a significant optimization for directories containing many files with few sidecars.

Sidecar discovery

Discovery matches sibling filenames against the indexed item's name combined with known metadata suffix patterns. The identification patterns are defined in `config.metadata_identify` — a dict mapping type names to lists of compiled regex patterns. These patterns are ported from the original `$global:MetadataFileParser.Identify` configuration.

The discovery algorithm:

1. Escape the indexed item's filename for use in regex: `escaped_name = re.escape(item_name)`.
2. For each sibling file in the directory, check whether its name matches any pattern in the identification configuration. A match means the sibling is a sidecar of the identified type.
3. Exclude sidecars whose names match the configured exclusion patterns (`config.metadata_exclude_patterns`).
4. Return the matched sidecars grouped by type.

> **Historical note:** The original's `GetFileMetaSiblings` rescans the parent directory via `Get-ChildItem` for every indexed file. In a directory with 1,000 files, this means 1,000 redundant directory reads. Passing the pre-enumerated `siblings` list performs the directory read once.

Type detection

Each discovered sidecar is classified into exactly one type by matching its filename against the type identification patterns. The recognized types (carried forward from the original `$global:MetadataFileParser.Identify` keys) are:

Type key	Description	Data handling
`description`	Text description files (youtube-dl `.description`)	JSON → text → binary fallback
`desktop_ini`	Windows `desktop.ini` files	Text
`generic_metadata`	Generic config/metadata (`.cfg`, `.conf`, `.yaml`, `.meta`)	JSON → text → binary fallback
`hash`	Hash/checksum files (`.md5`, `.sha256`, `.crc32`)	Lines (non-empty lines only)
`json_metadata`	JSON metadata (`.info.json`, `.meta.json`)	JSON

Type key	Description	Data handling
`link`	URL shortcuts (`.url`) and filesystem shortcuts (`.lnk`)	URL/path extraction
`screenshot`	Screen capture images	Base64-encoded binary
`subtitles`	Subtitle tracks (`.srt`, `.sub`, `.vtt`, `.lrc`)	JSON → text → binary fallback
`thumbnail`	Thumbnail/cover images (`.cover`, `.thumb`)	Base64-encoded binary
`torrent`	Torrent files	Base64-encoded binary

If a sidecar matches zero patterns, it is ignored (not an error — the file simply is not a recognized sidecar type). If a sidecar matches multiple patterns, it is logged as a warning and classified as the first match.

> **Historical note:** The original also takes the first match when iterating `$MetadataFileParser.Identify` keys.

Format-specific readers

After type detection, the sidecar's content is read by a format-specific handler. The handler selection follows a type-to-reader mapping:

JSON reader. Reads the file content and parses it via `json.loads()`. Used directly for `json_metadata` type. Used as the first attempt in the fallback chain for `description`, `generic_metadata`, and `subtitles`.

> **Historical note:** The original's `MetaFileRead-Data-ReadJson` pipes the file through `jq -c '.'` and then `ConvertFrom-Json`. Using `json.loads()` directly eliminates the `jq` dependency for sidecar parsing as well.

Text reader. Reads the file as UTF-8 text via `path.read_text(encoding="utf-8")`. Returns the content as a string. Used for `description` (when JSON parsing fails), `generic_metadata` (same), and `desktop_ini`.

Lines reader. Reads the file as UTF-8 text and splits into a list of non-empty lines. Used for `hash` files and `subtitles` (when JSON and text fallbacks are exhausted).

Binary reader. Reads the file as raw bytes and Base64-encodes them via `base64.b64encode(path.read_bytes()).decode("ascii")`. Used for `screenshot`, `thumbnail`, and `torrent` types, and as the final fallback for types where text and JSON reading both fail.

> **Historical note:** The original's binary reader (`MetaFileRead-Data-Base64Encode`) uses `certutil -encode` to convert binary data to Base64, writing to a temporary file and stripping the header/footer lines that `certutil` adds. Using `base64.b64encode()` directly eliminates the external binary dependency and temporary file.

Link reader. For `.url` files, parses the INI-format content to extract the `URL=` value. For `.lnk` files on Windows, `pylnk3` or `comtypes` MAY be used to resolve the shortcut target; on non-Windows platforms, `.lnk` files are read as binary (Base64-encoded), since the `.lnk` format is Windows-specific.

> **Historical note:** The original uses the external pslib functions `UrlFile2Url` and `Lnk2Path` for link resolution. The tool internalizes this logic.

Fallback chain

For types that support multiple formats (`description`, `generic_metadata`, `subtitles`), the reader attempts formats in order:

1. JSON → if valid JSON, store as `format: "json"` with `transforms: ["json_compact"]`.
2. Text → if readable as UTF-8 text, store as `format: "text"` with `transforms: []`.
3. Binary → Base64-encode the raw bytes, store as `format: "base64"` with `transforms: ["base64_encode"]`.

Each step catches the relevant exception (JSON parse error, Unicode decode error) and falls to the next. Only if all three fail is the sidecar recorded with `attributes.type: "error"` and `data: null`.

MetadataEntry construction

For each successfully read sidecar, the module constructs a `MetadataEntry` (§5.10) with full provenance:

MetadataEntry field	Source
`id`	`"y" + hash_file(sidecar_path).sha256` (sidecar content hash)
`origin`	`"sidecar"`
`name`	`NameObject(text=sidecar_filename, hashes=hash_string(sidecar_filename))`
`hashes`	`hash_file(sidecar_path)` (content hashes of the original sidecar file)
`file_system`	`{"relative": relative_path_from_index_root}`
`size`	`SizeObject(bytes=sidecar_stat.st_size, text=human_readable_size)`
`timestamps`	`extract_timestamps(sidecar_stat)`

MetadataEntry field	Source
`attributes.type`	Detected type (e.g., `"json_metadata"`, `"thumbnail"`)
`attributes.format`	Data format (`"json"`, `"text"`, `"base64"`, `"lines"`)
`attributes.transforms`	Applied transforms (e.g., `["base64_encode"]`, `["json_compact"]`)
`attributes.source_media_type`	MIME type of binary sidecars (e.g., `"image/jpeg"` for thumbnails); `null` for text-based sidecars
`data`	Parsed content (varies by format)

The provenance fields (`file_system`, `size`, `timestamps`) are the v2 additions that enable MetaMergeDelete reversal (§5.10, principle P3). They are present only for sidecar entries (`origin: "sidecar"`) and absent for generated entries.

MetaMergeDelete queue

When `config.meta_merge_delete` is `True` and the `delete_queue` parameter is not `None`, each successfully parsed sidecar's absolute path is appended to the delete queue. The actual deletion is deferred to Stage 6 post-processing (§4.4) — the sidecar module only records the path. This separation ensures that if the process is interrupted, no sidecar files have been deleted while their parent entries may be only partially written.

The original's `$global:DeleteQueue` pattern is preserved but with explicit parameter passing rather than global state.

6.8. Index Entry Construction

Module: `core/entry.py` **Operations Catalog:** Category 8 **Original functions:** `MakeObject`, `MakeFileIndex`, `MakeDirectoryIndex`, `MakeDirectoryIndexLogic`, `MakeDirectoryIndexRecursive`, `MakeDirectoryIndexRecursiveLogic`

Purpose

Orchestrates the assembly of a complete `IndexEntry` (the v2 schema object defined in §5) from a filesystem path. This is the hub of the hub-and-spoke architecture described in §4.2 — `entry.py` is the sole module that calls into the component modules (`paths`, `hashing`, `timestamps`, `exif`, `sidecar`) and wires their outputs together into the final schema object. No component module calls another component module directly; all coordination flows through `entry.py`.

Public interface

```
def build_file_entry(
    path: Path,
    config: IndexerConfig,
    siblings: list[Path] | None = None,
    delete_queue: list[Path] | None = None,
) -> IndexEntry:
    """Build a complete IndexEntry for a single file.

    Args:
        path: Absolute path to the file.
        config: Resolved configuration.
        siblings: Pre-enumerated sibling files in the same directory
                  (for sidecar discovery). If None, the module will
                  enumerate the parent directory.
        delete_queue: MetaMergeDelete accumulator (see §6.7).

    Returns a fully populated IndexEntry conforming to the v2 schema.
    """

def build_directory_entry(
    path: Path,
    config: IndexerConfig,
    recursive: bool = False,
    delete_queue: list[Path] | None = None,
    progress_callback: Callable[[ProgressEvent], None] | None = None,
    cancel_event: threading.Event | None = None,
) -> IndexEntry:
    """Build a complete IndexEntry for a directory.

    When recursive=True, descends into subdirectories and populates
    the items field with a fully nested tree of child IndexEntry objects.
    When recursive=False, populates items with only immediate children.

    Args:
        path: Absolute path to the directory.
```

```
        config: Resolved configuration.
        recursive: Whether to descend into subdirectories.
        delete_queue: MetaMergeDelete accumulator.
        progress_callback: Optional callable invoked after each child
            item is processed and once after child discovery. See
            ProgressEvent (\xc2\xa79.4) for the event contract.
        cancel_event: Optional threading.Event checked before each
            child item. When set, raises IndexerCancellationError.

    Returns a fully populated IndexEntry conforming to the v2 schema.

    Raises:
        IndexerCancellationError: cancel_event was set during
            the child-processing loop.
    """

def index_path(
    target: Path,
    config: IndexerConfig,
    *,
    progress_callback: Callable[[ProgressEvent], None] | None = None,
    cancel_event: threading.Event | None = None,
) -> IndexEntry:
    """Top-level entry point: classify target and dispatch.

    This is the single function consumed by the CLI, GUI, and public API.
    Resolves the target, determines whether it is a file or directory,
    and delegates to build_file_entry() or build_directory_entry().
    Forwards progress_callback and cancel_event to
    build_directory_entry() for directory targets; both parameters
    are ignored for single-file targets.
    """
```

File entry construction sequence

`build_file_entry()` executes the following steps in order. Each step calls into a component module and contributes one or more fields to the final `IndexEntry`.

Step 1 — Path components. Call `paths.extract_components(path)` to obtain `name`, `stem`, `suffix`, `parent_name`, and `parent_path`. Validate the extension via `paths.validate_extension()`.

Step 2 — Stat and symlink detection. Call `path.is_symlink()` to determine symlink status. Call `path.stat()` (or `path.lstat()` for symlinks) to obtain the stat result for timestamp and size extraction.

Step 3 — Hashing. If the file is not a symlink, call `hashing.hash_file(path)` to compute content hashes. If it is a symlink, call `hashing.hash_string(name)` to compute name hashes as the fallback. Also call `hashing.hash_string(name)` unconditionally to produce the `name.hashes` field.

Step 4 — Identity selection. Call `hashing.select_id(content_hashes, config.id_algorithm, "y")` to derive the `id` field.

Step 5 — Timestamps. Call `timestamps.extract_timestamps(stat_result, is_symlink=...)` to produce the `TimestampsObject`.

Step 6 — Size. Construct a `SizeObject` from `stat_result.st_size`. The `text` field is a human-readable representation (e.g., `"1.23 MB"`). For symlinks where the stat result comes from `lstat()`, the size reflects the symlink entry itself, not the target.

Step 7 — Parent identity. Compute the parent directory's identity via `hashing.hash_directory_id(parent_name, grandparent_name)`, then `hashing.select_id(..., "x")`. Construct a `ParentObject`. When the file is at a filesystem root (empty parent name), set `parent` to `null`.

Step 8 — EXIF metadata. If `config.extract_exif` is `True` and the file is not a symlink, call `exif.extract_exif(path, config)`. If the result is non-None, wrap it in a `MetadataEntry` with `origin: "generated"`, `attributes.type: "exiftool.json_metadata"`, `attributes.format: "json"`, and `attributes.transforms: ["key_filter"]`.

Step 9 — Sidecar metadata. If `config.meta_merge` is `True`, call `sidecar.discover_and_parse(path, name, siblings, config, delete_queue)`. Collect the returned `MetadataEntry` objects.

Step 10 — Metadata assembly. Combine the exiftool entry (if any) and sidecar entries into the `metadata` array. If metadata processing was active but produced no results, `metadata` is an empty list `[]`. If metadata processing was not active (neither exif nor sidecar flags enabled), `metadata` is `null`. See §5.10 for the semantic distinction.

Step 11 — Storage name. Construct `attributes.storage_name` from the `id` and `extension`: `f"{id}.{extension}"` for files with extensions, or just `id` for files without.

Step 12 — Assembly. Construct the final `IndexEntry` with all fields populated. Set `schema_version` to `2`, `type` to `"file"`, `items` to `null`.

> **Historical note:** The original's `MakeObject` contains a `switch` statement with five near-identical branches (`makeobjectfile`, `makeobjectdirectory`, `makeobjectdirectoryrecursive`, plus defaults) that all construct the same `[PSCustomObject]@{...}` with minor variations. The tool uses a single construction path per item type — `build_file_entry()` for files and `build_directory_entry()` for directories — with no switch duplication.

Directory entry construction sequence

`build_directory_entry()` follows the same steps as file entry construction with these differences:

Step	File behavior	Directory behavior
3 — Hashing	`hash_file()` for content	`hash_directory_id(name, parent_name)` for name-based identity
4 — Identity	Prefix `y`	Prefix `x`
6 — Size	`stat_result.st_size`	Sum of all child sizes (computed after child entries are built)
8 — EXIF	Active for eligible files	Skipped (directories have no embedded metadata)
9 — Sidecar	Active for files	Not applicable for the directory itself (but active for child files)
12 — Items	`items = null`	`items = [child entries]`

After constructing its own identity and timestamps, `build_directory_entry()` enumerates children via `traversal.list_children(path, config)` and builds child entries in a loop that integrates progress reporting and cooperative cancellation:

1. **Discovery.** Call `traversal.list_children(path, config)` to obtain the sorted file and directory lists. If `progress_callback` is not `None`, emit a `ProgressEvent` with `phase="discovery"`, `items_total` set to `len(files) + len(directories)`, `items_completed=0`, and `level="info"`.
2. **Cancellation checkpoint.** At the top of each loop iteration (before processing the next child, whether file or subdirectory), check `cancel_event.is_set()` if `cancel_event` is not `None`. If the event is set, raise `IndexerCancellationError` immediately. No partial results are returned — the partially assembled `items` list is discarded.
3. **File children.** For each child file: call `build_file_entry(child, config, siblings=all_child_files, delete_queue=...)`. After the call returns (or the child is skipped due to error), if `progress_callback` is not `None`, emit a `ProgressEvent` with `phase="processing"`, the updated `items_completed` count, `current_path=child`, and `level="info"` (or `"warning"` if the child was skipped).
4. **Directory children.** For each child subdirectory (when `recursive=True`): call `build_directory_entry(child, config, recursive=True, delete_queue=..., progress_callback=progress_callback, cancel_event=cancel_event)`. The same `progress_callback` and `cancel_event` are forwarded so that progress spans the entire tree and cancellation is effective at any recursion depth. After the recursive call returns, emit a `ProgressEvent` with updated counts.
5. **Assembly.** Collect all child `IndexEntry` objects into the `items` list (files first, then directories, each group sorted by name).
6. **Size aggregation.** Compute the directory's `size.bytes` as the sum of all child `size.bytes` values (recursive — includes all descendants).

When `recursive=False` (flat mode), child subdirectories are still processed by `build_directory_entry()` but with `recursive=False`, so they have a single level of children (or none, depending on implementation — the original processes immediate children only in flat mode). The flat-mode behavior produces a two-level tree: the target directory containing its immediate children, with child directories having their own identity and timestamps but no populated `items`.

Error handling during construction

Per-item error handling follows the strategy defined in §4.5:

Error condition	Behavior
`stat()` fails (permission denied, I/O error)	Item is skipped entirely. Warning logged. Item excluded from parent's `items` array.
Content hashing fails (I/O error on read)	`hashes` field set to `null`. Warning logged. Entry included with degraded fields.
Exiftool fails for this file	Exiftool entry omitted from `metadata`. Warning logged.
Sidecar file cannot be read	That sidecar's `MetadataEntry` has `attributes.type: "error"` and `data: null`. Warning logged. Other sidecars unaffected.
Child entry construction fails	Child excluded from parent's `items`. Warning logged. Remaining children processed.
`cancel_event` is set	`IndexerCancellationError` raised immediately at the next item boundary. Partially built `items` list is discarded. This is an invocation-level interrupt, not an item-level failure.

The entry builder never raises exceptions for item-level or field-level failures. Cancellation is the sole exception to this rule — `IndexerCancellationError` is an invocation-level interrupt that propagates to the caller, not an item-level or field-level failure. Only fatal conditions (target path does not exist, configuration is invalid, or cancellation requested) produce exceptions that propagate to the caller.

6.9. JSON Serialization and Output Routing

Module: `core/serializer.py` **Operations Catalog:** Category 9 **Original functions:** `MakeIndex` top-level output logic, `ConvertTo-Json` calls in traversal functions

Purpose

Converts `IndexEntry` model instances to JSON text and routes the result to one or more output destinations. This is Stage 5 of the processing pipeline (§4.1). The serializer is a pure presentation layer — it does not modify the `IndexEntry` data, only formats and delivers it.

Public interface

```
def serialize_entry(
    entry: IndexEntry,
    *,
    compact: bool = False,
) -> str:
    """Serialize an IndexEntry to a JSON string.

    When compact=False, output is pretty-printed with 2-space indent.
    When compact=True, output is a single line.
    """

def write_output(
    entry: IndexEntry,
    config: IndexerConfig,
) -> None:
    """Route serialized output to configured destinations.

    Examines config.output_stdout, config.output_file, and
    config.output_inplace to determine where output goes.
    Multiple destinations may be active simultaneously.
    """

def write_inplace(
    entry: IndexEntry,
    item_path: Path,
    item_type: str,
) -> None:
    """Write a single in-place sidecar file alongside an item.

    Called during traversal for each item when inplace mode is active.
    The sidecar path is constructed via paths.build_sidecar_path().
    """
```

Serialization

Serialization converts an `IndexEntry` dataclass to JSON via a two-step process: `dataclasses.asdict(entry)` to produce a plain dict, followed by `json.dumps()` to produce the JSON string.

The serialization helper applies the invariants defined in §5.12:

1. `schema_version` is placed first in the output by using an `OrderedDict` or a custom key-sorting function. JSON objects are unordered, but the serializer places `schema_version` first by convention for human readability.
2. Optional `HashSet.sha512` fields are omitted (not emitted as `null`) when their value is `None`.
3. Sidecar-only `MetadataEntry` fields (`file_system`, `size`, `timestamps`) are present for sidecar entries and absent for generated entries, controlled by the `origin` discriminator.
4. `ensure_ascii=False` is passed to `json.dumps()` to produce UTF-8 output with non-ASCII characters preserved, matching the original's `Out-File -Encoding UTF8`.

When the `orjson` package is available, the serializer MAY use it as a drop-in replacement for faster serialization of large entry trees. `orjson` handles dataclasses natively and produces bytes rather than strings; the serializer wraps the output appropriately. The `orjson` path is gated by a try/except import and is transparent to callers.

> **Historical note:** The original uses `ConvertTo-Json -Depth 100` for serialization, which has a known memory ceiling for very large nested structures (documented in the original's own source comments). Python's `json.dumps()` handles arbitrarily deep nesting without this limitation.

Output routing

The output routing model simplifies the original's seven-scenario matrix into three independent boolean flags that compose naturally:

Flag	Config field	Behavior
`--stdout`	`config.output_stdout`	Write the serialized JSON to `sys.stdout`.
`--outfile PATH`	`config.output_file`	Write the serialized JSON to the specified file path.
`--inplace`	`config.output_inplace`	Write individual sidecar files alongside each processed item.

Any combination of these flags is valid. When no output flags are specified, the default is `--stdout` only.

> **Historical note:** The original's default behavior was also stdout-only output.

> **Historical note:** The original's 7-scenario routing switch (`StandardOutput`/`NoStandardOutput` combined with `OutFile`/`OutFileInPlace`/both/neither) is replaced by three independent flags. The routing logic becomes a simple check-and-write for each enabled destination, with no complex scenario matrix. The `NoStandardOutput` negative flag is eliminated — the absence of `--stdout` is sufficient.

Timing of writes

Destination	When written
`--stdout`	After the complete entry tree is assembled (end of Stage 4). The entire tree is serialized and written in one operation.
`--outfile`	After the complete entry tree is assembled. The entire tree is serialized and written to the file in one operation.
`--inplace`	During traversal (within the Stage 3–4 loop). Each item's sidecar file is written as soon as that item's `IndexEntry` is complete, before the next item is processed.

The in-place write timing is a deliberate design choice preserved from the original: it ensures that partial results survive process interruption. If the indexer is killed mid-traversal, the sidecar files written so far are valid and usable. Stdout and outfile writes, by contrast, happen only after full completion — there is no meaningful partial-output behavior for a single JSON document.

File encoding

All output files are written as UTF-8 without a BOM, using `Path.write_text(json_string, encoding="utf-8")`. The original uses `Out-File -Encoding UTF8` on Windows, which also produces UTF-8 (with or without BOM depending on PowerShell version). The tool normalizes to UTF-8-no-BOM across all platforms.

6.10. File Rename and In-Place Write Operations

Module: `core/rename.py` **Operations Catalog:** Category 10 **Original functions:** Rename logic in `MakeDirectoryIndexLogic` / `MakeDirectoryIndexRecursiveLogic`, `Move-Item` calls

Purpose

Implements the `StorageName` rename operation: renames files and directories from their original names to their deterministic, hash-based `storage_name` values (§5.8). The rename operation is destructive — the original filename is replaced on disk — but the original name is preserved in the `IndexEntry.name.text` field of the in-place sidecar file that is always written alongside a rename (since the sidecar serves as the manifest for reversal).

Public interface

```
def rename_item(
    original_path: Path,
    entry: IndexEntry,
    *,
    dry_run: bool = False,
) -> Path:
    """Rename a file or directory to its storage_name.

    Returns the new path after renaming. If dry_run=True, returns
    the would-be new path without performing the rename.

    Raises RenameError if the target path already exists and is not
    the same file (collision detection).
    """
```

Rename behavior

The rename operation constructs the target path via `paths.build_storage_path(original_path, entry.attributes.storage_name)` and executes `original_path.rename(target_path)`.

`Path.rename()` performs an atomic rename when the source and target are on the same filesystem. For cross-filesystem moves (which should not occur in normal usage — the rename target is always in the same directory), the operation falls back to `shutil.move()`.

Collision detection

Before renaming, the module checks whether the target path already exists. If it does, and it is not the same inode as the source (checked via `os.stat()` comparison), a `RenameError` is raised. This prevents data loss from two files that happen to produce the same `storage_name` (which would require an identity hash collision — astronomically unlikely, but the guard is cheap and worth having).

If the target path exists and is the same inode as the source (meaning the file was already renamed in a previous run), the rename is a no-op and the existing path is returned.

Rename implies in-place output

When the `--rename` flag is active, the configuration MUST also activate `--inplace` output. The rename module does not write sidecar files itself — that is handled by `serializer.write_inplace()`. The configuration loader enforces the implication: `config.rename = True` → `config.output_inplace = True`.

> **Historical note:** The original enforces this same constraint (`MakeIndex` forces `OutFileInPlace = $true` when `Rename = $true`).

The sidecar file written alongside each renamed item serves as the reversal manifest: it contains the original filename in `name.text`, allowing a future revert operation to reconstruct the original path.

Safety: MetaMergeDelete guard

When MetaMergeDelete is active, the configuration MUST require that at least one output mechanism (`--outfile` or `--inplace`) is enabled. This prevents the scenario where sidecar metadata files are deleted without their content being captured in any output. This constraint is enforced as a configuration validation rule during Stage 1.

> **Historical note:** The original enforces this via the `$MMDSafe` variable.

If MetaMergeDelete is requested with no output mechanism, the configuration loader raises a fatal error before any processing begins.

Dry-run mode

The `dry_run` parameter causes the function to compute and return the target path without performing the actual rename. The serializer still writes the sidecar file (using the would-be new path), and the `IndexEntry` still contains the `storage_name`. Dry-run mode allows users to preview what a rename operation would do before committing to it.

> **Historical note:** The original does not support dry-run for renames. This capability is a safety feature, particularly useful for users running rename operations on large directory trees for the first time.

Revert capability

The original's source comments include a "To-Do" note about adding a `Revert` parameter. The v2 schema's enriched `MetadataEntry` provenance fields (§5.10, principle P3) and the in-place sidecar files provide the data foundation for reversal. A `revert_rename()` function that reads the sidecar's `name.text` field and renames the file back to its original name is a natural post-MVP enhancement. The architecture supports it without structural changes — the sidecar file is the revert manifest.

7. Configuration

This section defines the externalized configuration system. It specifies every configurable field, its type, its default value, the TOML file format used for user overrides, and the layered resolution strategy that merges defaults, user files, and CLI/API arguments into a single immutable configuration object.

> **Historical note:** The original hardcoded all configuration in the `$global:MetadataFileParser` object and in literal values scattered throughout the `MakeIndex` function body. The externalized configuration system is a deliberate architectural improvement (design goal G4).

The configuration system implements design goal G4 (§2.3): a user SHOULD be able to add a new sidecar metadata pattern, extend the exiftool exclusion list, or modify the filesystem exclusion filters without editing source code. It also implements the no-global-state principle from §4.4: the configuration is constructed once during Stage 1 of the processing pipeline (§4.1), frozen into an immutable object, and threaded through the entire call chain as an explicit function parameter.

§3.2 defines the module layout (`config/types.py`, `config/defaults.py`, `config/loader.py`). §3.3 defines the file resolution paths. §4.2 defines the dependency rules (Rule 4: `config/` is consumed, not called back into). This section defines the content — what fields exist, what values they hold, and how they compose.

7.1. Configuration Architecture

The `IndexerConfig` dataclass

All configuration is represented by a single top-level frozen dataclass, `IndexerConfig`, defined in `config/types.py`. Every `core/` module that consumes configuration receives it as an `IndexerConfig` parameter. No module inspects environment variables, reads files, or accesses global state to obtain configuration at runtime — the `IndexerConfig` is the sole source of truth.

```
@dataclass(frozen=True)
class IndexerConfig:
    """Immutable configuration for a single indexing invocation."""

    # Target and traversal
    recursive: bool = True
    id_algorithm: str = "md5"  # "md5" or "sha256"
    compute_sha512: bool = False

    # Output routing
    output_stdout: bool = True
    output_file: Path | None = None
    output_inplace: bool = False

    # Metadata processing
    extract_exif: bool = False
    meta_merge: bool = False
    meta_merge_delete: bool = False

    # Rename
    rename: bool = False
    dry_run: bool = False

    # Filesystem exclusion filters
    filesystem_excludes: frozenset[str] = ...  # see §7.2
    filesystem_exclude_globs: tuple[str, ...] = ...

    # Extension validation
    extension_validation_pattern: str = ...  # compiled regex string

    # Exiftool
    exiftool_exclude_extensions: frozenset[str] = ...
    exiftool_args: tuple[str, ...] = ...

    # Metadata file parser
    metadata_identify: MappingProxyType[str, tuple[re.Pattern, ...]] = ...
    metadata_attributes: MappingProxyType[str, MetadataTypeAttributes] = ...
    metadata_exclude_patterns: tuple[re.Pattern, ...] = ...
    extension_groups: MappingProxyType[str, tuple[str, ...]] = ...
```

The `frozen=True` parameter ensures that the object is immutable after construction. Mutable collection types (`list`, `dict`, `set`) are replaced with their immutable counterparts (`tuple`, `frozenset`, `MappingProxyType`) to enforce this at the field level. If using Pydantic instead of stdlib `dataclasses`, the equivalent is `model_config = ConfigDict(frozen=True)`.

> **Historical note:** The original distributes configuration across six global variables (`$global:MetadataFileParser`, `$global:ExiftoolRejectList`, `$global:MetaSuffixInclude`, `$global:MetaSuffixIncludeString`, `$global:MetaSuffixExclude`, `$global:MetaSuffixExcludeString`) plus inline literals in the function body. The single typed, immutable object consolidates everything. See §4.4 for the full rationale.

Nested configuration types

Two nested types provide structure within `IndexerConfig`:

```
@dataclass(frozen=True)
class MetadataTypeAttributes:
    """Behavioral attributes for a single sidecar metadata type."""
    about: str
    expect_json: bool
    expect_text: bool
    expect_binary: bool
    parent_can_be_file: bool
    parent_can_be_directory: bool
```

This is the Python equivalent of the legacy `$MetadataFileParser.Attributes.<TypeName>` sub-objects (e.g., `Description`, `GenericMetadata`, `Hash`, etc.). The field names are converted from PascalCase to snake_case per Python convention.

```
@dataclass(frozen=True)
class ExiftoolConfig:
    """Exiftool-specific configuration."""
    exclude_extensions: frozenset[str]
    base_args: tuple[str, ...]
```

Implementations MAY flatten these nested types into top-level `IndexerConfig` fields if the nesting adds no practical value for their codebase. The specification uses the nested form because it mirrors the logical grouping of the legacy `MetadataFileParser` and produces cleaner TOML sections (§7.6).

Configuration construction flow

The `load_config()` function in `config/loader.py` is the sole factory for `IndexerConfig` objects. It is called once per invocation during Stage 1:

1. Start with compiled defaults from `config/defaults.py` (§7.2).
2. If a user config file exists at a resolved path (§3.3), read and parse it as TOML via `tomllib.loads()`.
3. If a project-local config file exists (`.shruggie-indexer.toml` in the target directory or its ancestors), read and parse it.
4. Apply CLI argument overrides or API keyword argument overrides.
5. Apply parameter implications (e.g., `rename=True` → `output_inplace=True`; `meta_merge_delete=True` → `meta_merge=True` → `extract_exif=True`).
6. Validate the fully-resolved configuration (§7.1, Validation rules).
7. Compile regex patterns from string form into `re.Pattern` objects.
8. Freeze and return the `IndexerConfig`.

Steps 2–4 follow the layered precedence defined in §3.3. Steps 5–6 are described in detail below.

Parameter implications

Certain flag combinations carry implicit dependencies. The configuration loader enforces these implications after merging all layers:

If this is `True`...	...then force this to `True`	Legacy equivalent
`rename`	`output_inplace`	`$Rename` → `$OutFileInPlace = $true` (line ~9360)
`meta_merge_delete`	`meta_merge`	`$MetaMergeDelete` → `$MetaMerge = $true` (line ~9447)
`meta_merge`	`extract_exif`	`$MetaMerge` → `$Meta = $true` (line ~9450)

These implications are applied in reverse dependency order — `meta_merge_delete` first, then `meta_merge`, then `rename` — so that transitive chains propagate correctly. For example, `meta_merge_delete=True` with all other flags at their defaults produces `meta_merge=True, extract_exif=True, meta_merge_delete=True`.

Output mode defaulting

The output mode resolution logic applies a single rule:

- If neither `output_file` nor `output_inplace` is specified, and `output_stdout` was not explicitly set to `False`, then `output_stdout` defaults to `True`.
- If `output_file` or `output_inplace` is specified, `output_stdout` defaults to `False` unless the user explicitly passes `--stdout`.

The `NoStandardOutput` / `StandardOutput` dual-flag pattern (a positive and negative flag for the same boolean) from the legacy interface is eliminated. A single `output_stdout` field is used. The CLI provides only `--stdout` and `--no-stdout`; absence of either flag triggers the defaulting logic above.

> **Historical note:** The original's output mode resolution is a 90-line block with nested conditionals, redundant inverse-variable tracking (`$NoStandardOutput = !$StandardOutput`), and commented-out debug logging. The current rule is two sentences. The behavioral outcome is identical.

Validation rules

After implication propagation and output mode defaulting, the configuration loader validates the following invariants. Violations are fatal errors raised before any processing begins:

1. **MetaMergeDelete safety.** If `meta_merge_delete` is `True`, at least one of `output_file` or `output_inplace` MUST also be `True`. This prevents the scenario where sidecar files are deleted without their content being captured in any persistent output.

   > **Historical note:** The original enforces this via the `$MMDSafe` variable (lines ~9354–9427); the current design enforces it as a declarative validation rule.

2. **IdType validity.** `id_algorithm` MUST be either `"md5"` or `"sha256"`. Validation occurs during configuration construction.

> **Historical note:** The original validates this with a `switch` statement that falls through to an error (line ~8776).

3. **Path conflicts.** If `output_file` is specified, it MUST NOT point to a path inside the target directory when `output_inplace` is also active. (Writing the aggregate output file into the same directory tree being indexed with in-place writes active would cause the aggregate file to be indexed on subsequent runs.)

4. **Regex compilation.** All regex strings in the metadata identification and exclusion pattern lists MUST compile without error. If a user-provided pattern is invalid, the loader raises a `ConfigurationError` with the offending pattern and the `re.error` message.

7.2. Default Configuration

The compiled defaults in `config/defaults.py` define the baseline configuration that applies when no user configuration file is present. These defaults reproduce the behavioral intent of the original's hardcoded values while extending them for cross-platform coverage (DEV-10) and correcting known issues.

The tool MUST operate correctly using only compiled defaults — no configuration file is required (§3.3).

Scalar defaults

Field	Default	Rationale
`recursive`	`True`	Matches original: recursion is on unless `-NotRecursive` is specified.
`id_algorithm`	`"md5"`	Deviates from original `$IdType` based on a majority of historical user preference.
`compute_sha512`	`False`	SHA512 is available but not computed by default for performance. The original does not compute SHA512 at runtime either (despite declaring it in the output schema).
`output_stdout`	`True` (conditional)	Applied via the defaulting logic in §7.1 — only `True` when no file-based output is active.
`output_file`	`None`	No aggregate output file by default.
`output_inplace`	`False`	No in-place sidecar writes by default.
`extract_exif`	`False`	Matches original: `-Meta` defaults to `$false`.
`meta_merge`	`False`	Matches original: `-MetaMerge` defaults to `$false`.
`meta_merge_delete`	`False`	Matches original: `-MetaMergeDelete` defaults to `$false`.
`rename`	`False`	Matches original: `-Rename` defaults to `$false`.
`dry_run`	`False`	New field (not in original).

Extension validation pattern

Original pattern (from MakeObject, line ~8710):

```
EXTENSION_VALIDATION_PATTERN_ORIGINAL = r"^(([a-z0-9]){1,2}|([a-z0-9])([a-z0-9\-]){1,12}([a-z0-9]))$"
```

This is the original's extension validation regex, transcribed exactly. It accepts lowercase alphanumeric extensions of 1–14 characters where hyphens are permitted in interior positions but not at the start or end. The regex is applied to the extension string *without* the leading dot (e.g., `"mp4"`, not `".mp4"`).

Known edge case: The original implementation treats dotfiles (Unix hidden files like `.gitignore`, `.bashrc` that begin with a dot and contain no other dots) as having an extension equal to the entire name after the dot. When `MakeObject` encounters `.gitignore`, it extracts `gitignore` as the extension and validates it against the pattern above. This is a semantic error — by convention, dotfiles have no file extension.

Improved validation (with dotfile protection):

The tool addresses this edge case with a two-stage validation strategy:

1. **Dotfile detection (prerequisite check):** Before applying the extension validation regex, determine whether the file is a dotfile. A filename is classified as a dotfile if:

   - Its basename (after stripping directory path) begins with a dot
   - The basename contains **no additional dots** beyond the initial one

   Files matching this pattern are dotfiles. They have **no extension** and MUST NOT be subjected to extension validation.

2. **Extension pattern validation:** For files that are NOT dotfiles and have an extracted extension, apply the regex:

```
EXTENSION_VALIDATION_PATTERN = r"^(([a-z0-9]){1,2}|([a-z0-9])([a-z0-9\-]){1,12}([a-z0-9]))$"
```

The regex itself is unchanged from the original, ensuring backward compatibility for all existing valid extensions. The improvement lies in the guarding logic that prevents incorrect application to dotfiles.

Validation examples:

Filename	Dotfile?	Extracted Extension	Validation Result
`video.mp4`	No	`mp4`	✔ Validates against regex
`.gitignore`	Yes	*(none)*	✔ Regex not applied (dotfile)
`.bashrc`	Yes	*(none)*	✔ Regex not applied (dotfile)
`.vimrc`	Yes	*(none)*	✔ Regex not applied (dotfile)
`.config.json`	No	`json`	✔ Validates against regex
`archive.tar.gz`	No	`gz`	✔ Validates against regex
`main.js`	No	`js`	✔ Validates against regex
`.DS_Store`	Yes	*(none)*	✔ Regex not applied (dotfile)

Implementation note: Python's `os.path.splitext()` function handles this correctly — `os.path.splitext(".gitignore")` returns `(".gitignore", "")`, treating the file as having no extension. The implementation in `core/filesystem.py` (§6.2) leverages this behavior and adds an explicit dotfile check to ensure the extension validation step is skipped entirely for these files.

This pattern remains configurable (DEV-14) — users who encounter legitimate extensions rejected by the pattern can relax it in their config file. The default preserves the legacy validation logic while fixing the dotfile semantic error.

Filesystem exclusion defaults

The default filesystem exclusion set extends the legacy Windows-only pair to cover all three target platforms:

```
FILESYSTEM_EXCLUDES = frozenset({
    # Windows
    "$recycle.bin",
    "system volume information",
    "desktop.ini",
    "thumbs.db",
    # macOS
    ".ds_store",
    ".spotlight-v100",
    ".trashes",
    ".fseventsd",
    ".temporaryitems",
    ".documentrevisions-v100",
    # Version control (optional, included by default)
    ".git",
})

FILESYSTEM_EXCLUDE_GLOBS = (
    # Linux
    ".trash-*",
)
```

All exclusion names are stored in lowercase. Matching is performed case-insensitively (§6.1). Glob patterns are stored separately from exact-match names because they require `fnmatch` matching rather than set membership lookup. The original only excludes `$RECYCLE.BIN` and `System Volume Information` (DEV-10).

Exiftool exclusion defaults

```
EXIFTOOL_EXCLUDE_EXTENSIONS = frozenset({
    "csv", "htm", "html", "json", "tsv", "xml",
})
```

This matches the original's `$MetadataFileParser.Exiftool.Exclude` exactly. See §7.4 for the full rationale.

Exiftool argument defaults

```
EXIFTOOL_BASE_ARGS = (
    "-extractEmbedded3",
    "-scanForXMP",
    "-unknown2",
    "-json",
    "-G3:1",
    "-struct",
    "-ignoreMinorErrors",
    "-charset", "filename=utf8",
    "-api", "requestall=3",
    "-api", "largefilesupport=1",
    "--",
)
```

These are the legacy exiftool arguments, decoded from the Base64-encoded `$ArgsQ` string (the quiet-mode variant — the tool controls verbosity through its own logging system, not through exiftool's `-quiet` flag). The arguments are stored as a plain Python tuple rather than being Base64-encoded (DEV-05).

> **Historical note:** The original Base64-encodes the exiftool argument strings (`$ArgsV`, `$ArgsQ`) and decodes them at runtime via `Base64DecodeString`, writing the result to a temporary file that is then passed to exiftool via `-@ "$TempArgsFile"`. This three-step pipeline (encode → decode → write to file → pass as arg file) is entirely unnecessary. The arguments are stored as a plain tuple and passed directly to `subprocess.run()` as a list. The `-b` flag (binary output) present in the legacy verbose argument set is intentionally omitted from the default — it causes exiftool to emit binary-encoded values in the JSON output, which complicates downstream parsing. If binary output is needed for specific use cases, it can be added via configuration.

Metadata identification pattern defaults

The complete default pattern set is specified in §7.5. The defaults in `config/defaults.py` define these patterns as lists of regex strings. The `config/loader.py` compiles them into `re.Pattern` objects during configuration construction.

Metadata type attribute defaults

The complete default attribute set is specified in §7.3. The defaults in `config/defaults.py` define these as `MetadataTypeAttributes` instances.

Extension group defaults

The extension groups from the original's `$MetadataFileParser.ExtensionGroups` are carried forward verbatim. These groups classify common file extensions into categories (Archive, Audio, Font, Image, Link, Subtitles, Video) and are used during sidecar metadata parsing to infer expected content types. The complete lists are specified in §7.3.

7.3. Metadata File Parser Configuration

This subsection specifies the complete mapping of the legacy `$global:MetadataFileParser` ordered hashtable into the typed configuration system. The structure contains four top-level sub-objects — `Attributes`, `Identify`, `Exiftool`, and `ExtensionGroups` — plus a derived `Indexer` sub-object. Each is mapped to a corresponding field or group of fields in `IndexerConfig`.

The isolated reference script `MakeIndex(MetadataFileParser).ps1` (§1.5) is the authoritative source for the legacy patterns. This section reproduces the complete content with mapping guidance.

Attributes → `metadata_attributes`

The `Attributes` sub-object defines behavioral metadata for each sidecar type — what data formats to expect, whether the sidecar can be associated with a file parent, a directory parent, or both. These attributes guide the parsing strategy in `core/sidecar.py` (§6.7).

Type Name	about	expect_json	expect_text	expect_binary	parent_can_be_file	parent_can_be_directory
description	Likely a youtube-dl or yt-dlp information file containing UTF-8 text (with possible problematic characters).	True	True	False	True	False

Type Name	about	expect_json	expect_text	expect_binary	parent_can_be_file	parent_can_be_directory
desktop_ini	A Windows desktop.ini file used to customize folder appearance in Windows Explorer.	False	True	True	False	True
generic_metadata	Generic metadata file which may contain any type of metadata information related to files or directories.	True	True	True	True	True
hash	A file containing a hash value (MD5, SHA1, SHA256, etc.) of another file.	False	True	False	True	False
json_metadata	A JSON file containing metadata information related to files or directories.	True	False	False	True	True
link	A file containing an Internet URL or a link to another file or directory.	False	True	True	True	True
screenshot	A screenshot image file which may contain a screen capture of a computer desktop or application.	False	False	True	True	False
subtitles	A subtitle file which contains text-based subtitles for a video or audio file.	True	True	True	True	False
thumbnail	A thumbnail image file containing one or more reduced-size icon images related to another file or directory.	False	False	True	True	True
torrent	A torrent or magnet link file containing connection and/or identification information for peer-to-peer retrieval.	False	False	True	True	True

The type names are converted from PascalCase (`DesktopIni`) to snake_case (`desktop_ini`) per Python convention. The `about` strings are preserved from the original for documentation purposes.

Identify → `metadata_identify`

The `Identify` sub-object contains the regex patterns used to classify a filename as a sidecar metadata file and determine its type. Each type maps to an ordered list of regex patterns. A filename matches a type if it matches any pattern in that type's list. Patterns within a list are ordered from most specific to most generic (where applicable — notably the Subtitles patterns).

These patterns are the most critical configuration data to map correctly. The regex syntax used by the legacy implementation is PowerShell's `-match` operator, which uses .NET regular expressions. The patterns in `MetadataFileParser.Identify` use a subset of .NET regex that is fully compatible with Python's `re` module — specifically: character classes, alternation, anchors (`^`, `$`), quantifiers, and grouping. No .NET-specific regex features (lookbehind with variable length, named balancing groups) are used. The patterns can therefore be transcribed to Python without syntactic modification.

Compilation rule: Each pattern string is compiled in Python via `re.compile(pattern, re.IGNORECASE)`. The `re.IGNORECASE` flag is applied because PowerShell's `-match` operator performs case-insensitive matching by default. All patterns are applied using `re.search()` against the full filename (not just the extension), matching the legacy behavior where patterns may anchor to the start or end of the filename.

The complete pattern inventory, transcribed from `MakeIndex(MetadataFileParser).ps1`:

Description:

```
(r'\.description$',)
```

DesktopIni:

```
(r'\.desktop\.ini$', r'desktop\.ini$')
```

GenericMetadata:

```
(
    r'\.(exif|meta|metadata)$',
    r'\.comments$',
    r'^.(git(attributes|ignore))$',
    r'\.(cfg|conf|config)$',
    r'\.yaml$',
)
```

Hash:

```
(r'\.(md5|sha\d+|blake2[bs]|crc\d+|xxhash|checksum|hash)$',)
```

JsonMetadata:

```
(
    r'_directorymeta2?\.json$',
    r'_(subs|subtitles)\.json$',
    # BCP 47 language-code subtitles in JSON format
    r'\.(aa|af|sq|gsw-fr|ase|am|ar|arq|abv|arz|acm|ajp|afb-kw|apc|ayl|ary|acx|'
    r'afb-qa|ar-sa|ar-sy|aeb|ar-ae|ar-ye|arp|hy|as|az|az-cyrl|az-latn|ba|be|bn|'
    r'bn-in|bs|bs-cyrl|bzs|br|br-fr|bg|my|ca|tzm|tzm-arab-ma|tzm-dz|tzm-tfng|'
    r'tzm-tfng-ma|ckb|ckb-iq|chr|zh|yue|yue-hk|cmn|cmn-hans|cmn-hans-cn|'
    r'cmn-hans-hk|cmn-hans-mo|cmn-hans-my|cmn-hans-sg|cmn-hans-tw|cmn-tw|'
    r'cmn-hant|cmn-hant-cn|cmn-hant-hk|cmn-hant-mo|cmn-hant-my|cmn-hant-sg|'
    r'cmn-hant-tw|nan|zh-hans|zh-hans-cn|zh-hans-hk|zh-hans-mo|zh-hans-my|'
    r'zh-hans-sg|zh-hans-tw|zh-hant|zh-hant-cn|zh-hant-hk|zh-hant-mo|'
    r'zh-hant-my|zh-hant-sg|zh-hant-tw|com|co|co-fr|hr|hr-ba|quz|cs|da|prs|dv|'
    r'nl|dz|bin|en|en-au|en-bz|en-ca|en-029|en-hk|en-in|en-id|en-ie|en-jm|'
    r'en-my|en-nz|en-ph|en-sg|en-za|en-se|en-tt|en-ae|en-gb|en-us|en-zw|et|eu|'
    r'fo|fil|fi|nl-be|fr|fr-be|fr-cm|fr-ca|fr-029|fr-ci|fr-ht|fr-lu|fr-ml|'
    r'fr-mc|fr-ma|fr-re|fr-sn|fr-ch|fr-cd|ff|ff-latn|ff-latn-ng|ff-latn-sn|'
    r'ff-ng|gl|ka|de|de-at|de-li|de-lu|gsw|de-ch|el|gn|gu|ha|ha-latn|ha-latn-ng|'
    r'haw|he|hi|hu|ibb|is|ig|id|iu|iu-cans|ga|it|it-ch|ja|ja-jp|quc|kl|kn|kr|'
    r'kr-ng|ks|ks-deva-in|kk|km|rw|kok|ko|ky|lad|lo|la|la-va|lv|ln|lt|dsb|lb|'
    r'mk|ms-bn|ms-my|ms|ml|mt|mni|mni-beng-in|mi|arn|mr|fit|moh|mn|mn-cn|'
    r'mn-mong|mn-mong-cn|nv|ne|ne-in|no|nb|nn|oc|or|om|pap|pap-029|ps|fa|pl|'
    r'pt-br|pt|pa|pa-arab|qu|qu-bo|qu-ec|qu-pe|ro|ro-md|rm|ru|ru-md|aec|sah|smi|'
    r'smn|smj|smj-no|se|se-fi|se-no|se-se|sms|sma|sma-no|sm|sa|gd|sr|sr-cyrl|'
    r'sr-ba|sr-cyrl-me|sr-latn|sr-latn-ba|sr-me|sd|sd-arab|sd-in|si|sk|sl|so|st|'
    r'nso|es-ar|es-bo|es|es-cl|es-co|es-cr|es-cu|es-do|es-ec|es-sv|es-gt|es-hn|'
```

```
    r'es-419|es-mx|es-ni|es-pa|es-py|es-pe|es-pr|es-us|es-uy|es-ve|sw|sw-ke|sv|'
    r'sv-fi|syr|syr-sy|tl|tg|tg-cyrl|tg-cyrl-tj|ta|tt|te|th|bo|ti|ts|tn|tn-bw|'
    r'tr|tk|uk|und|hsb|ur|ug|uz|ca-es|ve|vi|cy|fy|wo|xh|ii|yi|yo|zu'
    r')(-orig)?\.json$',
    r'_[a-z0-9]{3,19}\.json$',
    r'\.exifjson$',
    r'\.(AI|exif|info|meta)\.json$',
)
```

The BCP 47 language-code alternation is the longest and most carefully crafted pattern in the configuration. It matches subtitle metadata files produced by youtube-dl / yt-dlp, which use the naming convention `<basename>.<language_code>.json`. The alternation covers all language codes recognized by the original author. This pattern MUST be ported exactly — any dropped or modified language code will cause the corresponding subtitle files to be missed during sidecar discovery. The Python string is split across multiple lines for readability using implicit string concatenation; when compiled, it produces a single contiguous pattern identical to the original.

Link:

```
(r'\.(url|lnk|link|source)$',)
```

Screenshot:

```
(r'(-|_)?(screen|screen(s|shot|shots)|thumb|thumb(nail|nails))((-|_)?([0-9]{1,9}))?\.(jpg|jpeg|png|webp)$',)
```

Subtitles:

```
(
    # Pattern 1: Language-tagged subtitle files (most specific)
    r'\.(aa|af|sq|...<same BCP 47 alternation as JsonMetadata>...)(-orig)?\.(srt|sub|sbv|vtt|lrc|txt)$',
    # Pattern 2: Bare subtitle extensions (most generic)
    r'\.(srt|sub|sbv|vtt|lrc)$',
)
```

The Subtitles pattern list reuses the same BCP 47 language-code alternation as the JsonMetadata patterns but with subtitle file extensions (`.srt`, `.sub`, `.sbv`, `.vtt`, `.lrc`, `.txt`) instead of `.json`. Pattern order matters: the more specific language-tagged pattern is listed first so that a file like `video.en.srt` is matched by the specific pattern before the generic `\.srt$` pattern. Both patterns classify the file as `subtitles`, so the ordering affects only the efficiency of matching, not the outcome. Nonetheless, the original's ordering SHOULD be preserved.

Thumbnail:

```
(
    r'\.(cover|thumb|thumb(s|db|index|nail))$',
    r'^(thumb|thumb(s|db|index|nail))\.db$',
)
```

Torrent:

```
(r'\.(torrent|magnet)$',)
```

Indexer include/exclude → `metadata_exclude_patterns` (plus computed inclusion)

The original's `$MetadataFileParser.Indexer` sub-object contains two derived arrays:

- `Exclude`: Regex patterns for files that should be excluded from the index entirely (e.g., existing indexer sidecar output files, thumbnail database files). These files are not indexed as standalone items when encountered during traversal.
- `Include`: A union of all patterns from the `Identify` sub-object, computed at load time by iterating `$MetadataFileParser.Identify.Keys` and collecting all per-type patterns into a flat array.
- `ExcludeString` / `IncludeString`: The `Exclude` and `Include` arrays joined with `|` into single alternation strings.

The `Exclude` patterns are preserved and extended as `metadata_exclude_patterns`:

```
METADATA_EXCLUDE_PATTERNS = (
    re.compile(r'_(meta2?|directorymeta2?)\.json$', re.IGNORECASE),
    re.compile(r'\.(cover|thumb|thumb(s|db|index|nail))$', re.IGNORECASE),
    re.compile(r'^(thumb|thumb(s|db|index|nail))\.db$', re.IGNORECASE),
)
```

v1/v2 sidecar coexistence in the exclusion pattern. The first pattern uses the `2?` quantifier to match both the v1 naming convention (`_meta.json`, `_directorymeta.json`) and the v2 naming convention (`_meta2.json`, `_directorymeta2.json`). This is necessary because the indexer may operate on directory trees that contain legacy v1 sidecar files from previous `MakeIndex` runs, newly generated v2 sidecar files, or both simultaneously during a migration period. Both generations of sidecar output files are indexer artifacts and MUST be excluded from traversal to prevent the indexer from indexing its own output. See §5.13 for the full v1/v2 coexistence rationale and §6.9 for the v2 naming convention used by the serializer.

> **Historical note:** The original pre-joins the `Include` and `Exclude` arrays into single `IncludeString` / `ExcludeString` alternation strings for use with PowerShell's `-match` operator. This is an optimization for PowerShell's regex engine, which compiles the pattern on every `-match` call. In Python, patterns are compiled once via `re.compile()` and reused. The pre-joined alternation strings are unnecessary — individual compiled patterns are stored and iterated instead. This is equally fast for the small number of patterns involved (typically under 30 total) and simpler to maintain, debug, and extend. The `IncludeString` / `ExcludeString` fields are not carried forward.

The `Include` set (the union of all `Identify` patterns) is not stored as a separate configuration field. Instead, the determination of whether a filename is a sidecar metadata file is performed by iterating `metadata_identify` and checking for any match — which is logically equivalent to checking against the union. If a fast "is this a sidecar at all?" check is needed as a hot-path optimization, the loader MAY compute a combined alternation pattern at construction time and expose it as a read-only property on `IndexerConfig`.

ExtensionGroups → `extension_groups`

The `ExtensionGroups` sub-object classifies common file extensions into seven categories. These groups are used by the sidecar parser (§6.7) to infer expected content types and validate parent-child relationships (e.g., a `subtitles` sidecar should be a sibling of a Video or Audio file, not an Image file). The complete lists, carried forward verbatim from the legacy configuration:

Group	Extensions
`archive`	`7z`, `ace`, `alz`, `arc`, `arj`, `bz`, `bz2`, `cab`, `cbr`, `cbz`, `chm`, `cpio`, `deb`, `dmg`, `egg`, `gz`, `hdd`, `img`, `iso`, `jar`, `lha`, `lz`, `lz4`, `lzh`, `lzma`, `lzo`, `qcow2`, `rar`, `rpm`, `s7z`, `shar`, `sit`, `sitx`, `sqx`, `tar`, `tbz`, `tbz2`, `tgz`, `tlz`, `txz`, `vdi`, `vhd`, `vhdx`, `vmdk`, `war`, `wim`, `xar`, `xz`, `z`, `zip`, `zpaq`, `zst`
`audio`	`3gp`, `8svx`, `aa`, `aac`, `aax`, `act`, `aif`, `aiff`, `amr`, `ape`, `au`, `awb`, `cda`, `dct`, `dss`, `dvf`, `flac`, `gsm`, `iklax`, `ivs`, `m4a`, `m4b`, `m4p`, `mka`, `mlp`, `mmf`, `mp2`, `mp3`, `mpc`, `msv`, `ogg`, `oga`, `opus`, `ra`, `rm`, `raw`, `sln`, `tta`, `voc`, `vox`, `wav`, `wma`, `wv`, `webm`, `wv`, `wvp`, `wvpk`
`font`	`eot`, `otf`, `svg`, `svgz`, `ttc`, `ttf`, `woff`, `woff2`
`image`	`3fr`, `ari`, `arw`, `bay`, `bmp`, `cr2`, `crw`, `dcr`, `dng`, `erf`, `fff`, `gif`, `gpr`, `icns`, `ico`, `iiq`, `jng`, `jp2`, `jpeg`, `jpg`, `k25`, `kdc`, `mef`, `mos`, `mrw`, `nef`, `nrw`, `orf`, `pbm`, `pef`, `pgm`, `png`, `ppm`, `psd`, `ptx`, `raf`, `raw`, `rw2`, `rwl`, `sr2`, `srf`, `svg`, `tga`, `tif`, `tiff`, `webp`, `x3f`
`link`	`link`, `lnk`, `shortcut`, `source`, `symlink`, `url`
`subtitles`	`srt`, `sub`, `sbv`, `vtt`, `lrc`
`video`	`3g2`, `3gp`, `3gp2`, `3gpp`, `amv`, `asf`, `avi`, `divx`, `drc`, `dv`, `f4v`, `flv`, `gvi`, `gxf`, `ismv`, `m1v`, `m2v`, `m2t`, `m2ts`, `m4v`, `mkv`, `mov`, `mp2`, `mp2v`, `mp4`, `mp4v`, `mpe`, `mpeg`, `mpeg1`, `mpeg2`, `mpeg4`, `mpg`, `mpv2`, `mts`, `mtv`, `mxf`, `nsv`, `nuv`, `ogm`, `ogv`, `ogx`, `ps`, `rec`, `rm`, `rmvb`, `tod`, `ts`, `tts`, `vob`, `vro`, `webm`, `wm`, `wmv`, `wtv`, `xesc`

All extensions are stored in lowercase without a leading dot. Extension groups are stored as a `MappingProxyType[str, tuple[str, ...]]` in `IndexerConfig`.

Note on duplicates: The legacy `audio` list contains `wv` twice. The loader deduplicates this during loading — `frozenset` or `tuple(sorted(set(...)))` naturally eliminates duplicates.

Note on overlapping extensions: Some extensions appear in multiple groups (e.g., `3gp` in both Audio and Video, `svg` in both Font and Image, `mp2` in both Audio and Video, `rm` in both Audio and Video, `webm` in both Audio and Video, `raw` in both Audio and Image). This is intentional — a `.3gp` file may contain audio, video, or both. The extension groups describe what a file *could* be, not a mutually exclusive classification. The sidecar parser uses these groups as heuristics, not definitive type assignments.

7.4. Exiftool Exclusion Lists

The `exiftool_exclude_extensions` configuration field specifies file extensions for which exiftool metadata extraction is skipped entirely. When a file's extension (lowercase, without the leading dot) appears in this set, `core/exif.extract_exif()` returns `None` without invoking the exiftool subprocess (§6.6).

Default exclusion set

```
EXIFTOOL_EXCLUDE_EXTENSIONS = frozenset({
    "csv", "htm", "html", "json", "tsv", "xml",
})
```

Rationale

These file types cause exiftool to emit the file's text content as metadata rather than extracting meaningful embedded metadata. The original documents this problem explicitly (line ~7717):

> *"If included, these types of files can result in (essentially) total inclusion of the file content inside the exiftool output (which is rediculous and defeats the purpose of trying to get simple metadata on a file in the first place). The exiftool authors have rejected numerous user requests for more limited metadata outputs from these file types, so we have to manually reject processing them entirely."*

The practical consequence of including these file types is twofold: the metadata output becomes massive (potentially gigabytes for large CSV or XML files), and the serialization layer may fail under the resulting memory pressure (the original's `ConvertTo-Json` has a known memory ceiling; Python's `json.dumps()` is more robust but the output is still uselessly large).

Extension mechanism

Users can extend the exclusion list via configuration (§7.6, §7.7). For example, a user working with large `.log` or `.txt` files that produce similarly excessive exiftool output can add those extensions to their config:

```
[exiftool]
exclude_extensions_append = ["log", "txt"]
```

The `_append` suffix triggers additive merge behavior (§7.7) rather than replacement.

Key exclusion configuration

The metadata key exclusion set (§6.6, "Output parsing and key filtering") is configurable via two TOML keys under the `[exiftool]` table, following the same replace/append convention as extension exclusions:

Config key	Type	Behavior
`exclude_keys`	list of strings	**Replace** — the specified list becomes the complete key exclusion set, overriding all compiled defaults.
`exclude_keys_append`	list of strings	**Append** — entries are added to the compiled default exclusion set (24 keys).

Values are base key names (e.g., `"Copyright"`, `"Artist"`), matched after stripping any group prefix. When no key exclusion configuration is present, the compiled default set applies unchanged.

```
[exiftool]
# Add extra keys to exclude (keeps all defaults)
exclude_keys_append = ["Copyright", "Artist"]

# Or replace the entire exclusion set (advanced):
# exclude_keys = ["SourceFile", "Directory", "FileName"]
```

7.5. Sidecar Suffix Patterns and Type Identification

This subsection consolidates the porting guidance for the sidecar metadata identification system — the regex patterns in `metadata_identify` that determine whether a given filename is a sidecar metadata file and, if so, what type it is.

How sidecar identification works

During sidecar discovery (§6.7), for each item being indexed, the sidecar module examines the item's sibling files. For each sibling, it tests the sibling's filename against every pattern in `metadata_identify`, iterating types in definition order. The first matching type wins — the sibling is classified as that type and no further patterns are tested.

This means type ordering in the configuration is significant when patterns could overlap. The definition order should place more specific types before more generic ones. The default ordering matches the original: Description, DesktopIni, GenericMetadata, Hash, JsonMetadata, Link, Screenshot, Subtitles, Thumbnail, Torrent.

Indexer exclusion during traversal

Independently from sidecar identification, the traversal module (§6.1) and entry construction module (§6.8) use `metadata_exclude_patterns` to skip certain files entirely. These are files that are themselves indexer output artifacts — both v1 legacy sidecars (`_meta.json`, `_directorymeta.json`) and v2 sidecars (`_meta2.json`, `_directorymeta2.json`) — or thumbnail databases that are not meaningful to index. When MetaMerge is active, files matching any `metadata_identify` pattern for the current item's basename are also excluded from the regular item list (they are processed as sidecars rather than indexed as standalone items).

The distinction is important: `metadata_identify` determines *type classification* of sidecars. `metadata_exclude_patterns` determines which files to *skip entirely* during traversal. The two systems are complementary but serve different purposes.

Pattern compilation and caching

All regex strings from the configuration are compiled into `re.Pattern` objects once during `load_config()` (Stage 1). The compiled patterns are stored in the `IndexerConfig` and reused for every filename test. Pattern compilation is a measurable cost for the BCP 47 language-code alternations (which produce large NFA state machines), but it happens exactly once and the compiled pattern is then O(n) in the input string length for each subsequent match.

The implementation SHOULD pre-compile all patterns using `re.compile(pattern, re.IGNORECASE)`. The `re.IGNORECASE` flag ensures that filename matching is case-insensitive across all platforms, matching the original's behavior.

Validation of user-provided patterns

If a user adds custom patterns via configuration (§7.6), the configuration loader MUST validate that each pattern compiles without error. Invalid patterns produce a fatal `ConfigurationError` with the offending pattern and the `re.error` message. The indexing operation does not proceed with invalid patterns — a user typo in a regex MUST NOT cause silent misidentification of sidecar files.

7.6. Configuration File Format

The configuration file format is TOML, parsed by Python's `tomllib` module (standard library, Python 3.11+). TOML is chosen over JSON (no comments, verbose for nested structures), YAML (whitespace-sensitive, multiple dialects, requires a third-party parser), and INI (no nested structures, no typed values) because it provides a reasonable balance of human readability, type preservation, and stdlib support.

File naming and location

The configuration file name is `config.toml`. The resolution paths are defined in §3.3:

Priority	Location	Description
1 (lowest)	Compiled defaults	`config/defaults.py`. Always present.
2	User config directory	`~/.config/shruggie-indexer/config.toml` (Linux/macOS) or `%APPDATA%\shruggie-indexer\config.toml` (Windows).
3	Project-local config	`.shruggie-indexer.toml` in the target directory or its ancestors (searched upward).
4 (highest)	CLI/API arguments	Command-line flags or `index_path()` keyword arguments.

TOML structure

The TOML file mirrors the `IndexerConfig` field structure. Top-level scalar fields are TOML key-value pairs. Nested structures become TOML tables. Collection fields (extension lists, regex patterns) become TOML arrays.

A complete example showing every configurable field:

```toml
# shruggie-indexer configuration

# Traversal and identity
recursive = true
id_algorithm = "md5"          # "md5" or "sha256"
compute_sha512 = false

# Output routing (typically set via CLI, not config file)
# output_stdout = true
# output_file = ""
# output_inplace = false

# Metadata processing
extract_exif = false
meta_merge = false
meta_merge_delete = false

# Rename
```

```
rename = false
dry_run = false

# Extension validation regex (applied without the leading dot)
extension_validation_pattern = '^(([a-z0-9]){1,2}|([a-z0-9])([a-z0-9\-]){1,12}([a-z0-9]))$'

[filesystem_excludes]
# Exact-match names (case-insensitive)
names = [
    "$recycle.bin",
    "system volume information",
    "desktop.ini",
    "thumbs.db",
    ".ds_store",
    ".spotlight-v100",
    ".trashes",
    ".fseventsd",
    ".temporaryitems",
    ".documentrevisions-v100",
    ".git",
]
# Glob patterns (fnmatch syntax)
globs = [".trash-*"]

[exiftool]
exclude_extensions = ["csv", "htm", "html", "json", "tsv", "xml"]

# Key exclusion (base key names, matching with or without group prefix)
# exclude_keys = ["SourceFile", "Directory"]        # Replace mode (advanced)
# exclude_keys_append = ["Copyright", "Artist"]      # Append to defaults

base_args = [
    "-extractEmbedded3",
    "-scanForXMP",
    "-unknown2",
    "-json",
    "-G3:1",
    "-struct",
    "-ignoreMinorErrors",
    "-charset", "filename=utf8",
    "-api", "requestall=3",
    "-api", "largefilesupport=1",
    "--",
]

[metadata_identify]
# Each key is a type name; each value is a list of regex pattern strings.
# Patterns are compiled with re.IGNORECASE at load time.
description = ['\.description$']
desktop_ini = ['\.desktop\.ini$', 'desktop\.ini$']
# ... (remaining types follow the same pattern)

[metadata_exclude]
patterns = [
    '_(meta2?|directorymeta2?)\.json$',
    '\.(cover|thumb|thumb(s|db|index|nail))$',
    '^(thumb|thumb(s|db|index|nail))\.db$',
]

[extension_groups]
archive = ["7z", "ace", "alz", "arc", "arj", "bz", "bz2", "..."]
audio = ["3gp", "8svx", "aa", "aac", "aax", "..."]
# ... (remaining groups)
```

Most users will not need a configuration file at all — the defaults cover the common case. Users who do need customization will typically modify only one or two sections (most commonly `filesystem_excludes` to add project-specific directories, or `exiftool.exclude_extensions` to add problematic file types).

Type mapping

TOML values map to Python types as follows in the configuration loader:

TOML type	Python type in `IndexerConfig`
String	`str`

TOML type	Python type in `IndexerConfig`
Boolean	`bool`
Array of strings	`tuple[str, ...]` or `frozenset[str]` (depending on whether order matters)
Table	Nested dataclass or `dict`
Integer	`int` (not currently used but supported)

The configuration loader validates types during parsing. A TOML value of the wrong type for its field produces a `ConfigurationError`. For example, `id_algorithm = 42` (integer instead of string) is a validation error.

7.7. Configuration Override and Merging Behavior

When multiple configuration layers are present (compiled defaults, user config file, project-local config file, CLI arguments), the layers must be merged into a single `IndexerConfig`. The merging strategy differs by field type.

Scalar fields: last-writer-wins

For scalar fields (`bool`, `str`, `Path | None`, `int`), the highest-priority layer that specifies a value wins. Lower-priority values are completely replaced.

Example: If the compiled default for `id_algorithm` is `"md5"` and the user config file specifies `id_algorithm = "sha256"`, the resolved value is `"sha256"`. If the CLI then passes `--id-type md5`, the resolved value reverts to `"md5"`.

Collection fields: replace by default, append on request

For collection fields (extension sets, exclusion lists, regex pattern lists), the default merge behavior is **replace** — the highest-priority layer that specifies a value completely replaces the lower-priority value. This matches the principle of least surprise: if a user specifies `exclude_extensions = ["csv", "xml"]`, they get exactly those two extensions, not a union with the compiled defaults.

However, the more common use case is **additive**: the user wants to keep the defaults and add a few extra entries. To support this, the configuration file recognizes `_append` suffixed field names:

Config file key	Behavior
`exclude_extensions = [...]`	**Replace**: the specified list becomes the complete exclusion set.
`exclude_extensions_append = [...]`	**Append**: the specified entries are added to the existing set (from the lower-priority layer).

Both forms may be present in the same file (though this is unusual). When both appear, `_append` is applied after the replacement.

This pattern applies to all collection fields:

Base field	Append variant
`filesystem_excludes.names`	`filesystem_excludes.names_append`
`filesystem_excludes.globs`	`filesystem_excludes.globs_append`
`exiftool.exclude_extensions`	`exiftool.exclude_extensions_append`
`exiftool.exclude_keys`	`exiftool.exclude_keys_append`
`exiftool.base_args`	`exiftool.base_args_append`
`metadata_exclude.patterns`	`metadata_exclude.patterns_append`
`metadata_identify.<type>`	`metadata_identify.<type>_append`
`extension_groups.<group>`	`extension_groups.<group>_append`

> **Historical note:** The original provides no mechanism for user customization of configuration — all values are hardcoded in the script source. The layered merge system with both replace and append semantics gives users full control without requiring source code modification, fulfilling design goal G4.

CLI/API overrides

CLI arguments and API keyword arguments are the highest-priority layer. They override all file-based configuration. Because CLI arguments are typically scalar (flags and options), not collection-valued, the replace-vs-append distinction rarely applies. The one exception is `--exiftool-exclude-ext`, which accepts a repeatable option: each occurrence appends to the set rather than replacing it.

The configuration loader processes layers in priority order (lowest to highest), building up the resolved configuration incrementally:

```
def load_config(
    *,
    cli_overrides: dict[str, Any] | None = None,
    target_directory: Path | None = None,
) -> IndexerConfig:
    config_dict = get_defaults_dict()                 # Layer 1
    merge_toml(config_dict, find_user_config())       # Layer 2
    if target_directory:
        merge_toml(config_dict, find_project_config(target_directory))  # Layer 3
    if cli_overrides:
        merge_overrides(config_dict, cli_overrides)   # Layer 4
    apply_implications(config_dict)                   # §7.1 implications
    validate(config_dict)                             # §7.1 validation
    return build_config(config_dict)                  # compile regexes, freeze
```

The `merge_toml()` function implements the replace/append semantics described above. The `merge_overrides()` function applies CLI/API values as simple key replacements (no append semantics). The `build_config()` function compiles regex strings, freezes collections, and constructs the final `IndexerConfig` instance.

Absent fields in configuration files

If a configuration file omits a field entirely, the value from the lower-priority layer (ultimately the compiled default) is preserved. Configuration files are sparse by design — a file containing only `id_algorithm = "md5"` changes only that one field and leaves all other fields at their defaults.

Unknown fields

If a configuration file contains a key that does not correspond to any `IndexerConfig` field, the loader logs a warning and ignores the key. Unknown keys are not fatal errors — this accommodates forward compatibility when new configuration fields are added in future versions.

8. CLI Interface

This section defines the command-line interface for `shruggie-indexer` — the command structure, every option and argument, their interactions, validation rules, output scenarios, and exit codes. The CLI is a thin presentation layer over the library API (§9) and the core indexing engine (§6). No indexing logic lives in the CLI module — it is responsible only for parsing user input, constructing an `IndexerConfig` object (§7), calling `index_path()`, and routing output to the requested destinations. This separation is design goal G3 (§2.3).

The CLI module is `cli/main.py` (§3.2). The entry point is registered as `shruggie-indexer = "shruggie_indexer.cli.main:main"` in `pyproject.toml` (§2.1), and `python -m shruggie_indexer` invokes the same function via `__main__.py`.

CLI framework: The CLI uses `click` as its argument parsing framework. `click` is a required runtime dependency declared in `[project.dependencies]` in `pyproject.toml` (§12.3). A resilience-only import guard in `__main__.py` catches `ImportError` and produces a clear message directing the user to reinstall (`pip install shruggie-indexer`) — this path is not expected in normal operation. The core library modules do not import `click` — it is consumed exclusively by `cli/main.py`.

> **Historical note:** The original's CLI is the `Param()` block of a PowerShell function — 14 parameters with aliases, switches, and manual validation logic spanning ~200 lines. `click` decorators replace this with declarative type conversion, mutual exclusion, default propagation, and help text generation. The tool also adds capabilities absent from the original: `--version`, `--dry-run`, `--config`, `--no-stdout`, structured exit codes, and graceful interrupt handling (§8.11).

8.1. Command Structure

`shruggie-indexer` exposes a single top-level command with no subcommands. All behavior is controlled via options and a single positional argument.

```
shruggie-indexer [OPTIONS] [TARGET]
```

The command name when installed via `pip install -e .` is `shruggie-indexer` (hyphenated). The `python -m shruggie_indexer` invocation behaves identically.

Help output

The `--help` flag produces a usage summary organized into logical option groups. The following is the canonical help layout — the implementation MUST produce output equivalent to this structure, though minor whitespace or wrapping differences are acceptable:

```
Usage: shruggie-indexer [OPTIONS] [TARGET]

  Index files and directories, producing structured JSON output with hash-based
  identities, filesystem metadata, EXIF data, and sidecar metadata.
```

```
Arguments:
  [TARGET]  Path to the file or directory to index. Defaults to the current
            working directory if not specified.

Target Options:
  --file / --directory    Force TARGET to be treated as a file or directory.
                          Normally inferred from the filesystem. Useful for
                          disambiguating symlinks.
  --recursive / --no-recursive
                          Enable or disable recursive traversal for directory
                          targets. Default: recursive.

Output Options:
  --stdout / --no-stdout  Write JSON output to stdout. Default: enabled when no
                          other output is specified; disabled when --outfile or
                          --inplace is used.
  --outfile, -o PATH      Write combined JSON output to the specified file.
  --inplace               Write individual sidecar JSON files alongside each
                          processed item.

Metadata Options:
  --meta, -m              Extract embedded metadata via exiftool.
  --meta-merge            Merge sidecar metadata into parent entries. Implies
                          --meta.
  --meta-merge-delete     Merge and delete sidecar files. Implies --meta-merge.
                          Requires --outfile or --inplace.

Rename:
  --rename                Rename files to their storage_name. Implies --inplace.
  --dry-run               Preview rename operations without executing them.

Identity:
  --id-type [md5|sha256]  Hash algorithm for the id field. Default: md5.
  --compute-sha512        Include SHA-512 in hash output. Default: disabled.

Configuration:
  --config PATH           Path to a TOML configuration file. Overrides the
                          default file resolution.

Logging:
  -v, --verbose           Increase verbosity. Repeat for more detail (-vv, -vvv).
  -q, --quiet             Suppress all non-error output.

General:
  --version               Show version and exit.
  --help                  Show this message and exit.
```

8.2. Target Input Options

Positional argument: `TARGET`

The `TARGET` argument specifies the file or directory to index. It is optional — when omitted, the current working directory is used as the target.

```
@click.argument("target", required=False, default=None, type=click.Path(exists=True))
```

If a `TARGET` is provided but does not exist on the filesystem, `click.Path(exists=True)` raises a `click.BadParameter` error before the indexing engine is invoked. The error message includes the path and the specific reason for failure (does not exist, permission denied, etc.).

When `TARGET` is `None` (omitted), the CLI sets it to `Path.cwd()`.

> **Historical note:** The original also defaults to the current directory when neither `-Directory` nor `-File` is specified.

> **Historical note:** The original uses two separate parameters (`-Directory` and `-File`) that are mutually exclusive. The tool uses a single positional argument and infers the target type from the filesystem. The `--file` / `--directory` flags (below) provide explicit disambiguation when needed but are not the primary input mechanism.

Target type disambiguation: `--file` / `--directory`

```
@click.option("--file/--directory", "target_type", default=None)
```

When omitted (the common case), the CLI resolves the target type from the filesystem: `Path.is_file()` → file target, `Path.is_dir()` → directory target. This auto-detection handles the vast majority of use cases without requiring the user to specify the target type.

The `--file` and `--directory` flags override auto-detection. They exist for two scenarios:

1. **Symlink disambiguation.** A symlink could point to either a file or a directory. When the user wants to force the classification (e.g., treat a symlink-to-a-directory as a file-like entry), the explicit flag overrides the filesystem inference.
2. **Scripting predictability.** Automated pipelines may prefer to state the expected target type explicitly rather than relying on filesystem inference, particularly when the target might not exist yet at script-authoring time.

If `--file` is specified but the target is a directory (or vice versa), the CLI logs a warning and proceeds with the user's explicit classification. This is a deliberate choice — the user may have a valid reason for the override (e.g., testing edge cases). The warning ensures visibility into the mismatch.

> **Historical note:** The original raises a fatal error when `-File` is specified but the path is not a file (`"The specified file does not exist"`). The tool relaxes this to a warning when the explicit flag conflicts with the filesystem type, since the positional argument already validates path existence. The hard error remains only when the target path does not exist at all.

Recursion control: `--recursive` / `--no-recursive`

```
@click.option("--recursive/--no-recursive", default=None)
```

Controls whether directory targets are traversed recursively. When omitted, the default is `True` (recursive), matching both the original's default behavior and the `IndexerConfig.recursive` default (§7.1).

For file targets, the recursion flag is silently ignored — there is nothing to recurse into. The original enforces this the same way (the `-Recursive` switch has no effect when `-File` is specified).

The `None` default (tri-state: `True`, `False`, or unspecified) allows the CLI to distinguish between "user explicitly requested non-recursive" and "user didn't specify" — relevant for configuration layering (§7.7), where a user config file setting `recursive = false` should be overridden by an explicit `--recursive` CLI flag but not by the CLI's default.

8.3. Output Mode Options

Output mode is controlled by three independent boolean flags that compose naturally. Each flag independently enables a destination, and any combination is valid.

> **Historical note:** This model replaces the original's seven-scenario routing matrix (§6.9).

`--stdout` / `--no-stdout`

```
@click.option("--stdout/--no-stdout", default=None)
```

Enables or disables writing the complete JSON output to `sys.stdout`. The tri-state default (`None`) triggers the output mode defaulting logic defined in §7.1:

- If neither `--outfile` nor `--inplace` is specified and the user did not pass `--no-stdout`, stdout is enabled.
- If `--outfile` or `--inplace` is specified, stdout is disabled unless the user explicitly passes `--stdout`.

This matches the original's behavior where `StandardOutput` defaults to `True` when no output files are specified, and defaults to `False` when `OutFile` or `OutFileInPlace` is present.

> **Historical note:** The original provides both `-StandardOutput` and `-NoStandardOutput` switches — a positive and negative flag for the same boolean, requiring a 10-line sanity check to resolve conflicts. The tool uses `click`'s built-in flag pair syntax (`--stdout/--no-stdout`), which enforces mutual exclusion at the parser level and produces the correct tri-state value without manual conflict resolution.

`--outfile`, `-o`

```
@click.option("--outfile", "-o", type=click.Path(dir_okay=False, writable=True), default=None)
```

Specifies a file path for the combined JSON output. The complete index entry tree is serialized and written to this file after all processing is complete (§6.9, timing of writes). The path is resolved to an absolute form before being stored in `IndexerConfig.output_file`.

If the parent directory of the specified path does not exist, the CLI raises a `click.BadParameter` error. The tool does not create parent directories — this is a deliberate safety choice to prevent accidental writes to unexpected locations.

If the file already exists, it is overwritten without prompting, matching the original's `Out-File -Force` behavior.

`--inplace`

```
@click.option("--inplace", is_flag=True, default=False)
```

Enables writing individual sidecar JSON files alongside each processed item during traversal (§6.9, timing of writes). File entries receive sidecar files named `<filename>_meta2.json`. Directory entries receive sidecar files named `<dirname>_directorymeta2.json`. The `2` suffix in the sidecar filenames distinguishes v2 sidecar output from any pre-existing v1 sidecar files (`_meta.json`, `_directorymeta.json`), preventing filename collisions during a transition period where both v1 and v2 indexes may coexist in the same directory tree.

8.4. Metadata Processing Options

These three flags control metadata extraction and sidecar file handling. They form an implication chain: `--meta-merge-delete` implies `--meta-merge`, which implies `--meta`. The configuration loader (§7.1) enforces these implications — the CLI passes only the flags the user explicitly specified, and the loader propagates the chain.

`--meta`, `-m`

```
@click.option("--meta", "-m", is_flag=True, default=False)
```

Enables embedded metadata extraction via `exiftool`. When active, the `metadata` array of each file entry's `IndexEntry` includes an entry with `source: "exiftool"` containing the exiftool output (§5.10). When inactive, the `metadata` array contains only sidecar-derived entries (if `--meta-merge` is also inactive, the array is empty or absent depending on the target type).

If `exiftool` is not available on the system `PATH`, the `--meta` flag does not cause a fatal error. A single warning is emitted at startup, and all EXIF metadata fields are populated with `null` for the entire invocation (§4.5, exiftool availability).

`--meta-merge`

```
@click.option("--meta-merge", is_flag=True, default=False)
```

Enables sidecar metadata merging. When active, sidecar metadata files discovered alongside indexed files are parsed and merged into the parent file's `metadata` array as entries with `origin: "sidecar"` (§6.7). The sidecar files themselves remain on disk — they are not deleted. Implies `--meta`.

`--meta-merge-delete`

```
@click.option("--meta-merge-delete", is_flag=True, default=False)
```

Extends `--meta-merge` by queuing the original sidecar files for deletion after their content has been merged into the parent entry's metadata array. Deletion occurs during Stage 6 (post-processing, §4.1), after all indexing is complete and all output has been written.

This flag carries a safety requirement: at least one persistent output mechanism (`--outfile` or `--inplace`) MUST be active when `--meta-merge-delete` is specified. Without a persistent output, the sidecar file content would be captured only in stdout — which is volatile and cannot be reliably recovered. If this safety condition is not met, the CLI exits with a fatal configuration error before any processing begins (§7.1, validation rules).

> **Historical note:** The original silently disables MetaMergeDelete when the safety condition is not met (`$MetaMergeDelete = $false`) and continues processing. The tool treats this as a fatal error. The rationale: silently disabling a destructive flag is surprising behavior that could lead a user to believe their sidecar files are safe for manual deletion when they are not. An explicit error forces the user to acknowledge the safety requirement.

8.5. Rename Option

`--rename`

```
@click.option("--rename", is_flag=True, default=False)
```

Enables the StorageName rename operation (§6.10): files are renamed from their original names to their hash-based `storage_name` values (e.g., `photo.jpg` → `yA8A8C089A6A8583B24C85F5A4A41F5AC.jpg`). Directories are not renamed — this matches the original's behavior. The `--rename` flag implies `--inplace`, because the in-place sidecar file written alongside each renamed item serves as the reversal manifest (containing the original filename in `name.text`).

The rename operation is destructive — the original filename is lost on disk. The in-place sidecar is the recovery mechanism. This implication is enforced by the configuration loader (§7.1): `rename=True` → `output_inplace=True`.

`--dry-run`

```
@click.option("--dry-run", is_flag=True, default=False)
```

Previews rename operations without executing them. When active, the rename module computes and logs the target path for each file but does not perform the actual `Path.rename()` call. The in-place sidecar files are still written (using the would-be new path), and the `IndexEntry` still contains the `storage_name`. This allows users to inspect the sidecar output and verify the rename plan before committing.

The `--dry-run` flag is only meaningful when `--rename` is also active. If `--dry-run` is specified without `--rename`, it is silently ignored — there is nothing to preview.

> **Historical note:** The original does not support dry-run for renames. This capability was added as a safety feature, particularly valuable for users running rename operations on large directory trees for the first time.

8.6. ID Type Selection

`--id-type`

```
@click.option("--id-type", type=click.Choice(["md5", "sha256"], case_sensitive=False), default=None)
```

Selects which hash algorithm is used to derive the `id` field of each `IndexEntry`. The `id` field is the primary unique identifier for an item — it determines the `storage_name` (for rename operations), the in-place sidecar filename, and the parent identity reference in child entries.

Valid values are `md5` and `sha256`. The input is case-insensitive (`MD5`, `Md5`, `md5` are all accepted and normalized to lowercase). When omitted, the value is resolved from the configuration file or the compiled default (`"md5"`, §7.2).

Both MD5 and SHA256 hashes are always computed regardless of this setting (§6.3). This flag is a presentation choice — it selects which pre-computed hash is promoted to the `id` field. The full `HashSet` containing all computed algorithms is always available in the entry's `hashes` sub-objects.

> **Historical note:** The original defaults `IdType` to `"SHA256"`. The tool defaults to `"md5"` (§7.1). MD5 produces shorter identifiers (32 hex characters vs. 64), resulting in shorter `storage_name` values and shorter sidecar filenames. For the identity use case (uniquely naming files within a local index), MD5's collision resistance is more than sufficient. Users who prefer SHA256 identifiers can set `--id-type sha256` or configure it in the TOML file.

`--compute-sha512`

```
@click.option("--compute-sha512", is_flag=True, default=False)
```

Includes SHA-512 in the computed `HashSet` for all indexed items. SHA-512 is excluded from the default hash computation because it produces 128-character hex strings that significantly inflate output size without serving a practical purpose for most indexing use cases. When this flag is active, the `sha512` field of every `HashSet` in the output is populated; when inactive, the field is omitted (§5.2.1).

SHA-512 computation is folded into the same single-pass read used for MD5 and SHA-256 (§6.3), so the marginal CPU cost is minimal. The flag controls output inclusion, not computation strategy.

8.7. Verbosity and Logging Options

`-v`, `--verbose`

```
@click.option("-v", "--verbose", count=True)
```

Increases logging verbosity. The flag is repeatable — each occurrence increases the detail level by one step. Logging output is written to `stderr`, keeping `stdout` clean for JSON output.

Flag	Effective log level	Description
(none)	`WARNING`	Default. Only warnings and errors are emitted.
`-v`	`INFO`	Progress messages: items processed, traversal decisions, output destinations.
`-vv`	`DEBUG`	Detailed internal state: hash values, exiftool commands, sidecar discovery, configuration resolution.

Flag	Effective log level	Description
`-vvv`	`DEBUG` (all)	Maximum verbosity. Includes all `DEBUG` messages including internal timing and per-item trace data.

The mapping from `-v` count to Python `logging` level is:

```
log_level = {0: logging.WARNING, 1: logging.INFO}.get(verbose, logging.DEBUG)
```

Any count ≥ 2 maps to `DEBUG`. The distinction between `-vv` and `-vvv` is handled within the logging configuration by enabling or disabling specific logger names, not by defining additional log levels.

> **Historical note:** The original uses a binary `$Verbosity` boolean (`$true`/`$false`). The tool adopts a graduated verbosity model consistent with Unix CLI conventions and Python's `logging` framework. The original's `$Verbosity = $true` maps approximately to `-v` (INFO level).

`-q`, `--quiet`

```
@click.option("-q", "--quiet", is_flag=True, default=False)
```

Suppresses all logging output except fatal errors. Equivalent to setting the log level to `CRITICAL`. When `--quiet` is active, only errors that prevent the tool from producing output are emitted to `stderr`.

If both `--verbose` and `--quiet` are specified, `--quiet` wins. This follows the principle that silence is an explicit request and should not be overridden by another flag.

`--log-file`

> **Added 2026-02-23:** Persistent log file support. See §11.1, Principle 3 and §11.5, File destinations.

```
@click.option("--log-file", default=None, is_flag=False, flag_value="")
```

Enables persistent log file output alongside the default stderr handler. The flag has two forms:

Usage	Behavior
`--log-file` (no argument)	Write to the platform-specific default app data directory (§11.1, Principle 3)
`--log-file /path/to/file.log`	Write to the specified path

The log file uses the CLI's full format including session ID. The log level written to the file matches the currently configured verbosity (controlled by `-v`/`-q`). The log file is also expressible in the TOML configuration file via `[logging] file_enabled = true` (§11.5, File destinations).

8.8. Mutual Exclusion Rules and Validation

The CLI validates flag combinations before passing them to the configuration loader. Some validations are enforced by `click` declaratively; others are checked programmatically in the CLI's main function body.

Parser-level enforcement (handled by `click`)

Constraint	Mechanism
`--file` and `--directory` are mutually exclusive	`click.option("--file/--directory")` flag pair
`--recursive` and `--no-recursive` are mutually exclusive	`click.option("--recursive/--no-recursive")` flag pair
`--stdout` and `--no-stdout` are mutually exclusive	`click.option("--stdout/--no-stdout")` flag pair
`--id-type` accepts only `md5` or `sha256`	`click.Choice(["md5", "sha256"])`
`--verbose` and `--quiet` coexistence	Programmatic: `--quiet` overrides `--verbose`

Programmatic validation (checked before config construction)

Rule	Behavior
`--meta-merge-delete` requires `--outfile` or `--inplace`	Fatal error with exit code 2 (§8.10). Error message explains the safety requirement.
`--dry-run` without `--rename`	Silently ignored. No error, no warning — the flag simply has no effect.

Rule	Behavior
`TARGET` does not exist	Fatal error raised by `click.Path(exists=True)` before main function body executes.
`--outfile` parent directory does not exist	Fatal error raised by the CLI after path resolution.

Implication propagation (handled by configuration loader)

The CLI does NOT perform implication propagation itself. It passes the user's raw flag values to the configuration loader (§7.1), which applies the implication chain:

- `--rename` → `output_inplace = True`
- `--meta-merge-delete` → `meta_merge = True`
- `--meta-merge` → `extract_exif = True`

The CLI communicates these implications to the user through log messages at the `INFO` level when an implied flag is activated. For example, when the user specifies `--rename` without `--inplace`, the log emits: `"--rename implies --inplace; enabling in-place output"`. This visibility mirrors the original's `Vbs` warnings for the same conditions.

8.9. Output Scenarios

The three output flags (`--stdout`, `--outfile`, `--inplace`) compose into all valid output configurations. The following table enumerates the practical scenarios, mapped to the original's seven-scenario numbering for traceability.

Scenario	Flags	Stdout	Outfile	Inplace	Original equivalent
1 (default)	(none)	✔	—	—	Scenario 1: `StandardOutput` only
2	`--outfile index.json`	—	✔	—	Scenario 2: `OutFile` only
3	`--outfile index.json --stdout`	✔	✔	—	Scenario 3: `OutFile` + `StandardOutput`
4	`--inplace`	—	—	✔	Scenario 4: `OutFileInPlace` only
5	`--outfile index.json --inplace`	—	✔	✔	Scenario 5: `OutFile` + `OutFileInPlace`
6	`--inplace --stdout`	✔	—	✔	Scenario 6: `OutFileInPlace` + `StandardOutput`
7	`--outfile index.json --inplace --stdout`	✔	✔	✔	Scenario 7: all three
Silent	`--no-stdout`	—	—	—	Original: `NoStandardOutput` without output files

The "Silent" scenario (no output of any kind) is valid but produces a warning: `"No output destinations are enabled. The indexing operation will execute but produce no output."` The original similarly warns when this condition occurs. The scenario is not prevented because it can be useful for side-effect-only invocations (e.g., `--rename` without needing the JSON output) or for measuring indexing performance without I/O overhead.

Example invocations

```
# Index the current directory recursively, output to stdout (default)
shruggie-indexer

# Index a specific directory, write to a file
shruggie-indexer /path/to/photos --outfile index.json

# Index a single file with metadata extraction
shruggie-indexer /path/to/photo.jpg --meta

# Index recursively with metadata merge, write in-place sidecars
shruggie-indexer /path/to/media --meta-merge --inplace

# Rename files to storage names (dry run)
shruggie-indexer /path/to/files --rename --dry-run -v

# Merge and delete sidecars, write both in-place and aggregate
shruggie-indexer /path/to/archive --meta-merge-delete --outfile archive.json --inplace

# Index non-recursively with SHA-256 identifiers
shruggie-indexer /path/to/dir --no-recursive --id-type sha256

# Index with maximum verbosity and custom config
shruggie-indexer /path/to/dir -vvv --config my-config.toml

# Quiet mode — errors only, output to file
shruggie-indexer /path/to/dir --outfile out.json -q
```

8.10. Exit Codes

The CLI uses structured exit codes to communicate outcome status to calling processes. Exit codes are integers in the range 0–5.

Code	Name	Meaning
0	`SUCCESS`	Indexing completed successfully. All requested items were processed and all output destinations were written.
1	`PARTIAL_FAILURE`	Indexing completed with one or more item-level errors. The output was produced but some items may have degraded fields (null hashes, missing metadata) or may have been skipped entirely. The count of failed items is logged at the `WARNING` level.
2	`CONFIGURATION_ERROR`	The invocation failed before any processing began due to invalid configuration: invalid flag combination (e.g., `--meta-merge-delete` without a persistent output), unreadable configuration file, invalid TOML syntax, or unrecognized `--id-type` value. No output is produced.
3	`TARGET_ERROR`	The target path does not exist, is not accessible, or is neither a file nor a directory. This is the exit code produced when `click.Path(exists=True)` rejects the target, or when the resolved target fails classification (§4.6). No output is produced.
4	`RUNTIME_ERROR`	An unexpected error occurred during processing that prevented the tool from completing. This covers unhandled exceptions, filesystem errors that affect the entire operation (e.g., the output file path becomes unwritable mid-operation), or `exiftool` crashes that propagate beyond the per-item error boundary. Partial output may exist for `--inplace` mode (sidecar files written before the failure); `--stdout` and `--outfile` output is not produced.
5	`INTERRUPTED`	The operation was cancelled by the user via `SIGINT` (`Ctrl+C`). See §8.11 for the full signal handling specification. Partial `--inplace` sidecar output may exist on disk for items completed before the interrupt. No `--stdout` or `--outfile` output is produced. The MetaMergeDelete deletion queue (§4.1, Stage 6) is discarded — no sidecar files are deleted.

Exit codes are defined as an `IntEnum` in `cli/main.py`:

```
from enum import IntEnum

class ExitCode(IntEnum):
    SUCCESS = 0
    PARTIAL_FAILURE = 1
    CONFIGURATION_ERROR = 2
    TARGET_ERROR = 3
    RUNTIME_ERROR = 4
    INTERRUPTED = 5
```

The `main()` function wraps the entire invocation in a `try`/`except` structure that maps exception types to exit codes:

```
# Illustrative — not the exact implementation.
def main():
    cancel_event = threading.Event()
    _install_signal_handlers(cancel_event)  # §8.11

    try:
        # ... click CLI setup, config construction ...
        entry = index_path(
            target, config,
            progress_callback=progress_cb,
            cancel_event=cancel_event,
        )
        # ... output routing ...
        if failed_items > 0:
            sys.exit(ExitCode.PARTIAL_FAILURE)
        sys.exit(ExitCode.SUCCESS)
    except IndexerCancellationError:
        logger.warning("Operation interrupted — exiting cleanly.")
        sys.exit(ExitCode.INTERRUPTED)
    except ConfigurationError:
        sys.exit(ExitCode.CONFIGURATION_ERROR)
    except TargetError:
        sys.exit(ExitCode.TARGET_ERROR)
    except KeyboardInterrupt:
        # Second Ctrl+C bypassed cooperative cancellation (§8.11)
        logger.warning("Forced termination.")
```

```
        sys.exit(ExitCode.INTERRUPTED)
    except Exception:
        logger.exception("Unexpected error during indexing")
        sys.exit(ExitCode.RUNTIME_ERROR)
```

> **Historical note:** The original does not define exit codes. It uses PowerShell's default `return` behavior, which yields `$null` for most error conditions and relies on `Vbs` log messages for error visibility. Scripts calling `MakeIndex` cannot programmatically distinguish between "completed with warnings" and "failed entirely." The tool's structured exit codes enable reliable error handling in automation pipelines.

Exit code interaction with `--quiet`

When `--quiet` is active, the exit code becomes the primary signal for success or failure. The calling process MUST inspect `$?` (shell) or the subprocess return code (Python) to determine the outcome. Fatal error messages (exit codes 2–4) are still emitted to `stderr` even in quiet mode — `--quiet` suppresses informational and warning messages, not fatal errors. The interruption message (exit code 5) is also emitted to `stderr` in quiet mode, as it represents a non-normal termination the caller must be able to detect.

Exit code interaction with `--inplace`

Because `--inplace` writes sidecar files incrementally during traversal (§6.9), a `RUNTIME_ERROR` (exit code 4) or `INTERRUPTED` (exit code 5) does not necessarily mean zero output was produced. Sidecar files written before the failure or interruption point are valid and usable. Calling processes that use `--inplace` SHOULD handle exit codes 4 and 5 by checking for partial sidecar output rather than assuming complete failure.

8.11. Signal Handling and Graceful Interruption

The CLI supports cooperative cancellation via `SIGINT` (`Ctrl+C`), reusing the same cancellation infrastructure defined for the GUI (§10.5). This ensures that long-running indexing operations on large directory trees can be interrupted cleanly — with the current item allowed to complete, partial `--inplace` output preserved, and destructive operations (MetaMergeDelete) safely aborted.

Design rationale

Without explicit signal handling, Python's default `SIGINT` behavior raises `KeyboardInterrupt` at an arbitrary point in execution. This is problematic for `shruggie-indexer` because an uncontrolled interrupt can leave partially written sidecar files, corrupt an in-progress `--outfile` write, or — most critically — interrupt a MetaMergeDelete operation after some sidecar files have been deleted but before their content has been persisted to the aggregate output. The cooperative model eliminates these failure modes by channeling interruption through the same `cancel_event` mechanism the GUI uses, which the engine checks only at safe item boundaries.

Mechanism

The CLI installs a custom `SIGINT` handler before invoking the indexing engine. The handler interacts with the `cancel_event` (`threading.Event`) that is passed to `index_path()`.

```
import signal
import threading
import sys

def _install_signal_handlers(cancel_event: threading.Event) -> None:
    """Install SIGINT handler for cooperative cancellation.

    First SIGINT: set the cancel_event so the engine stops at the
    next item boundary. Second SIGINT: restore default behavior and
    re-raise, forcing immediate termination.
    """
    def _handle_sigint(signum, frame):
        if cancel_event.is_set():
            # Second interrupt — user wants out NOW.
            signal.signal(signal.SIGINT, signal.SIG_DFL)
            raise KeyboardInterrupt
        cancel_event.set()
        # Message goes to stderr so it doesn't corrupt stdout JSON.
        print(
            "\nInterrupt received — finishing current item. "
            "Press Ctrl+C again to force quit.",
            file=sys.stderr,
        )

    signal.signal(signal.SIGINT, _handle_sigint)
```

Two-phase interruption model

The handler implements a two-phase model that balances safety with user control:

Phase 1 — Cooperative cancellation (first `Ctrl+C`):

1. The handler sets `cancel_event`.
2. A message is printed to `stderr` informing the user that the current item will complete before the operation stops.
3. The indexing engine detects `cancel_event.is_set()` at the next item boundary in the child-processing loop of `build_directory_entry()` (§6.8) and raises `IndexerCancellationError`.
4. The `main()` function catches `IndexerCancellationError`, logs a clean shutdown message, and exits with code 5 (`INTERRUPTED`).

Phase 2 — Forced termination (second `Ctrl+C`):

1. Because `cancel_event` is already set, the handler knows this is a repeat interrupt.
2. The handler restores Python's default `SIGINT` disposition (`signal.SIG_DFL`) and raises `KeyboardInterrupt`.
3. The `main()` function catches `KeyboardInterrupt` in its outermost exception handler and exits with code 5 (`INTERRUPTED`).
4. If `main()` does not catch it (e.g., the interrupt occurs in a context where the handler's frame is inside a system call), Python's default behavior terminates the process immediately.

The two-phase model is a well-established CLI pattern (used by `git`, `rsync`, `docker`, among others) that gives the user an escape hatch without sacrificing safety in the common case.

Cancellation granularity

Cancellation is per-item, not mid-item. This is inherited from the core engine's cancellation contract (§6.8, §10.5). Once the engine begins processing a single file or directory entry — hashing, exiftool invocation, sidecar discovery — that item runs to completion before the cancellation flag is checked. This guarantee ensures that every item in the output is complete: no partial hashes, no half-written sidecar files, no items with metadata from exiftool but missing hash fields.

For single-file targets, cooperative cancellation has no practical effect — the entire operation is essentially atomic (a single `build_file_entry()` call). The `cancel_event` is passed to `index_path()` but is only checked during directory traversal loops. A `SIGINT` during single-file processing falls through to the `KeyboardInterrupt` handler in `main()` and exits with code 5.

Cancellation and output state

The following table describes what happens to each output destination when the operation is interrupted:

Output mode	State after interruption
`--stdout`	No output produced. The JSON tree is serialized only after the complete `IndexEntry` is assembled (§6.9), which does not happen on cancellation.
`--outfile`	No output produced. The aggregate file is written atomically after processing completes. If the file existed before the invocation, it is not modified (the write had not yet begun).
`--inplace`	Partial output. Sidecar files for items completed before the interrupt are valid and present on disk. No sidecar exists for the item that was in progress when `cancel_event` was set (it completes, but the engine raises `IndexerCancellationError` before advancing to the next item — the just-completed item's sidecar IS written).

Cancellation and MetaMergeDelete

If a MetaMergeDelete operation is interrupted, the deletion queue (Stage 6, §4.1) is discarded. No sidecar files are deleted. This matches the GUI's cancellation behavior (§10.5) and is the safe default — the user can re-run the operation to completion if desired, and all source sidecar files remain intact.

Cancellation and `--rename`

If a rename operation is interrupted, files renamed before the interrupt retain their new names. The corresponding in-place sidecar files (the reversal manifests) are present alongside each renamed file. Files not yet reached by the traversal retain their original names. This is the same partial-completion behavior as `--inplace` — the operation is idempotent with respect to re-execution, since already-renamed files will produce the same `storage_name` on a subsequent run.

Platform considerations

`SIGINT` handling via `signal.signal()` is portable across all target platforms (Windows, Linux, macOS). On Windows, `SIGINT` is delivered when the user presses `Ctrl+C` in a console window or when `GenerateConsoleCtrlEvent(CTRL_C_EVENT)` is called programmatically. The `signal` module's behavior on Windows has some limitations (signals can only be handled on the main thread, `SIGINT` is the only reliably catchable signal), but these do not affect the CLI's use case — the CLI runs on the main thread and only needs to catch `SIGINT`.

`SIGTERM` is not handled by the CLI. On Unix systems, `SIGTERM` defaults to process termination, which is the expected behavior for external process managers (systemd, Docker, etc.) that send `SIGTERM` as a "please shut down" signal. Adding `SIGTERM` handling with cooperative cancellation is a reasonable future

enhancement but is not required for the MVP — `SIGTERM` senders typically follow up with `SIGKILL` after a timeout, and the two-phase `SIGINT` model already covers the interactive use case.

Progress bar interaction

When the CLI is displaying a progress bar (via `tqdm` or `rich`, see §11.6), the interrupt message from Phase 1 must not corrupt the progress bar rendering. Both `tqdm` and `rich` provide mechanisms for this: `tqdm.write()` prints a message without disrupting the active progress bar, and `rich.console.Console` supports `stderr` output that coexists with progress displays. The interrupt message MUST use the progress library's safe-print mechanism rather than a bare `print()` call. The illustrative code in the mechanism section above uses `print()` for clarity; the implementation MUST substitute the appropriate library-aware output method.

> **Historical note:** The original PowerShell implementation has no interrupt handling. `Ctrl+C` during a `MakeIndex` invocation triggers PowerShell's default `StopProcessing()` behavior, which terminates the pipeline immediately with no cleanup. This can leave partially written output files, orphaned temporary files (from `TempOpen`/`TempClose`), and — in MetaMergeDelete mode — a state where some sidecar files have been deleted but the aggregate output was never written. The tool's cooperative cancellation model eliminates all of these failure modes.

9. Python API

This section defines the public programmatic interface for consumers who `import shruggie_indexer` as a Python library rather than invoking it from the command line or GUI. The API is the contract between the library and its consumers — it specifies which names are importable, what each function accepts and returns, what exceptions it raises, and what behavioral guarantees it provides. The CLI (§8) and GUI (§10) are both consumers of this API; they add no indexing logic of their own.

The API is design goal G3 in action (§2.3): a single core engine exposed through three delivery surfaces. The Python API is the most direct of the three — it provides access to the core functions without the overhead of argument parsing (CLI) or event-loop management (GUI). Consumers include: automation scripts that process index output programmatically, applications that embed indexing functionality, test harnesses that need fine-grained control over individual indexing steps, and the CLI/GUI modules themselves.

Module location: The public API surface is defined in `__init__.py` at the top level of the `shruggie_indexer` package (§3.2). Internal modules (`core/`, `config/`, `models/`) are implementation details — consumers import from the top-level namespace, not from subpackages.

> **Historical note:** The original `MakeIndex` is a single PowerShell function with 14 parameters that conflates configuration, execution, and output routing in one call. There is no "library API" — calling `MakeIndex` from another PowerShell script is possible but not designed for, and the function relies on global state (`$global:MetadataFileParser`, `$global:DeleteQueue`) that makes isolated invocation fragile. The tool's API separates configuration construction from indexing execution from output routing, each with its own function and return type, enabling clean programmatic composition.

9.1. Public API Surface

The `__init__.py` module exports the following names. These constitute the complete public API — all other modules and names are internal.

```python
# src/shruggie_indexer/__init__.py

from shruggie_indexer._version import __version__
from shruggie_indexer.config.loader import load_config
from shruggie_indexer.config.types import IndexerConfig, MetadataTypeAttributes
from shruggie_indexer.core.entry import (
    index_path,
    build_file_entry,
    build_directory_entry,
    IndexerCancellationError,
)
from shruggie_indexer.core.progress import ProgressEvent
from shruggie_indexer.core.serializer import serialize_entry
from shruggie_indexer.models.schema import (
    IndexEntry,
    MetadataEntry,
    HashSet,
    NameObject,
    SizeObject,
    TimestampPair,
    TimestampsObject,
    ParentObject,
)

__all__ = [
    # Version
    "__version__",
    # Configuration
    "load_config",
    "IndexerConfig",
```

```
    "MetadataTypeAttributes",
    # Core functions
    "index_path",
    "build_file_entry",
    "build_directory_entry",
    "serialize_entry",
    # Progress and cancellation
    "ProgressEvent",
    "IndexerCancellationError",
    # Data models
    "IndexEntry",
    "MetadataEntry",
    "HashSet",
    "NameObject",
    "SizeObject",
    "TimestampPair",
    "TimestampsObject",
    "ParentObject",
]
```

The `__all__` list is exhaustive — it defines the complete set of names available via `from shruggie_indexer import *`. Names not in `__all__` are not part of the public API and may change without notice between versions. Consumers who import from subpackages (`from shruggie_indexer.core.hashing import hash_file`) do so at their own risk — those paths are internal and may be restructured.

API stability scope

For the v0.1.0 MVP release, the public API is provisional. Breaking changes may occur in minor version bumps (0.2.0, 0.3.0) without a deprecation cycle. Once the project reaches 1.0.0, the public API surface defined here becomes subject to semantic versioning: breaking changes require a major version bump. The JSON output schema (§5) and the public API are the two stability boundaries that downstream consumers can depend on.

Dependency isolation

The public API does not require any optional dependencies beyond the four required runtime packages (`click`, `orjson`, `pyexiftool`, `tqdm`). All exported names are importable after a standard `pip install shruggie-indexer`. The optional GUI dependency (`customtkinter`) is isolated behind an import guard in its module — it is never imported at the top-level `__init__.py` scope.

This is an explicit design constraint: a consumer who `pip install shruggie-indexer` can immediately `from shruggie_indexer import index_path, load_config` and use the library without encountering `ImportError`. The GUI extra adds a presentation layer, not core functionality.

9.2. Core Functions

These are the primary functions for performing indexing operations. They compose in a layered hierarchy: `index_path()` calls `build_file_entry()` or `build_directory_entry()`, which in turn call into the `core/` component modules. Consumers can invoke at any layer depending on their needs.

`index_path()`

The top-level entry point. Given a filesystem path and a configuration, produces a complete `IndexEntry`. This is the function the CLI and GUI call — it handles target resolution, classification, and dispatch.

```
def index_path(
    target: Path | str,
    config: IndexerConfig | None = None,
    *,
    progress_callback: Callable[[ProgressEvent], None] | None = None,
    cancel_event: threading.Event | None = None,
) -> IndexEntry:
    """Index a file or directory and return the complete entry.

    Args:
        target: Path to the file or directory to index. Strings are
                converted to Path objects. Relative paths are resolved
                against the current working directory.
        config: Resolved configuration. When None, compiled defaults
                are used (equivalent to load_config() with no arguments).
        progress_callback: Optional callable invoked after each item
                is processed during directory traversal. Receives a
                ProgressEvent with phase, item counts, current path,
                and log-level messages. Ignored for single-file
                targets (the operation is effectively atomic).
                Callers MUST NOT perform long-running work inside the
                callback — it executes on the indexing thread and
```

```
                blocks further processing until it returns. GUI and
                CLI consumers should enqueue events for asynchronous
                display rather than updating UI directly.
        cancel_event: Optional threading.Event checked between items
                during directory traversal. When the event is set,
                the engine stops processing at the next item boundary
                and raises IndexerCancellationError. The partially
                built entry tree is not returned — cancellation is
                a clean abort, not a partial result. Ignored for
                single-file targets.

    Returns:
        A fully populated IndexEntry conforming to the v2 schema.

    Raises:
        IndexerTargetError: The target path does not exist, is not
            accessible, or is neither a file nor a directory.
        IndexerConfigError: The configuration is invalid (e.g.,
            meta_merge_delete=True with no output destination).
            Only raised when config is constructed internally
            from defaults.
        IndexerRuntimeError: An unrecoverable error occurred during
            processing (e.g., filesystem became unavailable mid-
            operation).
        IndexerCancellationError: The cancel_event was set during
            processing. No partial output is returned.
    """
```

Behavioral contract:

1. If `target` is a string, it is converted to a `Path` via `Path(target)`.
2. The path is resolved to an absolute canonical form via `core.paths.resolve_path()` (§6.2).
3. If `config` is `None`, the function calls `load_config()` with no arguments to obtain compiled defaults. This is a convenience for simple scripting — production callers should construct and reuse a config explicitly.
4. The target is classified as file or directory per §4.6 and dispatched to `build_file_entry()` or `build_directory_entry()`.
5. For directory targets, `progress_callback` and `cancel_event` are forwarded to `build_directory_entry()`, which checks the cancellation flag and invokes the callback at each item boundary during the child-processing loop. For single-file targets, neither parameter has any effect.
6. The returned `IndexEntry` is complete and ready for serialization via `serialize_entry()`, inspection via attribute access, or further programmatic processing.

Default config convenience: The `config=None` default is a deliberate API usability decision. The most common programmatic use case — "index this path with default settings" — should require exactly one function call, not two. The config object is still constructed properly through the full `load_config()` pipeline, including user config file resolution. Callers who need non-default settings must construct a config explicitly.

> **Historical note:** The original `MakeIndex` accepts a `-Directory` or `-File` parameter, one of four output mode switches, a `-Meta`/`-MetaMerge`/`-MetaMergeDelete` switch, and performs output routing internally. The tool's `index_path()` does none of that — it accepts a path and a config, produces a data structure, and returns it. Output routing is the caller's responsibility. This separation is what makes the function composable: the same `IndexEntry` can be serialized to stdout, written to a file, stored in a database, or piped into another processing step, without `index_path()` knowing or caring about the destination.

`build_file_entry()`

Builds an `IndexEntry` for a single file. This is the lower-level function that `index_path()` dispatches to when the target is a file. Consumers who already know they are indexing a file (and want to skip the target classification step) can call this directly.

```
def build_file_entry(
    path: Path,
    config: IndexerConfig,
    *,
    siblings: list[Path] | None = None,
    delete_queue: list[Path] | None = None,
) -> IndexEntry:
    """Build a complete IndexEntry for a single file.

    Args:
        path: Absolute path to the file. Must exist and be a file
              (or a symlink to a file).
        config: Resolved configuration.
        siblings: Pre-enumerated list of all files in the same
                  directory. Used for sidecar metadata discovery.
                  When None, the directory is enumerated on demand
```

```
                  (acceptable for single-file use, inefficient for
                  batch use within a directory traversal).
        delete_queue: When MetaMergeDelete is active, sidecar paths
                      discovered during metadata processing are
                      appended to this list for deferred deletion
                      by the caller. When None, no paths are
                      collected (MetaMergeDelete is inactive or
                      the caller manages deletion separately).

    Returns:
        A fully populated IndexEntry with type="file" and items=null.

    Raises:
        IndexerTargetError: The path does not exist or is not a file.
        OSError: Filesystem-level errors (permission denied, I/O error)
            propagate for the file itself. Component-level errors
            (exiftool failure, sidecar parse error) are handled
            internally and result in degraded fields (null values),
            not exceptions.
    """
```

The `siblings` parameter is an optimization for directory traversal. When `build_directory_entry()` calls `build_file_entry()` for each child file, it passes the pre-enumerated sibling list from `traversal.list_children()`. This avoids re-scanning the directory for sidecar discovery on every file (§6.7). When `build_file_entry()` is called standalone (via `index_path()` for a single-file target), `siblings` is `None` and the function enumerates the parent directory once internally.

The `delete_queue` parameter implements the MetaMergeDelete accumulation pattern described in §6.7. The function does not delete files itself — it only appends paths to the caller-provided list. Actual deletion is a post-processing step (Stage 6, §4.1) handled by the top-level orchestrator. This separation ensures that file deletion never occurs mid-traversal if the operation is interrupted.

`build_directory_entry()`

Builds an `IndexEntry` for a directory, optionally recursing into subdirectories.

```
def build_directory_entry(
    path: Path,
    config: IndexerConfig,
    *,
    recursive: bool = False,
    delete_queue: list[Path] | None = None,
    progress_callback: Callable[[ProgressEvent], None] | None = None,
    cancel_event: threading.Event | None = None,
) -> IndexEntry:
    """Build a complete IndexEntry for a directory.

    Args:
        path: Absolute path to the directory. Must exist and be a
              directory.
        config: Resolved configuration.
        recursive: When True, descends into subdirectories and
                   populates items with a fully nested tree. When
                   False, populates items with immediate children
                   only (flat mode).
        delete_queue: MetaMergeDelete accumulator (see
                      build_file_entry).
        progress_callback: Optional callable invoked after each
                child item is processed (file built or directory
                recursion completed) and once after child discovery.
                The callback receives a ProgressEvent whose
                items_total field is populated once discovery
                completes and whose items_completed field
                increments with each processed child. When this
                function calls itself recursively for child
                directories, it forwards the same callback so
                that progress reporting spans the entire tree.
        cancel_event: Optional threading.Event checked at the top
                of each iteration of the child-processing loop.
                When set, the function raises
                IndexerCancellationError immediately, before
                processing the next child. Items already processed
                are discarded (the partially built entry is not
                returned). When this function calls itself
```

```
                recursively, it forwards the same Event so that
                cancellation is effective at any depth.

    Returns:
        A fully populated IndexEntry with type="directory" and
        items populated with child entries.

    Raises:
        IndexerTargetError: The path does not exist or is not a
            directory.
        IndexerCancellationError: The cancel_event was set during
            the child-processing loop.
        OSError: Fatal directory-level errors (cannot enumerate
            children) propagate. Item-level errors within child
            entries are handled per the error tier model (§4.5) —
            the child is either skipped or included with degraded
            fields, and the parent entry is still returned.
    """
```

Recursion control: The `recursive` parameter directly maps to `config.recursive` when called from `index_path()`, but is a separate explicit parameter rather than being read from the config object. This is intentional — it allows `build_directory_entry()` to call itself with `recursive=True` for child directories regardless of the top-level config value, without mutating the config. The config controls whether recursion starts; the parameter controls whether it continues.

> **Architectural note:** The `recursive` parameter on `build_directory_entry()` may appear redundant with `config.recursive`, but the separation is necessary. When `index_path()` processes a recursive directory target, it calls `build_directory_entry(path, config, recursive=True)`. The directory entry builder then calls itself for child directories with `recursive=True` — the recursion depth is not a config concern, it is a call-graph concern. If a future enhancement adds depth-limited recursion (`max_depth=3`), the control mechanism is this parameter, not a config mutation.

`serialize_entry()`

Converts an `IndexEntry` to a JSON string. This is the serialization boundary — everything before this function operates on Python data structures; everything after operates on text.

```
def serialize_entry(
    entry: IndexEntry,
    *,
    indent: int | None = 2,
    sort_keys: bool = False,
) -> str:
    """Serialize an IndexEntry to a JSON string.

    Args:
        entry: The IndexEntry to serialize.
        indent: JSON indentation level. 2 (default) produces
                human-readable output matching the legacy
                ConvertTo-Json formatting. None produces compact
                single-line JSON.
        sort_keys: Whether to sort JSON object keys alphabetically.
                   Default False preserves the schema-defined field
                   order (schema_version first, then identity fields,
                   then content fields, etc.).

    Returns:
        A UTF-8 JSON string conforming to the v2 schema.
    """
```

Serialization invariants are enforced here, not by the caller (§5.12):

1. `schema_version` appears first in the output (when `sort_keys=False`).
2. Required fields are always present.
3. `HashSet.sha512` is omitted (not emitted as `null`) when it was not computed.
4. Sidecar-only `MetadataEntry` fields (`file_system`, `size`, `timestamps`) are present for sidecar entries and absent for generated entries.
5. All hash hex strings are uppercase.

The function uses `json.dumps()` from the standard library by default. When `orjson` is installed, the serializer MAY use it for improved performance on large trees — this is an internal optimization that does not change the output format. The selection is transparent to the caller.

9.3. Configuration API

Configuration construction is separated from indexing execution. Callers build an `IndexerConfig` object first, then pass it to the core functions. This separation enables config reuse across multiple indexing operations (e.g., indexing several directories with the same settings) and makes the configuration inspectable and testable independently of the indexing engine.

`load_config()`

The sole factory for `IndexerConfig` objects. Implements the full four-layer resolution pipeline described in §7.1.

```
def load_config(
    *,
    config_file: Path | str | None = None,
    target_directory: Path | str | None = None,
    overrides: dict[str, Any] | None = None,
) -> IndexerConfig:
    """Construct a fully resolved, immutable IndexerConfig.

    Resolution layers (lowest to highest priority):
    1. Compiled defaults (§7.2).
    2. User config file at the platform-standard location (§3.3),
       unless config_file is specified.
    3. Project-local config file in target_directory or its ancestors,
       if target_directory is provided.
    4. Explicit overrides from the overrides dict.

    After merging, the function applies parameter implications
    (§7.1), validates the result, compiles regex patterns, and
    returns an immutable IndexerConfig.

    Args:
        config_file: Explicit path to a TOML configuration file.
                     When provided, this replaces the user config
                     file resolution (layer 2) — the platform-
                     standard location is not searched. When None,
                     the standard resolution logic applies.
        target_directory: The directory being indexed. Used to
                          search for a project-local config file.
                          When None, no project-local config is
                          loaded.
        overrides: Dict of field-name-to-value overrides applied
                   as the highest-priority layer. Keys correspond
                   to IndexerConfig field names. Unknown keys
                   produce a warning and are ignored.

    Returns:
        A frozen IndexerConfig instance.

    Raises:
        IndexerConfigError: The config file does not exist, contains
            invalid TOML, contains values of the wrong type, or the
            resolved configuration fails validation (e.g.,
            meta_merge_delete without an output destination).
    """
```

Override dict convention: The `overrides` dict uses flat `IndexerConfig` field names as keys. Nested configuration (e.g., exiftool settings) uses dotted key paths: `{"exiftool.exclude_extensions": frozenset({"csv", "xml"})}`. This matches the TOML section structure and avoids requiring callers to construct nested dicts.

The CLI module constructs the `overrides` dict from parsed `click` arguments. API callers construct it directly:

```
config = load_config(overrides={
    "id_algorithm": "sha256",
    "extract_exif": True,
    "recursive": True,
})
```

Implications and validation are applied identically to the CLI path — the configuration loader does not distinguish between a CLI caller and an API caller. If `overrides={"meta_merge_delete": True}` is passed without also setting an output destination, the function raises `IndexerConfigError` with the same message the CLI would produce.

`IndexerConfig`

The frozen dataclass that carries all configuration. Fully defined in §7.1 — this section documents only the API-facing aspects.

```
@dataclass(frozen=True)
class IndexerConfig:
    """Immutable configuration for a single indexing invocation.

    All fields have defaults — an IndexerConfig constructed with
    no arguments represents the compiled default configuration.

    This class is frozen (immutable). To create a modified copy,
    use dataclasses.replace():

        new_config = replace(config, id_algorithm="sha256")
    """
```

The `frozen=True` guarantee means callers can safely share a single `IndexerConfig` across threads or async tasks without synchronization. The `dataclasses.replace()` pattern provides the modification mechanism — rather than mutating the config, callers create a new instance with the desired field(s) changed. This is the same pattern used for immutable configuration in the broader Python ecosystem (e.g., `attrs`, Pydantic's `model_copy()`).

Direct construction vs. `load_config()`: Callers MAY construct `IndexerConfig` directly for testing or for scenarios where the full config resolution pipeline is unnecessary. However, direct construction bypasses parameter implications and validation. A directly-constructed config with `meta_merge_delete=True, meta_merge=False` is structurally valid (the dataclass accepts it) but behaviorally invalid (MetaMergeDelete requires MetaMerge). `load_config()` prevents this; direct construction does not. The API does not enforce implications on direct construction because it would require the dataclass `__post_init__` to have side effects, which conflicts with the frozen guarantee.

Callers who construct `IndexerConfig` directly MUST ensure that parameter implications are satisfied:

- `rename=True` → `output_inplace=True`
- `meta_merge_delete=True` → `meta_merge=True`
- `meta_merge=True` → `extract_exif=True`

9.4. Data Classes and Type Definitions

All data classes used in the public API are defined in `models/schema.py` (§3.2). They are the Python representation of the v2 JSON Schema types defined in §5. Each class maps directly to a schema `$ref` definition or to the root `IndexEntry` type.

Model implementation strategy

The data classes are implemented as `@dataclass` types using the standard library. They are NOT frozen — unlike `IndexerConfig`, the entry models are constructed incrementally during the building process (§6.8) and may have fields set in multiple steps. The immutability boundary is the API return: once `index_path()`, `build_file_entry()`, or `build_directory_entry()` returns an `IndexEntry`, the caller SHOULD treat it as immutable. Mutation after return is not prohibited but is unsupported and may produce inconsistent serialization output.

For consumers who want runtime type validation (e.g., when ingesting index output from untrusted sources), `models/schema.py` provides optional Pydantic models behind an import guard. The Pydantic models mirror the dataclass definitions and add `model_validate_json()` for schema-validating a JSON string. They are not used by the core engine.

```
# Standard usage — dataclass models (no extra dependency)
from shruggie_indexer import IndexEntry

# Optional — Pydantic models for runtime validation
try:
    from shruggie_indexer.models.schema import IndexEntryModel  # Pydantic
    entry = IndexEntryModel.model_validate_json(json_string)
except ImportError:
    # Pydantic not installed; fall back to json.loads() + manual access
    import json
    entry_dict = json.loads(json_string)
```

`IndexEntry`

The root data class. Represents a single indexed file or directory.

```
@dataclass
class IndexEntry:
    """A single indexed file or directory (v2 schema)."""
```

```
    schema_version: int  # Always 2
    id: str              # Prefixed hash: y... (file), x... (directory)
    id_algorithm: str    # "md5" or "sha256"
    type: str            # "file" or "directory"

    name: NameObject
    extension: str | None
    size: SizeObject
    hashes: HashSet | None  # Content hashes (file) or null (directory/symlink)

    file_system: FileSystemObject
    timestamps: TimestampsObject
    attributes: AttributesObject

    items: list[IndexEntry] | None = None    # Children (directory) or null (file)
    metadata: list[MetadataEntry] | None = None  # Metadata entries or null
    mime_type: str | None = None
```

All fields listed in §5.3 are present. Required schema fields have no default value; optional schema fields (`items`, `metadata`, `mime_type`) default to `None`. The field order matches the schema-defined serialization order.

HashSet

```
@dataclass
class HashSet:
    """Cryptographic hash digests (§5.2.1)."""

    md5: str      # 32 uppercase hex characters
    sha256: str   # 64 uppercase hex characters
    sha512: str | None = None  # 128 uppercase hex characters, optional
```

NameObject

```
@dataclass
class NameObject:
    """Name with associated hash digests (§5.2.2)."""

    text: str | None       # Filename including extension, or null
    hashes: HashSet | None  # Hashes of UTF-8 bytes of text, or null

    def __post_init__(self) -> None:
        """Enforce co-nullability: text and hashes are both null or both populated."""
        if (self.text is None) != (self.hashes is None):
            raise ValueError("NameObject.text and .hashes must be co-null")
```

The `__post_init__` validation enforces the co-nullability invariant from §5.2.2. This is one of the few validation checks performed at construction time — it catches a common construction error (providing a name without hashes, or vice versa) immediately rather than at serialization time.

SizeObject

```
@dataclass
class SizeObject:
    """File size in human-readable and machine-readable forms (§5.2.3)."""

    text: str   # e.g., "15.28 MB", "135 B"
    bytes: int  # Exact byte count, >= 0
```

TimestampPair

```
@dataclass
class TimestampPair:
    """Single timestamp in ISO 8601 and Unix millisecond forms (§5.2.4)."""

    iso: str   # ISO 8601 with timezone offset
    unix: int  # Milliseconds since epoch
```

TimestampsObject

```
@dataclass
class TimestampsObject:
    """Three standard filesystem timestamps (§5.2.5)."""

    created: TimestampPair
    modified: TimestampPair
    accessed: TimestampPair
```

ParentObject

```
@dataclass
class ParentObject:
    """Parent directory identity and name (§5.2.6)."""

    id: str          # x-prefixed directory ID
    name: NameObject
```

FileSystemObject

```
@dataclass
class FileSystemObject:
    """Filesystem location fields (§5.6)."""

    relative: str                # Forward-slash-separated relative path
    parent: ParentObject | None  # Null for root of single-file index
```

AttributesObject

```
@dataclass
class AttributesObject:
    """Item attributes (§5.8)."""

    is_link: bool     # Whether the item is a symbolic link
    storage_name: str  # Deterministic name for rename operations
```

MetadataEntry

```
@dataclass
class MetadataEntry:
    """A single metadata record associated with an IndexEntry (§5.10)."""

    id: str              # z-prefixed (generated) or y-prefixed (sidecar)
    origin: str          # "generated" or "sidecar"
    name: NameObject
    hashes: HashSet
    attributes: MetadataAttributes
    data: Any            # JSON object, string, array, or null

    # Sidecar-only fields (absent for generated entries)
    file_system: FileSystemObject | None = None
    size: SizeObject | None = None
    timestamps: TimestampsObject | None = None
```

MetadataAttributes

```
@dataclass
class MetadataAttributes:
```

```
    """Classification and format info for a MetadataEntry (§5.10)."""

    type: str                       # e.g., "exiftool.json_metadata", "description"
    format: str                     # "json", "text", "base64", or "lines"
    transforms: list[str]           # Ordered transformation identifiers
    source_media_type: str | None = None  # MIME type of original source data
```

`ProgressEvent`

Defined in `core/progress.py`. A lightweight event object used by `build_directory_entry()` to report progress to callers via the `progress_callback` parameter. This type is part of the public API because GUI and CLI consumers construct callbacks that receive it.

```
@dataclass
class ProgressEvent:
    """Progress report emitted during directory indexing."""

    phase: str                  # "discovery", "processing", "output", "cleanup"
    items_total: int | None     # Total items discovered; None during discovery phase
    items_completed: int        # Items processed so far (0 during discovery)
    current_path: Path | None   # Item currently being processed, or None
    message: str | None         # Optional human-readable log message
    level: str                  # Log level: "info", "warning", "error", "debug"
```

Phase values:

Phase	When emitted	`items_total`	`items_completed`
`"discovery"`	After `list_children()` returns for each directory level.	`None` until the top-level directory's children are fully enumerated, then the count of all immediate children (files + subdirectories). For recursive mode, the total grows as child directories are entered.	`0`
`"processing"`	After each child item's entry construction completes (or is skipped due to error).	Known count from discovery.	Incrementing count.
`"output"`	During serialization and output writing (Stage 5).	Same as final `items_total`.	Same as final `items_completed`.
`"cleanup"`	During post-processing (Stage 6): MetaMergeDelete file removal, final logging.	Same.	Same.

Threading safety: The callback is invoked on the indexing thread (the background thread in the GUI, the main thread in CLI/API use). The callback implementation MUST be non-blocking. GUI consumers should enqueue the event into a `queue.Queue` for main-thread processing rather than touching widgets directly. CLI consumers using `tqdm` or `rich` can update the progress bar directly since both libraries are thread-safe for single-bar updates.

Callback errors: If the callback raises an exception, the indexing engine logs a warning and continues processing. A broken progress callback does not abort the indexing operation.

9.5. Programmatic Usage Examples

These examples demonstrate the primary usage patterns for the Python API. They are illustrative — not the exact implementation — and assume the library is installed via `pip install -e ".[dev]"`.

Basic: index a single file with defaults

```
from pathlib import Path
from shruggie_indexer import index_path, serialize_entry

entry = index_path(Path("/path/to/photo.jpg"))
print(entry.id)          # "yA8A8C089A6A8583B24C85F5A4A41F5AC"
print(entry.type)        # "file"
print(entry.size.text)   # "3.45 MB"
print(entry.name.text)   # "photo.jpg"

# Serialize to JSON
json_str = serialize_entry(entry)
print(json_str)
```

Custom configuration: SHA-256 with metadata extraction

```
from pathlib import Path
from shruggie_indexer import index_path, load_config, serialize_entry

config = load_config(overrides={
    "id_algorithm": "sha256",
    "extract_exif": True,
    "meta_merge": True,
})

entry = index_path(Path("/path/to/media/folder"), config)

# The entry is a directory with nested children
print(entry.type)             # "directory"
print(len(entry.items))       # Number of children

# Access a child file's metadata
child = entry.items[0]
if child.metadata:
    for meta in child.metadata:
        print(f"{meta.origin}: {meta.attributes.type}")
```

Batch indexing: process multiple directories with shared config

```
from pathlib import Path
from shruggie_indexer import index_path, load_config, serialize_entry

config = load_config(overrides={"compute_sha512": True})
targets = [Path("/data/set-a"), Path("/data/set-b"), Path("/data/set-c")]

for target in targets:
    entry = index_path(target, config)
    json_str = serialize_entry(entry, indent=None)  # compact JSON
    output_path = target / "index.json"
    output_path.write_text(json_str, encoding="utf-8")
```

Low-level: index a single file without target classification

```
from pathlib import Path
from shruggie_indexer import build_file_entry, load_config, serialize_entry

config = load_config()
path = Path("/path/to/document.pdf").resolve()

entry = build_file_entry(path, config)
print(entry.hashes.md5)     # "A8A8C089A6A8583B24C85F5A4A41F5AC"
print(entry.hashes.sha256)  # "E3B0C44298FC1C149AFBF4C8996FB924..."
```

Configuration from a TOML file

```
from pathlib import Path
from shruggie_indexer import index_path, load_config

config = load_config(config_file=Path("my-config.toml"))
entry = index_path(Path("."), config)
```

Inspecting and filtering the entry tree

```
from shruggie_indexer import IndexEntry

def find_large_files(entry: IndexEntry, threshold_bytes: int) -> list[IndexEntry]:
    """Walk the entry tree and find files exceeding the size threshold."""
```

```
    results = []
    if entry.type == "file" and entry.size.bytes > threshold_bytes:
        results.append(entry)
    if entry.items:
        for child in entry.items:
            results.extend(find_large_files(child, threshold_bytes))
    return results

# Usage
entry = index_path(Path("/path/to/archive"))
large_files = find_large_files(entry, threshold_bytes=100_000_000)  # > 100 MB
for f in large_files:
    print(f"{f.name.text}: {f.size.text}")
```

Progress reporting and cancellation

```
import threading
from pathlib import Path
from shruggie_indexer import index_path, load_config, ProgressEvent

config = load_config(overrides={"extract_exif": True})
cancel = threading.Event()

def on_progress(event: ProgressEvent) -> None:
    if event.phase == "processing" and event.items_total:
        pct = event.items_completed / event.items_total * 100
        print(f"\r{event.items_completed}/{event.items_total} ({pct:.0f}%)", end="")
    if event.level == "warning" and event.message:
        print(f"\n  WARN: {event.message}")

try:
    entry = index_path(
        Path("/path/to/large/archive"),
        config,
        progress_callback=on_progress,
        cancel_event=cancel,
    )
    print(f"\nDone: {entry.id}")
except IndexerCancellationError:
    print("\nCancelled by user.")
```

To cancel from another thread (e.g., a GUI button handler), call `cancel.set()`. The engine will stop at the next item boundary.

Error handling

```
from pathlib import Path
from shruggie_indexer import index_path, load_config
from shruggie_indexer.core.entry import IndexerTargetError, IndexerRuntimeError
from shruggie_indexer.config.loader import IndexerConfigError

try:
    config = load_config(overrides={"id_algorithm": "blake2"})
except IndexerConfigError as e:
    print(f"Configuration error: {e}")
    # "id_algorithm must be 'md5' or 'sha256', got 'blake2'"

try:
    entry = index_path(Path("/nonexistent/path"))
except IndexerTargetError as e:
    print(f"Target error: {e}")
    # "Target does not exist: /nonexistent/path"
```

Exception hierarchy

The library defines a small exception hierarchy for programmatic error handling. All exceptions inherit from a common base class to enable catch-all handling when fine-grained distinction is unnecessary.

```
class IndexerError(Exception):
    """Base class for all shruggie-indexer exceptions."""

class IndexerConfigError(IndexerError):
    """Configuration is invalid or cannot be loaded."""

class IndexerTargetError(IndexerError):
    """Target path is invalid, inaccessible, or unclassifiable."""

class IndexerRuntimeError(IndexerError):
    """Unrecoverable error during indexing execution."""

class RenameError(IndexerError):
    """File rename failed (collision, permission, etc.)."""

class IndexerCancellationError(IndexerError):
    """The indexing operation was cancelled by the caller."""
```

These exceptions are defined in `exceptions.py` within the `shruggie_indexer` package. The top-level `__init__.py` re-exports the full hierarchy so that consumers can import directly from `shruggie_indexer` or from `shruggie_indexer.exceptions`. `IndexerCancellationError` is also available from the top-level namespace for convenience.

The exception hierarchy maps directly to the CLI exit codes (§8.10): `IndexerConfigError` → exit code 2, `IndexerTargetError` → exit code 3, `IndexerRuntimeError` → exit code 4, `IndexerCancellationError` → exit code 5 (`INTERRUPTED`). `RenameError` is a subclass of `IndexerRuntimeError` for exit code purposes but is distinct for programmatic callers who want to handle rename failures specifically. `IndexerCancellationError` is raised when the `cancel_event` parameter (available on `index_path()` and `build_directory_entry()`) is set by a caller. In the GUI, the cancel event is set by the Cancel button (§10.5). In the CLI, it is set by the `SIGINT` handler when the user presses `Ctrl+C` (§8.11).

> **Historical note:** The original communicates errors through `Vbs` log messages and PowerShell's unstructured error stream. There are no typed exceptions, no error hierarchy, and no programmatic way to distinguish between "target not found," "configuration invalid," and "runtime failure." The tool's typed exceptions enable callers to implement precise error recovery strategies.

10. GUI Application

This section defines the standalone desktop GUI for `shruggie-indexer` — a visual frontend to the same library code consumed by the CLI (§8) and the public API (§9). The GUI uses a CustomTkinter foundation, dark theme, consistent font stack, and two-panel layout pattern. Where this specification does not explicitly define a visual convention — such as padding values, border radii, or widget spacing — the `shruggie-feedtools` GUI (§1.5, External References) serves as the normative reference for project-family consistency.

The GUI serves a fundamentally different user need than the CLI. The CLI is a power-user and automation tool — its flag-based interface composes naturally in scripts but requires the user to internalize the full option space, understand the implication chain (`--meta-merge-delete` → `--meta-merge` → `--meta`), and manage output routing mentally. The GUI eliminates this cognitive overhead by consolidating the three operation types (Index, Meta Merge, Meta Merge Delete) into a single Operations page with an inline selector, presenting the rename feature as an optional toggle that can apply to any operation, and always displaying all controls with context-sensitive enable/disable logic and safe defaults pre-applied. The GUI is the recommended entry point for users who are unfamiliar with the tool's option space or who want visual confirmation of their configuration before executing.

Module location: `gui/app.py` (§3.2). The GUI is a single-module implementation for the MVP. If the GUI grows in complexity (custom widgets, asset files, reusable components), additional modules and an `assets/` subdirectory can be added under `gui/` without restructuring (§3.2).

Dependency isolation: The `customtkinter` dependency is declared as an optional extra (`pip install shruggie-indexer[gui]`) and is imported only within the `gui/` subpackage. No module outside `gui/` imports from it. The core library, CLI, and public API function without any GUI dependency installed (design goal G5, §2.3). If the user launches the GUI entry point without `customtkinter` installed, the application MUST fail with a clear error message directing the user to install the dependency (`pip install shruggie-indexer[gui]`).

> **Updated 2026-02-23:** Reflects tab consolidation (Operations/Settings/About model), destructive operation indicator, auto-suggest output paths, dual browse buttons for auto mode, output panel enhancements (auto-clear, Clear button, copy feedback, resizable panel), About tab, version label in sidebar, tooltips, and corrected post-job display behavior.

10.1. GUI Framework and Architecture

Framework selection

The GUI uses CustomTkinter as its widget toolkit, consistent with `shruggie-feedtools`. CustomTkinter wraps Python's standard `tkinter` library with modern-styled widgets, dark/light theme support, and DPI scaling. It requires no additional runtime beyond Python itself and `tkinter` — both of which are available in standard CPython distributions and in PyInstaller bundles.

The choice of CustomTkinter over alternatives (PyQt, wxPython, Kivy) is inherited from the ShruggieTech project family and driven by three factors: zero licensing restrictions (MIT-licensed, compatible with Apache 2.0), no binary compilation step (pure Python distribution), and minimal bundle size impact in PyInstaller builds. This decision is not reconsidered here — the `shruggie-feedtools` GUI has validated the framework for this class of desktop tool.

Architectural role

The GUI is a thin presentation layer over the library API — identical in architectural role to the CLI (design goal G3, §2.3). The boundary is explicit:

- The GUI constructs `IndexerConfig` objects via `load_config()` (§9.3), translating widget state into configuration overrides.
- The GUI calls `index_path()` (§9.2) to perform indexing, receiving an `IndexEntry` in return.
- The GUI calls `serialize_entry()` (§9.2) to convert the entry to JSON for display.
- The GUI handles output routing (writing to files) using the same logic as the CLI, but through user-driven save actions rather than automatic flag-based routing.

No indexing logic lives in the GUI module. The GUI does not import from `core/` directly — it interacts exclusively through the public API surface defined in §9.1. This constraint ensures that the GUI cannot introduce behavioral divergence from the CLI or the library.

Threading model

All indexing operations run in a background thread to keep the UI responsive. The GUI uses Python's `threading.Thread` for background execution, not `asyncio` or multiprocessing. This matches the `shruggie-feedtools` threading model and avoids the complexity of cross-thread tkinter access patterns that arise with process-based parallelism.

The threading contract:

1. The main thread owns all widget state. Only the main thread creates, reads, or modifies tkinter widgets.
2. Background threads perform indexing via `index_path()` and communicate results back to the main thread through `tkinter`'s `after()` method (polling a thread-safe queue) or by setting a `threading.Event` that the main thread checks on a timer.
3. Background threads MUST NOT touch any widget directly. Widget updates — progress messages, output population, button re-enabling — are always dispatched to the main thread via `widget.after(0, callback)` or an equivalent event-loop-safe mechanism.
4. Only one background indexing thread may be active at any time (see §10.5, job exclusivity). The GUI enforces this by disabling all operation tabs and action buttons when a job is in flight.

> **Historical note:** The `shruggie-feedtools` GUI uses a simpler fire-and-forget `threading.Thread` model sufficient for its shorter-lived operations. The `shruggie-indexer` GUI requires the more sophisticated model described above because indexing operations can be long-running (minutes for large directory trees), cancellable, and produce incremental progress data.

Session persistence

The GUI persists user session settings — window geometry, last-used operation tab, per-tab input values, and settings panel preferences — to a JSON file in the platform-appropriate application data directory. This ensures the GUI reopens in the state the user left it.

Session file location:

Platform	Path
Windows	`%APPDATA%\shruggie-indexer\gui-session.json`
Linux	`~/.config/shruggie-indexer/gui-session.json`
macOS	`~/Library/Application Support/shruggie-indexer/gui-session.json`

This uses the same platform directory resolution as the indexer's TOML configuration file (§3.3), under the same `shruggie-indexer` application directory. The session file is a separate file from the TOML configuration — it stores GUI-specific presentation state, not indexing configuration. The TOML file governs indexing behavior; the session file governs window preferences.

Session file format: Plain JSON with a `session_version` discriminator for forward compatibility. The file is written on application exit (window close) and on each successful operation completion. It is read once at startup. If the file does not exist, is unreadable, or has an unrecognized `session_version`, the GUI starts with default values and does not produce an error — the file is purely a convenience optimization.

```
{
  "session_version": "2",
  "window": {
    "geometry": "1100x750+200+100",
    "active_tab": "operations"
  },
  "operations": {
    "operation_type": "index",
    "target_path": "/path/to/last/target",
    "target_type": "auto",
    "recursive": true,
    "id_algorithm": "md5",
    "sha512": false,
    "extract_exif": false,
    "rename": false,
    "dry_run": true,
```

```
    "inplace": true,
    "output_mode": "view",
    "output_file": ""
  },
  "settings": {
    "id_algorithm": "md5",
    "sha512": false,
    "indent": "2",
    "verbosity": "normal",
    "tooltips": true,
    "config_file": ""
  },
  "output_panel_height": 250
}
```

> **Updated 2026-02-23:** The session format was revised from `session_version: "1"` (per-tab state with separate `tabs` dictionary) to `session_version: "2"` (single `operations` state object) to reflect the tab consolidation. The application performs backward-compatible migration when encountering a version 1 session file. The `output_panel_height` field was added to persist the resizable output panel height.

Session state: Unlike the previous per-tab design, the consolidated Operations page maintains a single set of input state. The selected operation type is stored in the `operations.operation_type` field. Switching operation types within the Operations page updates the visible controls but the shared fields (target path, type, recursive, ID algorithm, SHA-512) persist across operation type changes.

> **Historical note:** Session persistence is a `shruggie-indexer`-specific addition. The `shruggie-feedtools` GUI starts fresh every launch, which is acceptable for its simpler input space. The indexer's more complex configuration surface (target paths, multiple flag combinations, per-operation preferences) makes persistence substantially more valuable.

Module structure

For the MVP, the GUI is a single file: `gui/app.py`. This file contains the `ShruggiIndexerApp` class (the top-level application window), all tab frame classes, the progress display logic, and the session persistence code. The `gui/__init__.py` file exports a single `main()` function that instantiates and runs the application.

```python
# gui/__init__.py
def main():
    """Launch the shruggie-indexer GUI application."""
    try:
        from shruggie_indexer.gui.app import ShruggiIndexerApp
    except ImportError as e:
        print(
            "The GUI requires customtkinter. "
            "Install it with: pip install shruggie-indexer[gui]",
            file=sys.stderr,
        )
        sys.exit(1)
    app = ShruggiIndexerApp()
    app.mainloop()
```

If the GUI outgrows a single file during implementation, the recommended decomposition is: `app.py` (main window and tab container), `tabs/` subdirectory with one module per operation tab, `widgets/` for reusable custom widgets (progress bar, JSON viewer), and `session.py` for persistence logic. This decomposition is optional for the MVP — the single-file approach is acceptable if the file remains under ~1500 lines.

10.2. Window Layout

> **Updated 2026-02-23:** Reflects tab consolidation per Pending Updates document, Section 2, item 2.1. The four separate operation tabs (Index, Meta Merge, Meta Merge Delete, Rename) have been consolidated into a single Operations page with an inline operation type selector. The sidebar now contains three entries: Operations, Settings, and About.

Overall structure

The window uses a two-panel layout: a narrow left sidebar for page navigation and a main working area on the right. This is the same structural pattern used by `shruggie-feedtools`, adapted for the indexer's consolidated operations architecture.

```
┌──────────────────────────────────────────────────────────────┐
│  Shruggie Indexer                                  [−][□][×] │
├──────────────┬───────────────────────────────────────────────┤
│              │  Operations                                   │
│  ┌────────┐  │  ┌ Operation ──────────────────────────────┐  │
```

Window properties

Property	Value	Notes
Window title	`"Shruggie Indexer"`	No version in the title bar — keeps it clean. Version is visible in the sidebar and on the About page.
Minimum size	`1000×700`	Slightly larger than `shruggie-feedtools` (900×600) to accommodate the additional input complexity.
Default size	`1100×750`	Restored from session file if available.
Resizable	Yes	The output panel expands to fill available vertical space. The sidebar width is fixed.
Appearance mode	Dark	`customtkinter.set_appearance_mode("dark")`. Matches `shruggie-feedtools`.
Color theme	Default CustomTkinter dark	No custom theme for the MVP.

Sidebar

The left sidebar is a fixed-width panel (160px) containing vertically stacked navigation buttons for the three application pages and a version label at the bottom. The sidebar background uses the CustomTkinter frame default for the dark theme.

Each sidebar button is a `CTkButton` styled as a navigation element — the active page's button is visually distinguished (highlighted background, bold text) from inactive pages. Clicking a sidebar button switches the visible frame in the working area without destroying or recreating the frame — the frame is hidden/shown via `pack_forget()` / `pack()` (or equivalent grid management), preserving all widget state.

Sidebar navigation order (top to bottom):

Button label	Page identifier	Description
Operations	`operations`	Consolidated operations page with inline operation type selector, target input, configuration options, and output controls.
Settings	`settings`	Application preferences and persistent configuration.
About	`about`	Project information, version details, and links.

> **Updated 2026-02-23:** The previous sidebar layout contained five operation buttons (Index, Meta Merge, Meta Merge Delete, Rename, plus a separator, plus Settings). The consolidated layout replaces all four operation buttons with a single "Operations" button and adds an "About" button, reducing sidebar clutter and better reflecting the underlying architecture — the four operations are four preset configurations of the same `index_path()` pipeline, not four independent features.

Version label. Below all sidebar buttons, a small, muted-font label displays the application version string (from `shruggie_indexer.__version__`). This provides at-a-glance version identification without consuming valuable UI space.

Working area

The working area occupies the remaining space to the right of the sidebar. It contains a frame container that holds one frame per page. Only the active page's frame is visible at any time. Each page frame is instantiated once at application startup and persists for the lifetime of the application — page switches show and hide pre-built frames, they do not reconstruct them.

The Operations page uses a vertical layout structure:

1. **Input section** (top, scrollable) — Operation type selector, target path, operation-specific options, and output configuration, organized into labeled group frames (§10.3).
2. **Action button** (pinned below input section) — A single prominently-styled button to execute the selected operation.
3. **Drag handle** — A thin horizontal bar for resizing the output panel (§10.6).
4. **Output section** (bottom, resizable) — The JSON viewer, log stream, and toolbar (Clear, Copy, Save buttons).

The Settings page has its own layout (§10.4) and does not include an action button or output section. The About page has a static informational layout (§10.8.2).

Page frame architecture

The Operations page is implemented as the `OperationsPage` class inheriting from `customtkinter.CTkFrame`. It contains all operation controls, always visible, with context-sensitive enable/disable logic based on the selected operation type. The Settings and About pages are implemented as `SettingsTab` and `AboutTab` classes, respectively.

The output panel is a single shared `OutputPanel` widget instance that is always visible below the Operations page input area. It displays the result of the most recent operation regardless of which operation type initiated it. The panel is cleared at the start of each new operation.

> **Architectural decision: shared output panel.** A per-operation output panel would allow users to run an Index operation, view the output, switch to Rename, run that, and switch back to see the Index output still present. However, this introduces memory pressure for large outputs (index trees can produce megabytes of JSON), complicates the "one job at a time" invariant (§10.5), and adds visual confusion about which output corresponds to which operation. The shared output panel is the simpler, more predictable design — it always shows the result of the most recent operation. The panel is cleared at the start of each new operation, matching `shruggie-feedtools` behavior.

10.3. Target Selection and Input

> **Updated 2026-02-23:** Rewritten to reflect the consolidated Operations page model. The previous version described per-tab input forms for four separate operation tabs. The current implementation uses a single Operations page with an inline operation type selector, shared target input, and context-sensitive option controls.

The Operations page presents all input controls in a single scrollable area, organized into labeled group frames (`_LabeledGroup` — a `CTkFrame` with a bold header label, optional description, and content area). Controls that are not relevant to the currently selected operation type are hidden or disabled rather than shown in a flat list.

Visual grouping

All controls are organized into four labeled groups, displayed in order from top to bottom:

Group	Label	Description	Contents
1	Operation	Select the indexing operation to perform.	Operation type dropdown and destructive operation indicator (§10.8.1).
2	Target	Choose the file or directory to index.	Path entry, browse buttons, type radio group, recursive checkbox.
3	Options	Configure indexing parameters.	ID algorithm, SHA-512, EXIF, rename toggle, dry run, in-place — all always visible, disabled when not applicable with explanatory fine-print labels.
4	Output	Control where results are written.	Output mode radios and output file path field — always visible, disabled with explanation when not applicable.

Each group uses consistent padding and alignment. Section headers are bold with an optional one-line description in a muted font beneath. **All controls are always visible** — controls that do not apply to the current operation or target configuration are *disabled* (greyed-out) with small explanatory labels beneath them, never hidden.

Operation type selector

> **Updated 2026-02-23:** Rename is no longer listed as an independent operation type. It is an optional feature toggle that can be combined with any of the three core operations. See §10.3, context-sensitive options.

The Operation group contains a `CTkOptionMenu` dropdown for selecting between the three core operation types:

Label	Internal key	Configuration overrides

Label	Internal key	Configuration overrides
Index	`index`	Base indexing; optional EXIF extraction.
Meta Merge	`meta_merge`	`extract_exif=True`, `meta_merge=True`.
Meta Merge Delete	`meta_merge_delete`	`extract_exif=True`, `meta_merge=True`, `meta_merge_delete=True`.

The Rename feature is exposed as a separate "Rename files" checkbox in the Options group (§10.3, context-sensitive options) that can be combined with any of the three core operations. When enabled, it adds `rename=True` and optionally `dry_run=True` to the configuration overrides.

Changing the operation type immediately updates the enabled/disabled state of all controls, the destructive operation indicator, and the action button state. The selected operation type is persisted in the session file.

Target path widget group

The Target group contains the following controls:

Target entry field. A `CTkEntry` widget for the filesystem path. The user can type a path directly or use the Browse buttons to open a dialog. The field supports drag-and-drop of files and directories from the system file manager where the platform supports it. Drag-and-drop is a SHOULD requirement.

Browse buttons. The browse button behavior adapts to the Type radio selection:

Type value	Browse button(s)	Behavior
Auto	Two buttons: "File…" and "Folder…"	The user can choose either a file picker or a directory picker. This replaces the previous single-button behavior that defaulted to file-only.
File	Single "Browse" button	Opens a file picker (`askopenfilename`).
Directory	Single "Browse" button	Opens a directory picker (`askdirectory`).

> **Updated 2026-02-23:** The previous implementation opened only a file picker when Type was set to "Auto", preventing users from browsing to directories. The dual-button approach provides explicit access to both picker types without requiring a custom dialog or platform-specific workarounds.

After selection, the path is populated into the Target entry field and the output file auto-suggest is triggered (§10.3, output file auto-suggest).

Type radio buttons. Three options: Auto (default), File, Directory. These map directly to the CLI's target type disambiguation (§8.2): Auto infers the type from the filesystem, File forces `--file`, Directory forces `--directory`. The Auto option is pre-selected by default.

Recursive checkbox. A `CTkCheckBox` for enabling/disabling recursive traversal. Default: checked (matching `IndexerConfig.recursive` default, §7.2). This checkbox is only meaningful when the target is a directory — when "File" is explicitly selected in the Type radio group, or when the target path resolves to a file while Type is "Auto", the Recursive checkbox is visually dimmed (disabled) with a fine-print label: *"Recursive is not applicable when the target is a single file."*

Target / Type validation

> **Added 2026-02-23:** Real-time validation of the target path against the selected Type radio to prevent user confusion and invalid operation execution.

The GUI validates the combination of the Target path and the Type radio selection in real time (on key-release and focus-out of the path field, and on Type radio change). If the combination is invalid, a red fine-print error label appears below the path field, and the START button is disabled.

Conflict	Error message
Target path points to a file, but Type is "Directory"	*"Target appears to be a file, but Type is set to "Directory". Change the target or select a different Type."*
Target path points to a directory, but Type is "File"	*"Target appears to be a directory, but Type is set to "File". Change the target or select a different Type."*

Target kind detection uses the following heuristic:

1. If the path exists on disk, use `Path.is_dir()` / `Path.is_file()`.
2. If the path does not exist: trailing `/` or `\` → directory; has a file extension → file; otherwise indeterminate (no validation error).

The START button remains disabled for the duration of the validation conflict. The user must either change the target path or select a compatible Type to proceed.

Context-sensitive options

> **Updated 2026-02-23:** All controls are always visible. Controls that do not apply to the current operation are *disabled* with explanatory fine-print labels — they are never hidden. Rename is now an optional feature toggle, not a standalone operation type.

The Options group contains all operation-related controls, always visible. Controls that are not applicable to the current operation/target combination are disabled (greyed-out) with a small explanatory label beneath them.

Always enabled:

Control	Type	Default	Maps to
ID Algorithm	`CTkComboBox` (`md5`, `sha256`)	`md5`	`IndexerConfig.id_algorithm`

Conditionally enabled/disabled (always visible):

Control	Enabled when	Disabled info text	Type	Default	Maps to
Compute SHA-512	Settings → SHA-512 default is *off*	*"Forced on by Settings → 'Compute SHA-512 by default'."*	`CTkCheckBox`	Unchecked	`IndexerConfig.compute_sha512`
Extract EXIF metadata	Index (always for Meta Merge / Meta Merge Delete)	*"EXIF extraction is always enabled for Meta Merge operations."*	`CTkCheckBox`	Unchecked	`IndexerConfig.extract_exif`
Rename files	Always	—	`CTkCheckBox`	Unchecked	`IndexerConfig.rename`
Dry run (preview only)	Rename files is checked	*"Enable 'Rename files' to configure dry-run mode."*	`CTkCheckBox`	**Checked**	`IndexerConfig.dry_run`
Write in-place sidecar files	Index, Meta Merge (user-controlled); Meta Merge Delete (forced ON, disabled)	*"In-place output is required for Meta Merge Delete because original sidecar files are deleted after merging."*	`CTkCheckBox`	Checked	`IndexerConfig.output_inplace`

SHA-512 settings synchronization:

When the Settings tab's "Compute SHA-512 by default" checkbox is checked, the Operations page "Compute SHA-512" checkbox is force-checked and disabled, with a fine-print label explaining: *"Forced on by Settings → 'Compute SHA-512 by default'."* This ensures the user understands the override source and cannot accidentally uncheck it without first changing the Settings preference.

Rename feature toggle:

Rename is an optional feature that can be combined with any of the three core operations (Index, Meta Merge, Meta Merge Delete). When the "Rename files" checkbox is enabled, the dry-run checkbox becomes active (defaults to checked). When rename is enabled *and* dry-run is disabled, the destructive indicator turns red.

> **Design note:** The CLI treats `--dry-run` as opt-in (default off). The GUI inverts this to default on when rename is enabled. This is a deliberate UX decision: the GUI user is more likely to be exploring the tool's behavior and less likely to understand the implications of an irreversible rename.

Output controls

> **Updated 2026-02-23:** Output controls are always visible. Controls that do not apply are disabled with explanatory fine-print labels.

All output controls are always visible. Controls that are not applicable to the current operation are disabled with fine-print explanation labels:

Operation	Output mode radios	Output file field	Info text
Index, **Meta Merge**	Enabled (all three modes)	Enabled when mode is "Save to file" or "Both"; disabled otherwise with: *"Select 'Save to file' or 'Both' to specify an output file."*	—
Meta Merge Delete	Disabled (locked to "Save to file")	Enabled (mandatory)	*"Meta Merge Delete always saves to a file (output file required)."*

Output file auto-suggest

> **Added 2026-02-23:** Output file auto-suggestion based on the target path and the application's naming conventions.

When the output file field is visible. the GUI auto-populates it with a conventional output filename whenever the target path changes (on focus-out or after browse selection). The auto-suggested value is editable — it is a default, not a constraint.

Naming convention rules (from §6.5 and §6.9):

Target type	Output file pattern	Example
File	`{parent_dir}/{filename}_meta2.json`	Target: `C:/my/cool/file.txt` → Output: `C:/my/cool/file.txt_meta2.json`
Directory	`{directory_path}_directorymeta2.json`	Target: `C:/my/cool/dir` → Output: `C:/my/cool/dir_directorymeta2.json`

Edge case handling:

Target path	Behavior
`C:/`	Strip trailing separator, produce `C:/_directorymeta2.json`.
`/` (Unix root)	Fall back to user's home directory: `~/root_directorymeta2.json`.

The auto-suggest respects manual edits. If the user has manually changed the output field to a value different from the last auto-generated path, subsequent target changes do not overwrite the manual edit.

Centralized state reconciliation

> **Added 2026-02-23:** Implements §10.9.4 (Control Interdependency Transparency).

All control dependency logic is centralized in a single method, `_reconcile_controls()`, on the `OperationsPage` class. This method:

1. Reads the current values of all controls (operation type, target type, target path, recursive, output mode, dry-run, rename, in-place).
2. Evaluates the dependency rules defined in the context-sensitive options, recursive toggle, and output controls subsections above.
3. Sets the enabled/disabled/visible state, default value, and info-label text of every dependent control.
4. Updates the destructive operation indicator (§10.8.1).
5. Updates the output file placeholder text based on the current target and operation configuration.

`_reconcile_controls()` is called:

- On initial page construction.
- Whenever any input control value changes (via `trace` callbacks on control variables and event bindings on entry widgets).
- After restoring session state at application startup.

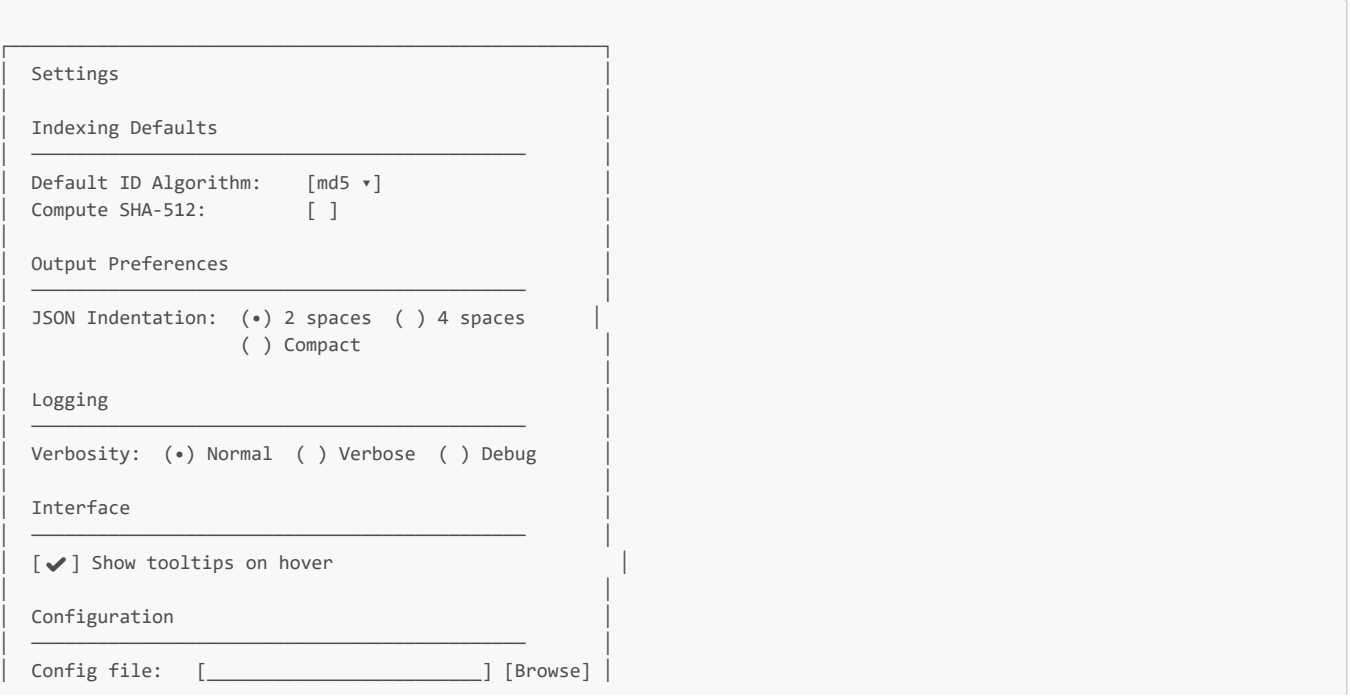
All existing per-widget callback logic that formerly evaluated dependencies independently has been replaced with calls to `_reconcile_controls()`. This ensures the UI never enters an inconsistent state where one control's appearance contradicts another's value.

10.4. Configuration Panel

The Settings page is accessed via the sidebar and provides a persistent configuration interface for preferences that apply across all operations and across sessions. Unlike the Operations page, the Settings page has no action button and no output panel — it is a pure configuration surface.

Settings layout

> **Updated 2026-02-23:** The About section previously embedded at the bottom of the Settings panel has been promoted to its own sidebar tab (§10.8.2). The Interface section has been added to control tooltip visibility.



```
  (Optional. Overrides compiled defaults.)
  ┌──────────────────┐  ┌──────────────────────┐
  │ Reset to Defaults │  │ Open Config Folder   │
  └──────────────────┘  └──────────────────────┘
```

Settings fields

Section	Field	Type	Default	Behavior
Indexing Defaults	Default ID Algorithm	`CTkComboBox`	`md5`	Pre-populates the ID Algorithm selector on the Operations page.
Indexing Defaults	Compute SHA-512	`CTkCheckBox`	Unchecked	When checked, force-enables and disables the SHA-512 checkbox on the Operations page with info text: "*Forced on by Settings → 'Compute SHA-512 by default'.*" The override is applied in real-time via a callback.
Output Preferences	JSON Indentation	`CTkRadioButton` group (`2 spaces`, `4 spaces`, `Compact`)	`2 spaces`	Controls the `indent` parameter passed to `serialize_entry()`. Compact produces single-line JSON. Stored in session as `"2"`, `"4"`, or `"none"`.
Logging	Verbosity	`CTkRadioButton` group (`Normal`, `Verbose`, `Debug`)	`Normal`	Maps to log level: Normal → WARNING, Verbose → INFO, Debug → DEBUG. Log output is directed to the progress display area (§10.5), not to a separate console window.
Interface	Show tooltips on hover	`CTkCheckBox`	Checked	Globally enables or disables hover tooltips (`_Tooltip.set_enabled()`). When disabled, no tooltip popups appear anywhere in the application.
Configuration	Config file	`CTkEntry` + Browse	*(empty)*	Optional path to a TOML configuration file. When set, this path is passed to `load_config(config_file=...)` for all operations. When empty, the standard resolution chain (§3.3) applies.

Settings persistence

Settings values are saved to the session file (§10.1, session persistence) when the application exits. Settings changes take effect immediately — there is no "Apply" button.

Utility buttons

Reset to Defaults. Resets all Settings fields to their compiled default values. Does NOT reset Operations page input state — only the Settings page's own fields. Presents a confirmation dialog before proceeding: "*Reset all settings to their default values?*"

Open Config Folder. Opens the platform-specific application data directory (`%APPDATA%\shruggie-indexer\` on Windows, `~/.config/shruggie-indexer/` on Linux, `~/Library/Application Support/shruggie-indexer/` on macOS) in the system file manager. This is a convenience for users who want to edit the TOML configuration file directly. If the directory does not exist, it is created before opening.

10.5. Indexing Execution and Progress

This subsection defines how the GUI executes indexing operations, displays progress, enforces job exclusivity, and supports cancellation.

Action button behavior

> **Updated 2026-02-23:** The action button now always displays "▶ START" (green, max 50% window width) regardless of the selected operation type. During execution, it changes to "■ Cancel" (red). The button label no longer reflects the operation type — the operation type is already visible in the Operation group's dropdown. This simplification reduces visual complexity and provides a clear, consistent call-to-action.

The Operations page has a single action button at the bottom of its input section, centered, with a maximum width of 50% of the application window (or 350 px, whichever is smaller). The button uses a distinct green color (`#1b8a1b` / `#22882a` for light/dark themes) to visually differentiate it from all other buttons in the application.

State	Button label	Color
Idle	▶ START	Green (`#1b8a1b` / `#22882a`)
Running	■ Cancel	Red (`#cc3333`)
Validation error	▶ START (disabled)	Grey (disabled state)

When clicked, the action button:

1. Validates the Operations page input fields (target path exists, required fields are populated).
2. Constructs an `IndexerConfig` via `load_config()` with the appropriate overrides from `OperationsPage.build_config()`.
3. Transitions the UI to the "running" state (§10.5, job exclusivity).
4. Spawns a background thread that calls `index_path()`.
5. Updates the progress display as the operation proceeds.
6. On completion, populates the output panel and transitions the UI back to the "idle" state.

If input validation fails, the action button does not spawn a thread. Instead, a validation dialog (`messagebox.showwarning`) is displayed with the validation error message.

Job exclusivity

Only one indexing operation may execute at a time. The GUI enforces this by entering a "running" state when any operation begins and exiting it when the operation completes (successfully, with error, or by cancellation).

Running state effects:

UI element	Behavior during running state
Operations sidebar button	Remains highlighted (active page) but the Operations page input controls are disabled.
Settings sidebar button	Remains clickable — users can view settings while an operation runs.
About sidebar button	Remains clickable.
Action button	Changes label to "■ Cancel" and changes color to a warning/stop color (red tint). Clicking it initiates cancellation (see below).
Input fields on Operations page	Disabled (non-editable). The user cannot modify the configuration of a running operation.
Output panel	Cleared at operation start (auto-clear). Replaced by the progress panel during execution. Restored with final output on completion.

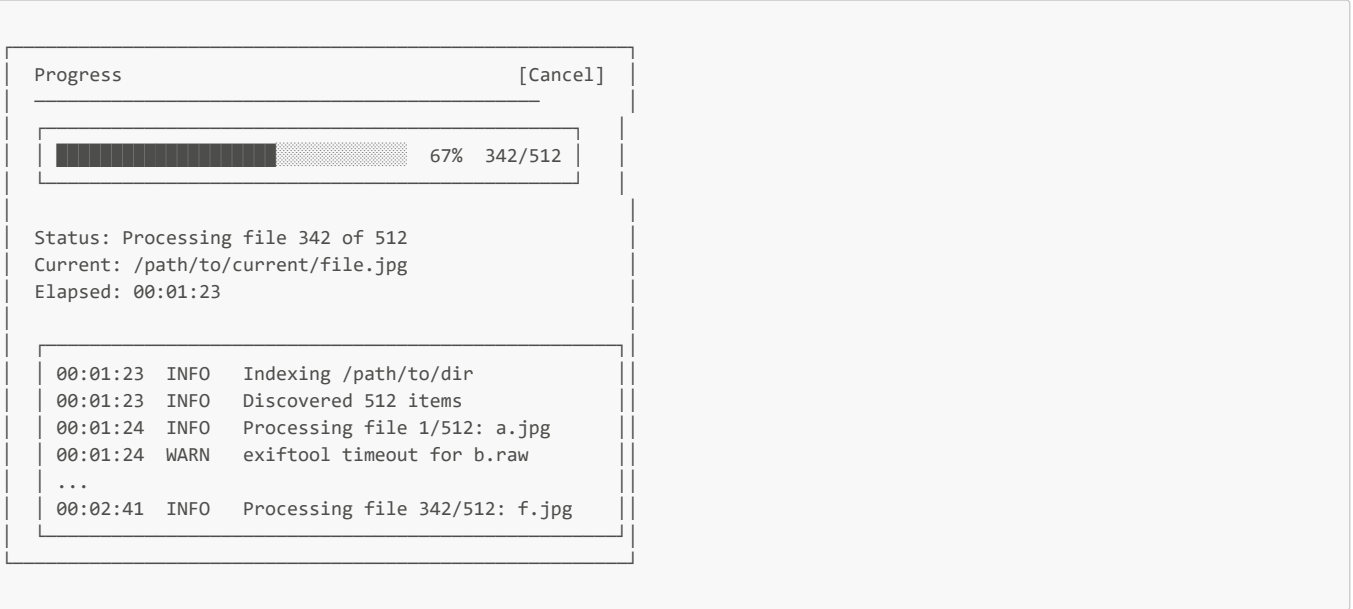
The running state is tracked by a single boolean flag (`_job_running: bool`) on the application instance. All UI transitions check this flag before allowing state changes.

Progress display

> **Updated 2026-02-23:** The progress display area is allocated as a fixed-height frame at initial window construction, per the layout stability standard (§10.9.7). The frame height accommodates the tallest state (progress bar + status text + current-file label + log stream) and does not change between idle, running, and completed states. In the idle state, the START button is centered within this pre-allocated region; the remaining space is empty but reserved. Additionally, all progress event messages (per-file "processing..." status) are forwarded to the log stream textbox in addition to updating the status and current-file labels. This ensures the log stream provides a complete sequential record of the operation.

The progress display area occupies the output panel during operation execution. It provides real-time feedback about the indexing operation's progress, replacing the blank or previous-output state of the output panel.

Progress display layout:



The progress display has three visual components:

1. Progress bar. A `CTkProgressBar` widget showing a determinate percentage when the total item count is known (directory targets after discovery), or an indeterminate animation (pulsing/marquee) when the total is not yet known (during the discovery phase of directory traversal, or for single-file targets where the operation is effectively instantaneous).

For directory targets, the progress percentage is `items_completed / total_items_discovered`. The total is known after the traversal discovery phase (Stage 3, §4.1) completes. During discovery, the progress bar shows indeterminate animation and the status line reads "Discovering items..." with a running count of items found so far. Once discovery completes, the bar switches to determinate mode and the status line shows the fraction.

2. Status summary. Three text labels displaying:

- **Status:** A one-line human-readable description of what the engine is currently doing ("Discovering items...", "Processing file 342 of 512", "Writing output...", "Deleting sidecar files...").
- **Current:** The path of the item currently being processed. Truncated with an ellipsis prefix if it exceeds the available width (`...very/long/path/to/file.jpg`).
- **Elapsed:** A running wall-clock timer in `HH:MM:SS` format, updated every second via a `tkinter.after()` timer.

3. Log stream. A scrollable, read-only `CTkTextbox` that displays log messages from the indexing engine in real time. This textbox uses the same monospaced font as the JSON output panel (JetBrains Mono → Consolas fallback). Messages are formatted as `HH:MM:SS LEVEL message` and are appended incrementally. The log stream auto-scrolls to the bottom as new messages arrive, unless the user has manually scrolled up to inspect earlier messages (auto-scroll pauses when the user scrolls away from the bottom and resumes when they scroll back to the end).

Progress callback integration: The GUI installs a progress callback on the `IndexerConfig` (or passes it as an argument to `index_path()`) that the core engine invokes at defined intervals during processing. The callback is a callable with the signature:

```
def on_progress(event: ProgressEvent) -> None:
    """Called by the indexing engine to report progress."""
```

Where `ProgressEvent` is a lightweight dataclass:

```
@dataclass
class ProgressEvent:
    phase: str           # "discovery", "processing", "output", "cleanup"
    items_total: int | None    # None during discovery
    items_completed: int
    current_path: Path | None  # The item currently being processed
    message: str | None        # Optional log-level message
    level: str                 # "info", "warning", "error", "debug"
```

The progress callback is invoked on the background thread. It MUST NOT touch widgets directly. Instead, it enqueues events into a `queue.Queue`, which the main thread drains on a 50ms `after()` timer to update the progress display widgets. This polling interval provides visually smooth progress updates without excessive main-thread load.

> **Architectural note:** The public API (§9) does not currently define a progress callback parameter on `index_path()`. The GUI will require the core engine to support this callback. The recommended approach is to add an optional `progress_callback: Callable[[ProgressEvent], None] | None = None` parameter to `index_path()` and `build_directory_entry()`, defaulting to `None` (no callback, matching CLI behavior). The callback is invoked after each item is processed. This is a GUI-driven addition to the core API that also benefits CLI users who want progress reporting via `tqdm` or `rich`. The §9 and §6 specifications should be updated to reflect this addition when the GUI sprint is implemented.

Cancellation

Long-running operations can be cancelled by clicking the "■ Cancel" button that replaces the action button during execution. Cancellation is cooperative — the indexing engine checks a cancellation flag at defined checkpoints and raises a `CancellationError` when cancellation is requested.

Cancellation mechanism:

1. The GUI creates a `threading.Event` object (`cancel_event`) before spawning the background thread.
2. The `cancel_event` is passed to the indexing engine (via the same mechanism as the progress callback — an optional parameter on `index_path()` or a field on `IndexerConfig`).
3. The engine checks `cancel_event.is_set()` at the start of each item's processing loop. If set, the engine stops processing and raises `IndexerCancellationError`.
4. The background thread catches `IndexerCancellationError` and signals completion to the main thread with a "cancelled" status.
5. The main thread updates the progress display to show "Operation cancelled" and transitions back to the idle state.

Cancellation granularity: Cancellation is per-item, not mid-item. Once the engine begins processing a single file or directory entry (hashing, exiftool invocation, etc.), that item runs to completion before the cancellation flag is checked. This ensures that partial items are never written to output — every item

in the output is complete. For single-file operations, cancellation is not meaningful (the operation is essentially atomic).

Cancellation and MetaMergeDelete: If a MetaMergeDelete operation is cancelled, NO sidecar files are deleted — the deletion queue (Stage 6, §4.1) is discarded. The output produced before cancellation (in-place sidecars, partial aggregate file) may exist on disk but the sidecar source files are preserved. This is the safe default.

Cancellation exception:

```
class IndexerCancellationError(IndexerError):
    """The operation was cancelled by the user."""
```

This exception is added to the hierarchy defined in §9.5 and maps to a new GUI-specific status rather than a CLI exit code (the CLI does not currently support mid-operation cancellation, though it could be extended to handle `SIGINT` in a future version).

Completion states

When the background thread finishes (successfully, with error, or by cancellation), it signals the main thread via the progress queue with a terminal event. The main thread then:

1. Transitions the UI back to the idle state (re-enable sidebar tabs, restore action button, re-enable input fields).
2. Updates the progress display:

Outcome	Progress bar	Status text	Log stream	Output panel
Success	100%, green tint	"Completed — N items indexed in MM:SS"	Final summary appended	Populated with JSON output
Partial failure	100%, amber tint	"Completed with N warnings — M of N items indexed"	Warning summary appended	Populated with JSON output (with degraded fields)
Error	Red tint, stopped	"Failed — `[error message]`"	Error details appended	Displays error JSON or message
Cancelled	Amber tint, stopped	"Cancelled after N of M items"	Cancellation notice appended	Empty or partial output (not displayed)

For the success and partial-failure cases, the progress display is replaced by the JSON output in the output panel after a brief delay (500ms) to allow the user to see the completion status. The user can toggle back to the progress/log view via a tab or toggle button at the top of the output panel (see §10.6).

10.6. Output Display and Export

> **Updated 2026-02-23:** Updated to document the Clear button, copy visual feedback, resizable output panel, auto-clear on new job, and corrected post-job display behavior for save-to-file mode.

Output panel

The output panel occupies the lower portion of the working area on the Operations page. It serves dual purpose: displaying the JSON result after completion and displaying log messages. A toggle at the top of the panel switches between the two views:

```
[Output]  [Log]                    [Save] [Copy] [Clear]
```

The "Output" button shows the JSON viewer. The "Log" button shows the log stream from the most recent operation (preserved after completion so the user can review warnings and timing information). The toggle defaults to "Output" after a successful operation and to "Log" after an error or cancellation. All toolbar buttons have hover tooltips (§10.8.3).

Auto-clear on new job

When a new operation begins, the output panel is automatically cleared (both JSON and log content) before the progress panel replaces it. This ensures that stale output from a previous operation is never visible when the new operation completes. The auto-clear is unconditional — it occurs regardless of output mode or operation type.

Resizable output panel

The output panel's height is adjustable via a drag handle (`_DragHandle`) positioned between the input controls and the output panel. The drag handle is a thin horizontal bar (6px, `sb_v_double_arrow` cursor) that the user clicks and drags vertically to resize the output panel.

Constraints:

Property	Value
Default height	250 px (`_DEFAULT_OUTPUT_HEIGHT`)
Minimum height	100 px (`_MIN_OUTPUT_HEIGHT`)
Maximum height	600 px (`_MAX_OUTPUT_HEIGHT`)

The current output panel height is persisted to the session file (`output_panel_height` field, §10.1) and restored on next launch. During a running operation, the drag handle and output panel are hidden and replaced by the progress panel; they are restored when the operation completes.

JSON viewer

The JSON viewer is a scrollable, read-only `CTkTextbox` configured with a monospaced font (JetBrains Mono → Consolas fallback, matching `shruggie-feedtools`). It displays the serialized JSON output from `serialize_entry()` with the indentation level configured in the Settings panel.

Syntax highlighting: The JSON viewer SHOULD apply basic syntax highlighting to improve readability. At minimum: keys in one color, string values in another, numeric values in a third, and `null`/`true`/`false` in a fourth. CustomTkinter's `CTkTextbox` supports tag-based coloring via the underlying tkinter `Text` widget's `tag_configure()` and `tag_add()` methods. A lightweight JSON token scanner (not a full parser — just regex-based token identification) applies color tags after the JSON text is loaded.

Syntax highlighting is a SHOULD requirement, not a MUST. If implementation complexity is prohibitive for the MVP, plain monospaced text without coloring is acceptable. The `shruggie-feedtools` GUI does not implement syntax highlighting in its output panel, so this would be a visual improvement over the reference implementation.

Large output handling: For large index trees (thousands of items), the JSON output can be several megabytes. The JSON viewer MUST remain responsive — it SHOULD NOT attempt to load the entire output into the textbox if doing so would cause a visible UI freeze. The recommended approach:

1. For outputs under 1 MB, load the full text into the textbox.
2. For outputs between 1 MB and 10 MB, load the text but disable syntax highlighting (tag application is the expensive operation, not text insertion).
3. For outputs over 10 MB, display a summary message in the viewer: "*Output is `[size]` — too large for inline display. Use Save to export.*" The full output is held in memory and available via the Save button.

These thresholds are approximate and SHOULD be tuned during implementation based on observed performance.

Copy button

Copies the full content of the current view (JSON output or log) to the system clipboard via `widget.clipboard_clear()` / `widget.clipboard_append()`.

Visual feedback: After a successful copy, the button text changes from "Copy" to "Copied!" with a green background color for 1.5 seconds (`_COPY_FEEDBACK_MS`), then reverts to its original appearance. This provides immediate confirmation that the clipboard operation succeeded without requiring a separate toast notification.

The Copy button is disabled when the active view (Output or Log) is empty. When viewing the Log tab, Copy is enabled if the log contains any entries, even if no JSON output exists. When viewing the Output tab, Copy is enabled if JSON output is present.

Clear button

> **Added 2026-02-23.**

Clears both JSON output and log content from the output panel. The Clear button calls `OutputPanel.clear()`, which empties the JSON text buffer, clears the log line list, and resets the textbox to an empty state. After clearing, the Copy and Save buttons are disabled (no content to act on).

The Clear button is always enabled — clearing an already-empty panel is a no-op and does not produce an error.

Save button

Opens a platform-native save-as dialog defaulting to `.json` extension and a filename derived from the target: `{target_stem}-index.json` for the most recent operation. The dialog remembers the last-used save directory across sessions (stored in the session file).

The Save button writes the full JSON output (regardless of whether the output panel is displaying a truncated preview) to the selected file using UTF-8 encoding. After a successful save, a brief toast/notification message appears near the Save button: "*Saved to {filename}*" (auto-dismisses after 3 seconds).

The Save button is disabled when the active view is empty. Like Copy, enablement is per-view: the Save button is active when viewing the Log tab if the log has content, and when viewing the Output tab if JSON output is present.

Output panel interaction with operations

The output panel is a single widget instance on the Operations page. It displays the result of the most recent operation regardless of which operation type was selected. When the user changes the operation type selector, the output panel retains its current content. A new operation clears the panel via auto-clear (§10.6, auto-clear).

Post-job display behavior by output mode:

Output mode	Display behavior
View only	JSON output is loaded into the output panel via `set_json()`.
Save to file	A status message is displayed: "*Output saved to: {filename}*". The JSON is not loaded into the viewer. Copy and Save buttons act on the active view — they are enabled for the Log tab if log content exists.
Both	JSON output is loaded into the output panel AND written to the output file.
(Meta Merge Delete)	Same as "Save to file" — mandatory output file, status message displayed.
(Rename active)	JSON preview is loaded into the output panel (same as "View only"). The rename feature is a toggle that can be active with any operation type.

This distinction ensures that users who selected "Save to file" mode see confirmation of the save rather than a confusing blank panel or redundant JSON display.

10.7. Keyboard Shortcuts and Accessibility

Keyboard shortcuts

> **Updated 2026-02-23:** `Ctrl+1` through `Ctrl+3` now switch between sidebar pages (Operations, Settings, About) instead of the former four operation tabs. `Ctrl+Shift+C` added for copy-to-clipboard.

The GUI defines a minimal set of keyboard shortcuts for common actions. Shortcuts use platform-standard modifier keys: `Ctrl` on Windows/Linux, `Cmd` on macOS.

Shortcut	Action	Notes
`Ctrl+R` / `Cmd+R`	Execute the current operation	Equivalent to clicking the action button. Disabled during running state.
`Ctrl+Shift+C`	Copy output to clipboard	Copies the current view (Output or Log) to clipboard. Uses `Shift` to avoid conflicting with standard `Ctrl+C` text copy in input fields.
`Ctrl+S` / `Cmd+S`	Save output to file	Opens the save-as dialog.
`Ctrl+.` / `Cmd+.`	Cancel running operation	Equivalent to clicking the Cancel button. No-op when idle.
`Ctrl+1`	Switch to Operations page	Navigates the sidebar to the Operations page. Disabled during running state.
`Ctrl+2`	Switch to Settings page	Navigates the sidebar to the Settings page.
`Ctrl+3`	Switch to About page	Navigates the sidebar to the About page.
`Ctrl+,` / `Cmd+,`	Open Settings	Standard "preferences" shortcut. Equivalent to `Ctrl+2`. Always available.
`Ctrl+Q` / `Cmd+Q`	Quit application	Prompts for confirmation if an operation is running.
`Escape`	Cancel running operation	Secondary cancel shortcut. No-op when idle.

Keyboard shortcuts MUST NOT conflict with standard text-editing shortcuts within input fields (`Ctrl+A` for select all, `Ctrl+V` for paste, `Ctrl+Z` for undo). The shortcuts above are chosen to avoid these conflicts.

Tab order

All interactive widgets follow a logical tab order (keyboard `Tab` key navigation) within the Operations page. The tab order proceeds top-to-bottom, left-to-right: Operation selector → Target path → Browse button(s) → Type radio group → Recursive checkbox → per-operation options → output controls → action button. The output panel is excluded from the tab order (it is read-only).

Accessibility notes

The GUI makes reasonable accommodations for keyboard-only operation but does not target formal accessibility compliance (WCAG, Section 508) for the MVP. CustomTkinter inherits `tkinter`'s native accessibility support, which varies by platform — Windows provides the best screen reader integration via UI

Automation, Linux/macOS support is limited.

Minimum requirements for MVP:

- All interactive controls are reachable via keyboard tab navigation.
- All buttons have descriptive text labels (no icon-only buttons without tooltips).
- Warning labels use both color AND text to convey their message (not color alone).
- The progress bar's percentage is exposed as text in the adjacent status label.
- Focus indicators are visible on all interactive widgets (CustomTkinter provides this by default in dark theme).

Formal accessibility improvements are a post-MVP consideration.

10.8. Supplemental GUI Components

> **Added 2026-02-23:** This subsection documents GUI components introduced during the 2026-02-23 update cycle that do not belong to any existing subsection.

10.8.1. Destructive operation indicator

The destructive operation indicator (`_DestructiveIndicator`) is a small inline widget displayed within the Operation group on the Operations page (§10.3). It provides immediate visual feedback about whether the currently configured operation will modify the filesystem.

Visual states:

State	Dot color	Label	Condition
Non-destructive	Green (`#228822` / `#44cc44`)	"Non-Destructive"	Index or Meta Merge without rename, or any operation with rename + dry run enabled.
Destructive	Red (`#cc3333` / `#ff4444`)	"Destructive"	Meta Merge Delete, or any operation with rename enabled and dry run disabled.

The indicator is a `CTkFrame` containing a colored dot (Unicode ●, `CTkLabel`) and a colored text label. It updates immediately when the operation type selector changes or when the dry-run checkbox is toggled. The color values use a tuple for light/dark theme variants (CustomTkinter convention).

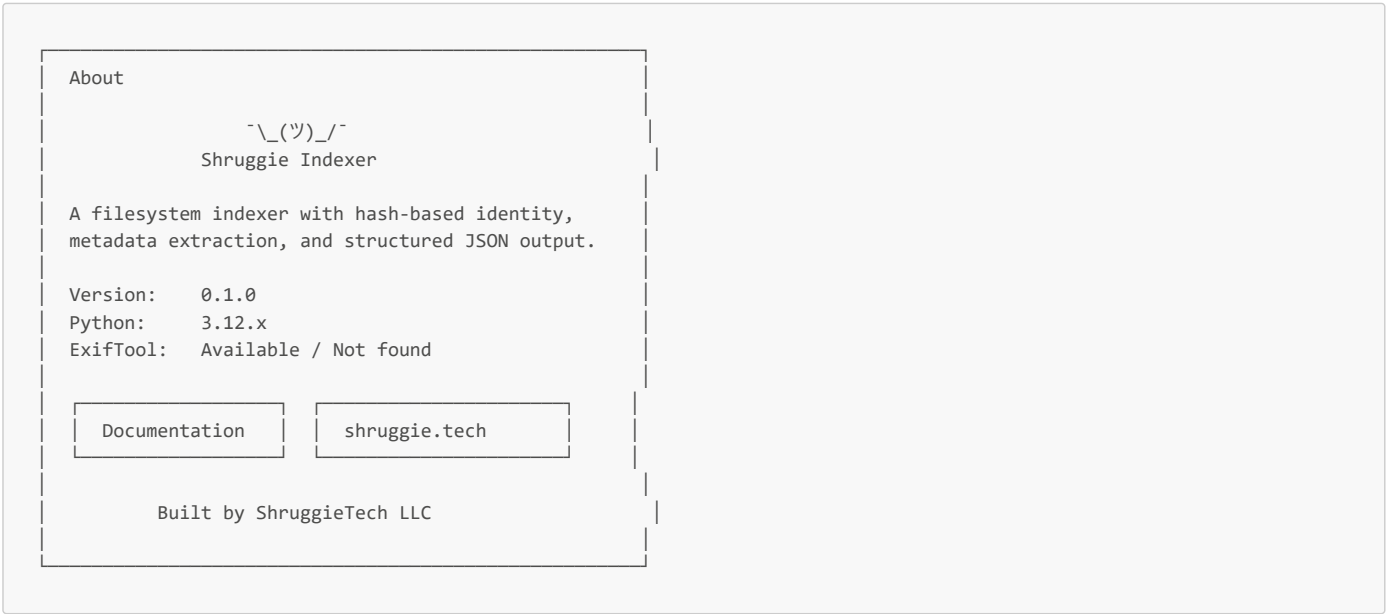
Update triggers:

- Changing the operation type selector updates the indicator via `_update_indicator()`.
- Toggling the "Rename files" checkbox updates the indicator.
- Toggling the dry-run checkbox (when rename is active) updates the indicator.
- No other controls affect the indicator state.

10.8.2. About tab

The About tab (`AboutTab`) is accessed via the third sidebar button and displays static project information, environment details, and navigation links. It has no interactive controls that affect indexing behavior.

Layout:



Information rows:

Field	Source	Notes
Version	`shruggie_indexer.__version__`	Read at import time.
Python	`sys.version_info`	Formatted as `major.minor.micro`.
ExifTool	`shutil.which("exiftool")`	Displays "Available" if found on PATH, "Not found" otherwise. Checked once at tab construction time.

Link buttons:

Button	URL	Action
Documentation	`https://shruggietech.github.io/shruggie-indexer/`	Opens in default browser via `webbrowser.open()`.
shruggie.tech	`https://shruggie.tech`	Opens in default browser via `webbrowser.open()`.

Attribution line: "Built by ShruggieTech LLC" is displayed at the bottom of the tab in a muted font.

The About tab has no state to persist — `get_state()` returns an empty dict and `restore_state()` is a no-op.

10.8.3. Tooltips

Hover tooltips (`_Tooltip`) provide contextual explanations for controls throughout the application. Every interactive widget that benefits from clarification has an associated tooltip string.

Tooltip behavior:

Property	Value
Delay	600 ms (`_TOOLTIP_DELAY_MS`) before appearing.
Dismissal	Hides immediately on mouse leave or click.
Positioning	Below the widget, offset 20px right and 4px below the widget's bottom edge.
Appearance	Dark background (`#333333`), white text, solid border, 6px horizontal / 4px vertical padding.
Font	Segoe UI, 9pt.
Implementation	Uses a `tk.Toplevel` with `wm_overrideredirect(True)` for a borderless popup.

Global enable/disable: Tooltips can be toggled globally via the "Show tooltips on hover" checkbox in Settings (§10.4). The `_Tooltip.set_enabled()` class method controls visibility for all tooltip instances. When disabled, the hover handlers remain bound but suppress the tooltip popup. The tooltip state is persisted in the session file.

10.8.4. Labeled group frames

The `_LabeledGroup` widget is a reusable container used throughout the Operations page (§10.3) to visually organize controls into named sections. Each group has a rounded border, a bold header label, an optional description in a muted font, and a content area.

Property	Value
Corner radius	8 px
Border width	1 px
Border color	`("gray75", "gray30")` (light/dark)
Header font	13pt bold
Description font	11pt, muted color (`("gray40", "gray60")`)
Content padding	12px horizontal, 4–10px vertical

The `content` attribute is a transparent `CTkFrame` inside the group where child widgets are packed.

10.8.5. Debug logging

The GUI installs a queue-based logging handler (`_LogQueueHandler`) on the root `shruggie_indexer` logger during operation execution. This handler captures log records from the core engine and routes them to the output panel's log buffer.

Architecture:

1. A `queue.Queue[str]` is created at application startup.
2. `_LogQueueHandler` extends `logging.Handler` and pushes formatted log messages to the queue via `put_nowait()`.
3. Before each operation, the handler is attached to the `shruggie_indexer` logger with a level determined by the Settings verbosity selection (Normal → WARNING, Verbose → INFO, Debug → DEBUG).

4. A 50ms polling timer (`_POLL_INTERVAL_MS`) on the main thread drains the queue and appends messages to `OutputPanel.append_log()`.
5. After the operation completes, the handler is removed from the logger.

This architecture ensures thread safety — the background indexing thread writes to the queue, and only the main thread reads from it and updates widgets.

10.9. GUI Design Standards

> **Added 2026-02-23:** This subsection codifies GUI design governance for `shruggie-indexer`. All GUI implementation work must comply with the standards defined in this subsection. AI coding agents should review §10.9 before beginning any GUI-related task.

Repeated GUI implementation cycles have demonstrated that UI coherence degrades when changes are made without a codified set of design principles. This subsection establishes two layers of governance:

1. **General usability heuristics** adopted by reference from an industry-standard source.
2. **Project-specific UI standards** tailored to the CustomTkinter desktop application.

10.9.1. Adopted usability heuristics

The project adopts **Jakob Nielsen's 10 Usability Heuristics for User Interface Design** (Nielsen Norman Group, 1994; revised 2020) as the baseline design evaluation framework.

> Nielsen, J. (1994). *10 Usability Heuristics for User Interface Design.* Nielsen Norman Group. URL: https://www.nngroup.com/articles/ten-usability-heuristics/

The following table summarizes each heuristic and its application to `shruggie-indexer`:

#	Heuristic	Application to shruggie-indexer
1	**Visibility of system status**	Progress bars, elapsed timers, log streams, and status labels MUST keep the user informed at all times during indexing operations (§10.5).
2	**Match between system and real world**	Use terminology familiar to the target audience (file indexing, metadata, sidecars). Avoid exposing internal implementation details (queue polling, dataclass names).
3	**User control and freedom**	Provide cancel/stop for long operations (§10.5). Allow undo where feasible (dry-run mode). Never trap the user in an irreversible flow without confirmation.
4	**Consistency and standards**	Widget behavior, label terminology, color semantics, and layout patterns MUST be uniform across all pages (Operations, Settings, About).
5	**Error prevention**	Validate inputs before execution. Disable destructive controls when preconditions are not met (§10.8.1). Use confirmation dialogs for destructive operations.
6	**Recognition rather than recall**	Tooltips (§10.8.3), inline descriptions, and labeled groups (§10.8.4) MUST eliminate the need to consult external documentation for basic operation. Persist session state so users do not need to re-enter settings.
7	**Flexibility and efficiency of use**	Keyboard shortcuts for power users (§10.7). Sensible defaults that minimize configuration for common workflows.
8	**Aesthetic and minimalist design**	Show only controls relevant to the current operation type. Hide or disable irrelevant options (see §10.9.3). Do not clutter the interface with rarely used settings.
9	**Help users recognize, diagnose, and recover from errors**	Error messages MUST be human-readable, identify the problem, and suggest corrective action. No raw tracebacks in the GUI.
10	**Help and documentation**	In-app help via tooltips (§10.8.3) and a link to the documentation site's GUI guide (§10.8.2).

10.9.2. Layout stability

Standard: The interface MUST NOT visually shift, resize, or "glitch" when transitioning between states (idle, running, complete, error). All layout regions MUST be pre-allocated at their maximum required size during initial window construction. Transitions between states (e.g., showing a progress bar where a START button was) MUST occur within pre-allocated regions.

Enforcement: Use fixed-height frames or `grid` geometry with explicit row/column weights. Never use `pack()` with dynamically created widgets that alter the geometry of sibling widgets. If a widget is hidden during certain states, reserve its space with a placeholder or use `grid_remove()` / `grid()` (which preserves geometry) rather than `pack_forget()` / `pack()` (which does not).

See also §10.9.7 for the specific application of this standard to the progress/feedback region.

10.9.3. State-driven control visibility

Standard: Controls that are irrelevant to the currently selected operation type or target type MUST be either visually disabled (grayed out with a brief tooltip explaining why) or hidden. The decision between disabling and hiding follows this rule:

Situation	Behavior
Control exists for this operation type but a precondition is unmet	**Disable** with explanatory tooltip
Control does not exist for this operation type at all	**Hide** (remove from layout, but reserve space if hiding causes layout shift per §10.9.2)

Rationale: Disabled controls communicate the existence of a feature and guide the user toward satisfying its preconditions. Hidden controls reduce clutter when a feature is categorically inapplicable.

10.9.4. Control interdependency transparency

Standard: When one control's state depends on another control's value, the dependency MUST be immediately visible and understandable. Specifically:

1. Changing a parent control MUST instantly update all dependent child controls (enable, disable, show, hide, change default value).
2. The UI MUST NOT enter a state where an enabled control has no effect due to an invisible dependency.
3. If a control is auto-set by a parent (e.g., "Recursive" toggling ON when "Type" changes to "Directory"), the user MUST be able to see the change happen and MUST be able to override it unless the override is logically invalid.

Enforcement: Implement a centralized state reconciliation function (e.g., `_reconcile_controls()`) that is called whenever any control value changes. This function evaluates the full state of all inputs and sets the enabled/disabled/visible state of all dependent controls in one pass. Do not scatter dependency logic across individual widget callbacks.

10.9.5. Output handling clarity

Standard: The user MUST always understand where output will go before pressing START. Output destination MUST be communicated through explicit labeling, not implied by control combinations. The following rules apply:

1. **Default output behavior** MUST be displayed in the UI as descriptive text (e.g., "Output: `<filename>_meta2.json` alongside input file").
2. **Optional output override** (the Output path selector) MUST be clearly labeled as optional, with placeholder text explaining the default.
3. **No-output mode** (view in output panel only, decide later) MUST be an explicit, selectable option.

See §10.6 for output panel behavior and §10.3 for target/output path conventions.

10.9.6. Destructive operation safeguards

Standard: Any operation that modifies or deletes files on disk MUST:

1. Display the persistent destructive/non-destructive indicator (§10.8.1).
2. List the specific destructive actions that will occur (e.g., "Will delete 3 sidecar files after merging").
3. Require a confirmation dialog before execution unless dry-run is active.

10.9.7. Progress and feedback area allocation

Standard: The area occupied by the START button, progress bar, STOP button, and associated status information MUST be a single, fixed-height region in the layout. The region's height MUST accommodate the tallest state (progress bar + status text + current-file label) at all times, even when displaying only the START button in idle state. This prevents layout shifts during state transitions.

This is a specific application of the layout stability standard (§10.9.2) to the progress/feedback region defined in §10.5.

10.9.8. Log and output panel behavior

Standard: The log/output panel is the user's primary feedback channel (§10.6). It MUST:

1. Auto-scroll to the bottom when new content arrives, unless the user has manually scrolled up.
2. Resume auto-scrolling when the user scrolls back to the bottom.
3. Display timestamped entries in the format `HH:MM:SS LEVEL message`.
4. Accept content from both the logging system (§10.8.5) and the progress event system (status messages about currently processing files, §10.5).
5. Have Save and Copy buttons enabled at all times when the panel contains content.

11. Logging and Diagnostics

This section defines the logging and diagnostics system for `shruggie-indexer`. It specifies the logger naming hierarchy, log level mapping to CLI flags, session identifiers, output destinations, log message formatting, and the progress reporting system that feeds the CLI and GUI progress displays.

The logging system is the implementation of two concerns that earlier sections defined in contract form: §4.5 (Error Handling Strategy) established the four error severity tiers (fatal, item-level, field-level, diagnostic) and their behavioral consequences. §8.7 defined the CLI flags (`-v`, `--quiet`) that control logging verbosity. This section provides the concrete `logging`-framework wiring that connects those contracts to runtime behavior.

Module location: There is no dedicated logging module in the source package layout (§3.2). Logging configuration is performed in the CLI entry point (`cli/main.py`), the GUI entry point (`gui/app.py`), and optionally by API consumers. Individual modules obtain their loggers at import time via

`logging.getLogger(__name__)` — no centralized logging module is needed beyond Python's built-in `logging` package. The one logging-specific artifact is the `SessionFilter` class (§11.4), which is defined in the CLI module where it is used and does not require its own file.

> **Historical note (DEV-08):** The original `Vbs` function is a 130-line PowerShell function with six internal sub-functions (`VbsFunctionStackTotalDepth`, `VbsLogPath`, `VbsLogRealityCheck`, `VbsLogWrite`, `VbsUpdateFunctionStack`, `VbsUpdateFunctionStackExtractNumber`) that manually constructs log entries, manages log file creation, compresses call stacks, and formats colorized console output. It is called explicitly by every function in the pslib library, requiring each caller to pass a `Caller` string, a `Status` shorthand, and a `Verbosity` flag through the call chain. The implementation replaces all of this with Python's standard `logging` framework — which provides named loggers, hierarchical level filtering, pluggable formatters and handlers, and automatic caller identification — eliminating 100% of the `Vbs` implementation and 100% of the manual call-stack bookkeeping that pervaded every function in the original.

11.1. Logging Architecture

Design principles

The logging system follows four principles that govern all implementation decisions:

Principle 1 — Standard library only. The core logging system uses Python's `logging` module exclusively. No third-party logging libraries (e.g., `structlog`, `loguru`) are required. Optional enhancements like `rich.logging.RichHandler` for colorized console output are welcome as extras but MUST NOT be assumed. A bare `pip install shruggie-indexer` provides full logging functionality.

Principle 2 — stderr for logs, stdout for data. All log output is directed to `sys.stderr`. The `sys.stdout` stream is reserved exclusively for JSON output (§6.9, §8.3). This separation is critical: it allows `shruggie-indexer /path | jq .` to work correctly even at maximum verbosity — the JSON stream on stdout is never contaminated by log messages.

> **Historical note:** The original's `Vbs` function conflated log output and console output through `Write-Host`, which writes to the PowerShell host stream rather than stdout or stderr. The explicit stderr routing follows Unix convention and avoids this conflation.

Principle 3 — No mandatory file logging. The tool does not write log files by default. Console (stderr) output is the sole default destination. Optional persistent log file output is available through three enablement mechanisms:

1. **CLI flag:** `--log-file` (no argument) writes to the platform-specific default directory; `--log-file <path>` writes to the specified path.
2. **TOML configuration:** `[logging] file_enabled = true` enables file logging to the default directory; `file_path` overrides the location.
3. **GUI toggle:** A "Write log files" checkbox on the Settings page enables per-operation log file output.

When enabled, log files are written to the platform-appropriate application data directory:

Platform	Directory
Windows	`%LOCALAPPDATA%\ShruggieTech\shruggie-indexer\logs\`
macOS	`~/Library/Application Support/ShruggieTech/shruggie-indexer/logs/`
Linux	`~/.local/share/shruggie-indexer/logs/`

Log files are named by date and session: `YYYY-MM-DD_HHMMSS.log`. The log file uses the CLI's full format including session ID (§11.5). The directory is created on first use via `platformdirs`. The tool does not manage log rotation — each invocation creates a new file. This is a deliberate design choice: a CLI tool that silently writes to the filesystem on every invocation is surprising behavior, so file logging is always opt-in.

> **Historical note:** The original wrote to monthly log files (`YYYY_MM.log`) in a hardcoded directory (`C:\bin\pslib\logs`) on every invocation, regardless of verbosity settings. That path was a Windows-specific constant that would not translate to cross-platform use.

Principle 4 — Logging and progress are separate systems. The `logging` framework handles diagnostic messages (warnings, errors, debug traces), while the `ProgressEvent` callback system (§9.4, §11.6) handles user-facing progress reporting. They share an output destination in the CLI (both appear on stderr), but they are architecturally independent — the progress system can drive a GUI progress bar, a `tqdm` progress bar, or be disabled entirely, without affecting diagnostic logging.

> **Historical note:** The original's `Vbs` function handled both diagnostic logging and user-facing progress reporting (e.g., `"[42/100 (42%)] Processing photo.jpg"`), interleaving them into the same output channel. The separation here eliminates that coupling.

How logging is configured

Logging configuration happens exactly once per invocation, at the entry point layer — before the core indexing engine is called. Each entry point is responsible for configuring the logging system appropriate to its context:

CLI (`cli/main.py`). The `main()` function calls `configure_logging()` (a private function within the CLI module) after parsing arguments but before constructing `IndexerConfig` or calling `index_path()`. The function creates a `StreamHandler` on `sys.stderr`, attaches a `Formatter` and the `SessionFilter` (§11.4), and sets the root logger level based on the `-v`/`-q` flags.

GUI (`gui/app.py`). The application's `__init__` method configures a `logging.Handler` subclass that enqueues log records into the same `queue.Queue` used by the progress display (§10.5). The handler formats records identically to the CLI formatter but routes them to the GUI's log stream textbox rather than stderr. The log level is controlled by the Settings panel's Verbosity combobox (§10.4).

API consumers. Library consumers who `import shruggie_indexer` are responsible for configuring their own logging. The library's `core/` modules emit log records through their per-module loggers (§11.2) but do not install any handlers. If the consumer does not configure logging, Python's `logging.lastResort` handler (a `StreamHandler` on stderr with WARNING level) provides minimal output. This is the correct default behavior for a library — the library should never configure the root logger or install handlers, because doing so would interfere with the consuming application's logging setup.

```
# Illustrative — not the exact implementation.
# cli/main.py

def configure_logging(verbose_count: int, quiet: bool, session_id: str) -> None:
    """Configure the logging system for CLI invocation."""
    if quiet:
        level = logging.CRITICAL
    else:
        level = {0: logging.WARNING, 1: logging.INFO}.get(verbose_count, logging.DEBUG)

    handler = logging.StreamHandler(sys.stderr)
    handler.setFormatter(logging.Formatter(
        fmt="%(asctime)s  %(session_id)s  %(levelname)-8s  %(name)s  %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    ))
    handler.addFilter(SessionFilter(session_id))

    # Configure the package root logger — not the global root logger.
    package_logger = logging.getLogger("shruggie_indexer")
    package_logger.setLevel(level)
    package_logger.addHandler(handler)

    # Prevent propagation to the root logger, which may have
    # handlers installed by the consuming environment.
    package_logger.propagate = False
```

> **Critical implementation note:** The CLI configures the `shruggie_indexer` package logger, not the root logger (`logging.getLogger()`). Configuring the root logger would affect all loggers in the process, including those from third-party libraries (`click`, `urllib3`, etc.), which is both intrusive and noisy. By scoping the handler to the `shruggie_indexer` namespace, only log records from the indexer's own modules are captured. Third-party library logging is left to the consumer's own configuration. The `propagate = False` setting prevents double-logging if a consumer has also configured a root handler.

11.2. Logger Naming Hierarchy

Every module in the `shruggie_indexer` package obtains its logger via `logging.getLogger(__name__)` as the first executable statement after imports. This produces a dotted-name logger hierarchy that mirrors the package structure:

```
shruggie_indexer                  # package root (configured by CLI/GUI)
shruggie_indexer.core.traversal   # filesystem traversal
shruggie_indexer.core.paths       # path resolution
shruggie_indexer.core.hashing     # hash computation
shruggie_indexer.core.timestamps  # timestamp extraction
shruggie_indexer.core.exif        # exiftool invocation
shruggie_indexer.core.sidecar     # sidecar discovery/parsing
shruggie_indexer.core.entry       # entry construction orchestrator
shruggie_indexer.core.serializer  # JSON output
shruggie_indexer.core.rename      # file rename operations
shruggie_indexer.config.loader    # configuration loading
shruggie_indexer.cli.main         # CLI entry point
shruggie_indexer.gui.app          # GUI entry point
```

Logger naming via `__name__`

Every module obtains its logger via `logging.getLogger(__name__)`. This produces a dotted-name logger hierarchy that mirrors the package structure. Logger names are derived automatically from the module's fully-qualified import path, requiring no parameter passing. The `%(name)s` format token in the handler's formatter produces the logger name in every log record, providing module-level traceability without manual bookkeeping.

> **Historical note:** The original's `Vbs` function required every caller to pass a `Caller` string containing a manually-maintained colon-delimited call stack (e.g., `"MakeIndex:MakeObject:GetFileExif"`). This approach was error-prone (callers could pass incorrect or stale stack strings), coupled every function to the logging interface, and required the `VbsUpdateFunctionStack` compression function to keep log lines manageable during deep recursion. Python's `logging.getLogger(__name__)` solves all three problems by construction.

Per-module logger pattern: Every `core/` module follows this pattern:

```
# core/hashing.py
import logging

logger = logging.getLogger(__name__)

def hash_file(path: Path, compute_sha512: bool = False) -> HashSet:
    logger.debug("Hashing file: %s", path)
    # ...
    logger.debug("Hash complete for %s: md5=%s", path, result.md5)
    return result
```

The `logger` variable is module-level, created once at import time. It is not a global mutable state concern (§4.4) because `logging.getLogger()` returns the same logger instance for the same name on every call — it is effectively a singleton lookup, and the logger's configuration (level, handlers) is controlled by the entry point, not by the module.

Logger hierarchy and level inheritance

Python's `logging` framework uses dot-separated names to form a hierarchy. Setting the level on `shruggie_indexer` (the package root logger) propagates to all child loggers — `shruggie_indexer.core.hashing`, `shruggie_indexer.core.exif`, etc. — unless a child logger has an explicitly overridden level. This hierarchy enables the `-vv` vs. `-vvv` distinction defined in §8.7:

CLI flag	Package root level	Effect
(none)	`WARNING`	Only warnings and errors from any module.
`-v`	`INFO`	Progress-level messages: items processed, output destinations, implication chain activations.
`-vv`	`DEBUG`	Detailed internal state from all modules: hash values, exiftool commands, sidecar regex matches, config resolution steps.
`-vvv`	`DEBUG` + specific loggers re-enabled	Maximum verbosity. Same as `-vv` but also enables loggers that are silenced at `-vv` for noise reduction (see below).

The `-vv` vs. `-vvv` distinction

At `-vv` (DEBUG), all modules emit their DEBUG messages. This is already verbose — large directory trees can produce thousands of per-file hash and timestamp log lines. The `-vvv` level provides a further increase by re-enabling two categories of messages that are suppressed at `-vv` for readability:

1. **Per-item timing data.** `core/entry.py` can emit the elapsed wall-clock time for each item's entry construction. At `-vv`, these are suppressed (the per-file overhead of `time.perf_counter()` calls and log formatting is undesirable for most debugging scenarios). At `-vvv`, they are enabled.
2. **Exiftool raw output.** `core/exif.py` can emit the complete JSON string returned by `exiftool` for each file. At `-vv`, only the filtered/processed result is logged. At `-vvv`, the raw subprocess output is also logged.

This distinction is implemented not through custom log levels (which would violate the standard `logging` API) but through a naming convention: the noisy loggers use a `.trace` suffix on their name — e.g., `shruggie_indexer.core.entry.trace`, `shruggie_indexer.core.exif.trace`. At `-vv`, these trace loggers are explicitly set to `WARNING` (silenced). At `-vvv`, they inherit the package root's `DEBUG` level (active).

```
# Illustrative — inside configure_logging()
if verbose_count == 2:
    # -vv: silence trace loggers specifically
    logging.getLogger("shruggie_indexer.core.entry.trace").setLevel(logging.WARNING)
    logging.getLogger("shruggie_indexer.core.exif.trace").setLevel(logging.WARNING)
# At verbose_count >= 3 (-vvv), no per-logger overrides — everything is DEBUG.
```

The `.trace` logger pattern is purely a convention — no separate `trace` modules exist. The trace loggers are obtained via `logging.getLogger(__name__ + ".trace")` within the module that uses them:

```
# core/entry.py
import logging

logger = logging.getLogger(__name__)
trace_logger = logging.getLogger(__name__ + ".trace")

def build_file_entry(path: Path, config: IndexerConfig) -> IndexEntry:
    t0 = time.perf_counter()
    # ...
```

```
    elapsed = time.perf_counter() - t0
    trace_logger.debug("Entry construction for %s took %.3fs", path.name, elapsed)
```

> **Historical note:** The graduated three-level model (`WARNING` → `INFO` → `DEBUG` → `DEBUG+trace`) provides substantially more diagnostic control than the original's binary `$Verbosity` boolean, which offered only two states: all output or file-only output.

11.3. Log Levels and CLI Flag Mapping

Standard level usage

The implementation uses Python's five standard log levels with consistent semantic meanings across all modules. No custom log levels are defined.

Level	Numeric	Usage	Example messages
`CRITICAL`	50	Fatal conditions that prevent the tool from operating at all. Only used for startup failures. The CLI exits immediately after a CRITICAL log.	`"Configuration file is malformed TOML: {path}"`, `"Target path does not exist: {path}"`
`ERROR`	40	Item-level failures that cause an item to be skipped or populated with degraded fields. The tool continues processing remaining items.	`"Permission denied reading file: {path}"`, `"exiftool returned non-zero for {path}: {stderr}"`, `"Failed to parse sidecar JSON: {path}"`
`WARNING`	30	Conditions that do not prevent processing but indicate unexpected or suboptimal behavior. The user should be aware but no action is required.	`"exiftool not found on PATH; EXIF extraction disabled for this invocation"`, `"--rename implies --inplace; enabling in-place output"`, `"Skipping dangling symlink: {path}"`, `"3 items failed during indexing"`
`INFO`	20	Progress milestones, operational decisions, and summary information. Useful for understanding what the tool did without drowning in per-item detail.	`"Indexing directory: {path} (recursive)"`, `"Discovered 1,247 items"`, `"Output written to: {outfile}"`, `"Elapsed time: 12.4s"`
`DEBUG`	10	Per-item internal state, intermediate values, and decision traces. Useful for diagnosing why a specific file produced unexpected output.	`"Hashing file: {path}"`, `"md5={hash}, sha256={hash}"`, `"Sidecar match: {pattern} → {type}"`, `"Extension '{ext}' failed validation; using empty string"`

Mapping to error severity tiers

The error severity tiers defined in §4.5 map to log levels as follows:

Severity tier (§4.5)	Log level	Behavioral consequence
Fatal	`CRITICAL`	Abort invocation. Exit with non-zero code.
Item-level	`ERROR`	Skip item or populate with degraded fields. Continue processing.
Field-level	`WARNING` or `ERROR`	Populate affected field with `null`. Continue processing current item. `WARNING` when the condition is expected (e.g., exiftool unavailable); `ERROR` when unexpected (e.g., exiftool crash on a specific file).
Diagnostic	`DEBUG`	No effect on output. Trace information for debugging.

CLI flag mapping (summary)

This table consolidates the CLI flag → log level mapping defined in §8.7 with the detailed level semantics above:

CLI flags	Effective level	What the user sees on stderr
(default)	`WARNING`	Only warnings and errors. Silent for normal operation.
`-v`	`INFO`	Progress milestones, summary stats, implication chain messages.
`-vv`	`DEBUG` (trace loggers silenced)	Per-item detail: hashes, timestamps, sidecar matches, exiftool invocations.
`-vvv`	`DEBUG` (all)	Maximum detail: per-item timing, raw exiftool JSON, config resolution trace.
`-q`	`CRITICAL`	Fatal errors only. Overrides `-v` if both are specified.

11.4. Session Identifiers

Each invocation of `shruggie-indexer` — whether from the CLI, GUI, or API — generates a unique session identifier. The session ID is a 32-character lowercase hexadecimal string derived from a UUID4:

```
import uuid

session_id = uuid.uuid4().hex  # e.g., "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6"
```

The session ID is generated once, at the start of the invocation, before any processing begins. It is immutable for the duration of the invocation.

Purpose

The session ID serves two functions:

1. **Log correlation.** When multiple invocations run concurrently (e.g., parallel indexing jobs in a CI pipeline), or when log output from multiple invocations is aggregated into a shared sink, the session ID uniquely identifies which log lines belong to which invocation. This is the same role served by the original's `$LibSessionID` (a GUID generated once per pslib session).
2. **Output provenance.** The session ID MAY be included in the JSON output metadata in future schema versions (post-MVP) to link an index entry back to the invocation that produced it. For the MVP, the session ID is logging-only.

Injection mechanism

The session ID is injected into log records via a `logging.Filter`, not a `logging.LoggerAdapter`. The Filter approach is preferred because it works transparently with all loggers in the hierarchy without requiring every module to use a special adapter class:

```
class SessionFilter(logging.Filter):
    """Inject the session ID into every log record."""

    def __init__(self, session_id: str) -> None:
        super().__init__()
        self.session_id = session_id

    def filter(self, record: logging.LogRecord) -> bool:
        record.session_id = self.session_id  # type: ignore[attr-defined]
        return True
```

The filter is attached to the handler (not to individual loggers), so it applies to every log record that reaches the handler regardless of which module emitted it. The `session_id` attribute is then available in the formatter via `%(session_id)s`.

> **Historical note:** The original's `$LibSessionID` was a global variable passed through every function's parameter chain to `Vbs`. The Python implementation eliminates this parameter entirely — no `core/` module function accepts or passes a session ID. The `SessionFilter` injects it transparently at the handler level, removing one parameter from every function signature in the call chain.

Lifecycle

Entry point	Where generated	How injected
CLI	In `main()`, before `configure_logging()` is called.	Passed to `SessionFilter` constructor, which is attached to the stderr handler.
GUI	In `ShruggiIndexerApp.__init__()`, once at application startup. A new session ID is generated for each indexing operation (not per-application-launch), matching the semantics of "one session = one invocation."	Attached to the GUI's queue-based log handler via the same `SessionFilter`. The session ID is updated when a new indexing job starts.
API	Not generated by the library. API consumers are responsible for their own logging configuration. If a consumer wants session IDs, they configure their own `SessionFilter`. The library's `core/` modules never read or depend on the session ID — it is purely a logging-layer concern.	

11.5. Log Output Destinations

CLI destinations

The CLI has a single log output destination: a `logging.StreamHandler` attached to `sys.stderr`. There is no file handler, no network handler, and no syslog handler configured by default.

Log message format (CLI):

```
2026-02-15 14:30:02  a1b2c3d4  WARNING   shruggie_indexer.core.exif  exiftool not found on PATH; EXIF extraction
disabled
2026-02-15 14:30:02  a1b2c3d4  INFO      shruggie_indexer.core.traversal  Discovered 1,247 items in /path/to/target
2026-02-15 14:30:03  a1b2c3d4  DEBUG     shruggie_indexer.core.hashing  Hashing file: photo.jpg (md5=A8A8...)
```

Format string:

```
fmt = "%(asctime)s  %(session_id)s  %(levelname)-8s  %(name)s  %(message)s"
datefmt = "%Y-%m-%d %H:%M:%S"
```

Format field descriptions:

Field	Token	Description
Timestamp	`%(asctime)s`	Local wall-clock time in `YYYY-MM-DD HH:MM:SS` format. Seconds-level precision is sufficient for CLI diagnostics.
Session ID	`%(session_id)s`	First 8 characters of the 32-character session ID. The abbreviated form balances identifiability with line length — 8 hex characters provide ~4 billion unique values, which is sufficient to distinguish concurrent invocations. The full 32-character ID is available in the `LogRecord` attribute for any consumer that configures a custom formatter.
Level	`%(levelname)-8s`	Left-aligned, padded to 8 characters for visual column alignment.
Logger name	`%(name)s`	The fully-qualified dotted logger name. Provides equivalent traceability to the original's `Caller` field without manual call-stack management.
Message	`%(message)s`	The log message body, formatted via `%`-style string interpolation (Python's `logging` default).

Deliberate omission: detailed call-stack prefix. The log format uses the logger name (`%(name)s`) to identify the source module. The function name is available via `%(funcName)s` if needed, but it is excluded from the default format because it adds visual noise without improving most debugging scenarios — the module name is almost always sufficient to locate the relevant code. Consumers who need function-level detail can reconfigure the formatter.

> **Historical note:** The original wrapped every log message in a prefix like `pslib(MakeIndex:MakeObject(3)):` that identified the library name and the compressed call stack. The Python logger name provides equivalent module-path traceability without manual stack management.

GUI destinations

The GUI uses a custom `logging.Handler` subclass that serializes log records into the `queue.Queue` shared with the progress display system (§10.5). The main thread's 50ms polling timer dequeues records and appends them to the log stream textbox.

Log message format (GUI):

```
14:30:02  WARNING   exiftool not found on PATH; EXIF extraction disabled
14:30:02  INFO      Discovered 1,247 items in /path/to/target
14:30:03  DEBUG     Hashing file: photo.jpg
```

Format string:

```
fmt = "%(asctime)s  %(levelname)-8s  %(message)s"
datefmt = "%H:%M:%S"
```

The GUI format is more compact than the CLI format: the session ID is omitted (the GUI runs one invocation at a time, so correlation is unnecessary), the logger name is omitted (the log stream textbox is already contextualized by the running operation), and the timestamp uses time-only format (the date is visible in the system clock). The GUI SHOULD apply level-based color coding to log messages in the textbox: `ERROR` and `CRITICAL` in red, `WARNING` in yellow/amber, `INFO` in the default text color, and `DEBUG` in a muted gray. This replaces the original's `Write-Host -ForegroundColor` colorization with the GUI's own text styling.

Optional: colorized CLI output

If `rich` is installed (it is listed as a recommended third-party package in §12.3), the CLI MAY use `rich.logging.RichHandler` instead of the plain `StreamHandler` to produce colorized, column-aligned log output on terminals that support ANSI escape codes. The detection and fallback logic:

```
# Illustrative — inside configure_logging()
try:
```

```
    from rich.logging import RichHandler
    handler = RichHandler(
        console=rich.console.Console(stderr=True),
        show_time=True,
        show_path=False,
        rich_tracebacks=True,
    )
except ImportError:
    handler = logging.StreamHandler(sys.stderr)
    handler.setFormatter(logging.Formatter(fmt=..., datefmt=...))
```

This is strictly optional. The core logging system works identically with or without `rich`. The `SessionFilter` is attached to whichever handler is constructed.

> **Historical note:** The original's `Vbs` function hardcoded console colors per severity level (`Gray`, `DarkRed`, `Magenta`, `DarkYellow`, `DarkCyan`, `DarkGreen`, `DarkGray`) via `Write-Host -ForegroundColor`. The `rich`-based approach adapts to the terminal's color capabilities (256-color, truecolor, or no-color fallback). The original's `SUCCESS` and `UNKNOWN` status levels have no direct Python `logging` equivalents — success messages are logged at `INFO` level, and the "unknown" status (a fallback for unrecognized status strings) is unnecessary since the Python framework does not accept freeform status strings.

File destinations

> **Added 2026-02-23:** Persistent log file support was introduced as an optional feature. See Principle 3 (§11.1) for enablement mechanisms.

When persistent log file output is enabled, a `logging.FileHandler` is attached to the `shruggie_indexer` package logger alongside the existing CLI or GUI handler. The file handler uses the CLI's full log format including session ID:

```
2026-02-23 14:30:02  a1b2c3d4  INFO      shruggie_indexer.core.hashing  Hashing file: photo.jpg
```

Format string:

```
fmt = "%(asctime)s  %(session_id)s  %(levelname)-8s  %(name)s  %(message)s"
datefmt = "%Y-%m-%d %H:%M:%S"
```

The log level written to the file matches the currently configured verbosity. The file handler is added in the CLI's `configure_logging()` function (when `--log-file` is specified) or in the GUI application initialization (when the "Write log files" toggle is enabled in Settings). In the CLI, the handler persists for the process lifetime. In the GUI, a new log file is created per indexing operation, and the handler is removed after each job finishes.

The `[logging]` TOML section is recognized by the configuration loader (§7.1) but is handled by the CLI/GUI entry points rather than `IndexerConfig`, because logging configuration is an entry-point concern that varies by invocation context.

11.6. Progress Reporting

Progress reporting is the user-facing system that communicates "what is the tool doing right now and how far along is it?" to the CLI and GUI. It is architecturally separate from diagnostic logging (Principle 4, §11.1) — they are produced by different systems, serve different audiences, and can be independently disabled.

Architecture

Progress reporting flows through the `ProgressEvent` callback system defined in §9.4. The core engine (`build_directory_entry()`) invokes the `progress_callback` at defined intervals during directory traversal. The callback receives a `ProgressEvent` dataclass instance containing the current phase, item counts, current path, and an optional human-readable message. The caller — CLI, GUI, or API consumer — decides how to present this information.

```
Core engine                  Callback            Consumer
———————————                  ————————            ————————
build_directory_entry()  ——→  progress_callback()  ——→  CLI: tqdm bar / log message
                                                    ——→  GUI: progress bar + log textbox
                                                    ——→  API: custom handler
```

This architecture decouples the engine from all presentation concerns. The engine does not know whether progress is displayed as a terminal progress bar, a GUI widget, a log message, or nothing at all. It simply invokes the callback if one is provided.

CLI progress reporting

The CLI's progress presentation depends on the verbosity level and available third-party packages:

Default behavior (no `-v` flag, `tqdm` not installed). No progress output. The tool runs silently until completion (only warnings/errors appear on stderr). This matches the Unix convention of silent-unless-broken.

With `-v` (INFO level). Progress milestones are emitted as log messages to stderr: discovery count, completion percentage at 25%/50%/75%/100% intervals, and the final summary (total items, elapsed time). These are standard `INFO`-level log records from the entry orchestrator — not a separate progress display. They interleave naturally with any warnings or errors.

```
2026-02-15 14:30:02  a1b2c3d4  INFO      shruggie_indexer.core.entry  Indexing: /path/to/target (recursive)
2026-02-15 14:30:02  a1b2c3d4  INFO      shruggie_indexer.core.entry  Discovered 1,247 items
2026-02-15 14:30:05  a1b2c3d4  INFO      shruggie_indexer.core.entry  Progress: 312/1,247 (25%)
2026-02-15 14:30:08  a1b2c3d4  INFO      shruggie_indexer.core.entry  Progress: 624/1,247 (50%)
2026-02-15 14:30:11  a1b2c3d4  INFO      shruggie_indexer.core.entry  Progress: 936/1,247 (75%)
2026-02-15 14:30:14  a1b2c3d4  INFO      shruggie_indexer.core.entry  Progress: 1,247/1,247 (100%)
2026-02-15 14:30:14  a1b2c3d4  INFO      shruggie_indexer.core.entry  Completed in 12.4s (3 warnings)
```

With `-v` active. The CLI displays a `tqdm` progress bar on stderr instead of percentage log lines. `tqdm` is a required dependency (§12.3). The progress callback feeds `tqdm.update()` calls, and `tqdm.write()` is used for log messages to avoid disrupting the progress bar. The log-based progress reporting described above serves as the fallback for non-TTY environments or when `--no-progress` is specified.

With `-q` (quiet mode). No progress output of any kind. The `progress_callback` is still invoked (it has negligible overhead), but the CLI's callback implementation ignores all events.

GUI progress reporting

The GUI's progress display system is defined in detail in §10.5. In summary: the `progress_callback` enqueues `ProgressEvent` objects into a `queue.Queue`, which the main thread drains on a 50ms timer to update the progress bar widget, the status text, and the log stream textbox. The GUI provides richer feedback than the CLI — a visual progress bar, a live item count, and a scrollable log stream — because the GUI context warrants it and the event-loop architecture supports it.

Progress callback implementation pattern

The CLI constructs a progress callback that bridges `ProgressEvent` data to its chosen display mechanism. The following illustrative example shows the log-based implementation:

```python
# Illustrative — inside cli/main.py

def make_progress_callback(
    logger: logging.Logger,
    milestone_pct: tuple[int, ...] = (25, 50, 75, 100),
) -> Callable[[ProgressEvent], None]:
    """Create a CLI progress callback that emits log messages at milestones."""
    last_milestone = 0

    def callback(event: ProgressEvent) -> None:
        nonlocal last_milestone

        if event.phase == "discovery" and event.items_total is not None:
            logger.info("Discovered %s items", f"{event.items_total:,}")
            return

        if event.phase == "processing" and event.items_total:
            pct = int(event.items_completed / event.items_total * 100)
            for m in milestone_pct:
                if pct >= m > last_milestone:
                    logger.info(
                        "Progress: %s/%s (%d%%)",
                        f"{event.items_completed:,}",
                        f"{event.items_total:,}",
                        pct,
                    )
                    last_milestone = m
                    break

        # Forward any embedded log messages from the engine.
        if event.message and event.level:
            level = getattr(logging, event.level.upper(), logging.INFO)
            logger.log(level, "%s", event.message)

    return callback
```

Elapsed time reporting

At the end of every invocation (CLI and GUI), the tool logs the total elapsed wall-clock time. The timer starts immediately before `index_path()` is called and stops immediately after it returns (or raises). The elapsed time is logged at `INFO` level.

```
t0 = time.perf_counter()
try:
    entry = index_path(target, config, progress_callback=callback)
finally:
    elapsed = time.perf_counter() - t0
    logger.info("Elapsed time: %.1fs", elapsed)
```

The original computes elapsed time using `(Get-Date) - $TimeStart` and formats it as `H:M:S.ms`. The implementation uses `time.perf_counter()` for sub-millisecond precision and formats the result as seconds with one decimal place for typical invocations, or `MM:SS` for invocations exceeding 60 seconds. The formatting is a presentation detail — the `perf_counter()` value is the authoritative measurement.

Item failure summary

When one or more items fail during processing (item-level or field-level errors, §4.5), the CLI logs a summary at `WARNING` level after the elapsed time:

```
2026-02-15 14:30:14  a1b2c3d4  WARNING   shruggie_indexer.core.entry  3 items encountered errors during indexing
```

The per-item errors were already logged individually as `ERROR` or `WARNING` messages during processing. The summary provides a quick indication that the output is incomplete, without requiring the user to scroll through the full log to discover whether any errors occurred. The count of failed items also determines the exit code: if `failed_items > 0`, the CLI exits with `PARTIAL_FAILURE` (exit code 1, §8.10).

12. External Dependencies

This section catalogs every dependency — binary, standard library, and third-party — that `shruggie-indexer` consumes at runtime, during testing, or during build/packaging. It defines which dependencies are required, which are optional, how optional dependencies are declared and gated, and which original dependencies have been eliminated. The section serves both as a dependency manifest for implementers and as the normative reference for the `[project.dependencies]` and `[project.optional-dependencies]` tables in `pyproject.toml` (§13.2).

The dependency architecture implements design goal G5 (§2.3): the tool declares four required runtime Python dependencies — `click`, `orjson`, `pyexiftool`, and `tqdm` — plus `exiftool` as an external binary. A bare `pip install shruggie-indexer` installs all four packages. The GUI package (`customtkinter`) is declared as an optional extra. Development and testing tools are declared in the `dev` extra. See §12.3 for the per-package rationale.

Every dependency is explicitly declared: external binaries are probed at runtime (§12.5), standard library modules are imported at the top of each file, and third-party packages are declared in `pyproject.toml` with version constraints. Nothing is silently assumed to exist.

> **Historical note:** The original `MakeIndex` had an implicit dependency set — it called functions, invoked binaries, and read global variables without any declaration mechanism. An implementer had to trace every code path to discover what was required.

12.1. Required External Binaries

`exiftool` is the sole external binary dependency of the entire project. No other binary is required at runtime on any platform.

exiftool

Field	Value
Binary name	`exiftool`
Minimum version	≥ 12.0
Purpose	Extraction of EXIF, XMP, IPTC, and other embedded metadata from media files
Required by	`core/exif.py` (§6.6)
Resolution	Must be present on the system `PATH`. Resolved via `shutil.which("exiftool")` at runtime (§12.5).
Cross-platform availability	Available on Windows, Linux, and macOS. See https://exiftool.org/ for platform-specific installation.
Failure behavior	Graceful degradation — if `exiftool` is absent, the `metadata` array in all `IndexEntry` output omits the `exiftool.json_metadata` entry; all other indexing operations (hashing, timestamps, sidecar metadata, identity generation) continue normally (§4.5).

The minimum version requirement of ≥ 12.0 is driven by the use of the `-api requestall=3` and `-api largefilesupport=1` arguments (§6.6), which were stabilized in exiftool 12.x. Version checking is not enforced at runtime — an older exiftool will likely work for most files but may produce incomplete or incorrect metadata for some edge cases. Version checking is a potential post-MVP enhancement.

> **Historical note (DEV-05, DEV-06):** The original depended on two external binaries: `exiftool` and `jq`. The `jq` dependency is eliminated — JSON parsing uses `json.loads()` and unwanted keys are removed with a dict comprehension. Additionally, `MetaFileRead` in the original depended on `certutil` (a Windows-only system utility) for Base64 encoding of binary sidecar file contents. Python's `base64.b64encode()` replaces this with a cross-platform, subprocess-free standard library call. See §12.4 for the complete elimination catalog.

12.2. Python Standard Library Modules

The core indexing engine (`core/`, `config/`, `models/`) uses only standard library modules. The following table lists every stdlib module imported by the source package, organized by the component that consumes them. This is not a theoretical "might use" list — it is the definitive set of imports an implementer will write.

Core engine modules

Standard library module	Consuming package module(s)	Purpose
`base64`	`core/sidecar.py`	Base64 encoding of binary sidecar file contents (screenshots, thumbnails, torrents). Replaces the original's `certutil` pipeline (DEV-05).
`dataclasses`	`config/types.py`, `models/schema.py`	Frozen dataclass definitions for `IndexerConfig`, `IndexEntry`, and all v2 schema sub-objects. `dataclasses.asdict()` used by the serializer (§6.9).
`datetime`	`core/timestamps.py`	ISO 8601 string generation from `os.stat()` float values via `datetime.fromtimestamp()` and `.isoformat()`.
`hashlib`	`core/hashing.py`	All hash computation — MD5, SHA1, SHA256, SHA512 for both file content and string inputs. Multi-algorithm single-pass hashing via `hashlib.new()` (§6.3).
`json`	`core/serializer.py`, `core/exif.py`, `core/sidecar.py`	JSON parsing of exiftool output (`json.loads()`), JSON parsing of sidecar metadata files, and JSON serialization of `IndexEntry` output (`json.dumps()`).
`logging`	All modules	Per-module loggers via `logging.getLogger(__name__)`. Logging configuration in CLI and GUI entry points (§11).
`os`	`core/traversal.py`, `core/timestamps.py`, `core/hashing.py`	`os.stat()` for timestamp extraction and file size. `os.walk()` as an alternative traversal backend. `os.fsdecode()` for filename handling.
`pathlib`	`core/traversal.py`, `core/paths.py`, `core/entry.py`, `core/rename.py`, `config/loader.py`	Path resolution, component extraction, directory enumeration (`Path.iterdir()`), glob matching, and path arithmetic. Central to the cross-platform path abstraction (§6.2).
`re`	`core/sidecar.py`, `config/defaults.py`, `config/loader.py`	Regex matching for sidecar file type detection, filesystem exclusion filters, and extension validation. Compiles the patterns from the `MetadataFileParser` configuration (§7.3).
`shutil`	`core/exif.py`, `core/rename.py`	`shutil.which()` for exiftool binary probing (§12.5). `shutil.move()` as a cross-filesystem rename fallback (§6.10).
`subprocess`	`core/exif.py`	Fallback exiftool invocation via `subprocess.run()` with argfile when `pyexiftool` batch mode is unavailable (§6.6).
`tempfile`	`core/exif.py`	`tempfile.NamedTemporaryFile` for writing exiftool argfiles in the `subprocess.run()` fallback backend (§6.6). Not used by the primary `pyexiftool` batch backend.
`time`	`cli/main.py`, `core/entry.py`	`time.perf_counter()` for elapsed-time measurement in progress reporting and per-item trace logging (§11.6).
`tomllib`	`config/loader.py`	TOML configuration file parsing (§7.6). Standard library since Python 3.11; the `>=3.12` floor (§2.5) guarantees availability.
`typing`	All modules	Type annotations: `Optional`, `Callable`, `Sequence`, `Literal`, etc. Used for function signatures, dataclass field types, and `TYPE_CHECKING` import guards.
`uuid`	`cli/main.py`, `gui/app.py`	Session ID generation via `uuid.uuid4()` (§11.4).

Note on `tempfile`: The primary exiftool backend (`pyexiftool` batch mode, DEV-16) communicates via stdin/stdout pipes and requires no temporary files. The fallback backend (`subprocess.run()` + argfile) uses `tempfile.NamedTemporaryFile` with automatic cleanup for writing exiftool argument files — a

cross-platform approach that supersedes the original's `TempOpen`/`TempClose` pattern. The `TempOpen`/`TempClose` functions themselves are not carried forward (DEV-05, §2.6).

Additional stdlib modules used by entry points and packaging

Standard library module	Consuming module(s)	Purpose
`sys`	`cli/main.py`, `gui/app.py`	`sys.stdout` for JSON output, `sys.stderr` for log output, `sys.exit()` for exit code propagation.
`queue`	`gui/app.py`	`queue.Queue` for thread-safe communication between the indexing worker thread and the GUI main thread (§10.5).
`threading`	`gui/app.py`	`threading.Thread` for running the indexing operation off the main thread to keep the GUI responsive (§10.5).
`platform`	`core/timestamps.py`	`platform.system()` for platform-conditional creation-time extraction (`st_birthtime` on macOS, `st_ctime` on Windows, fallback on Linux) (§6.5).
`functools`	`core/exif.py`, `config/defaults.py`	`functools.lru_cache` for caching the exiftool availability probe result (§12.5).
`collections`	`core/serializer.py`	`collections.OrderedDict` for placing `schema_version` first in serialized output (§6.9).

12.3. Third-Party Python Packages

The tool declares four required runtime dependencies in `[project.dependencies]` — a bare `pip install shruggie-indexer` installs all four. Each was promoted from optional to required because it provides functionality that would otherwise require significant reimplementation, degrade correctness (Unicode safety), or sacrifice order-of-magnitude performance gains that affect the tool's primary value proposition:

Package	Version	Justification for required status
`click`	`>=8.1`	Replaces manual argparse reimplementation of decorator-based option groups, mutual exclusion, and composable help text. The CLI is a primary delivery surface (G3).
`orjson`	`>=3.9`	Eliminates `dataclasses.asdict()` overhead and provides 5–10× serialization speedup. JSON output is the tool's sole product — serialization performance is core, not peripheral.
`pyexiftool`	`>=0.5`	Enables `-stay_open` batch mode (DEV-16) for ~100× faster metadata extraction. Also provides inherent Unicode filename safety via stdin pipe protocol, eliminating the argfile character-encoding risks documented in exiftool's FAQ §18.
`tqdm`	`>=4.65`	Provides progress bar display for interactive CLI sessions. Replaces log-line milestone reporting with a real-time visual progress indicator that is expected by users processing large directory trees.

Additional third-party packages are declared as optional extras in `[project.optional-dependencies]` for the GUI delivery surface and for development/testing:

```
# pyproject.toml (illustrative excerpt — see §13.2 for the full file)

[project.dependencies]
click = ">=8.1"
orjson = ">=3.9"
pyexiftool = ">=0.5"
tqdm = ">=4.65"

[project.optional-dependencies]
gui = ["customtkinter>=5.2"]
dev = [
    "pytest>=7.0",
    "pytest-cov>=4.0",
    "jsonschema>=4.17",
    "pydantic>=2.0",
    "ruff>=0.3",
    "rich>=13.0",
]
all = ["shruggie-indexer[gui]"]
```

Per-package details — required dependencies

`click` (required)

Field	Value
Version constraint	`>=8.1`
Purpose	CLI argument parsing, option groups, help text generation, mutual exclusion enforcement (§8)
Imported by	`cli/main.py` exclusively
Why required	`click` provides decorator-based option declaration, automatic mutual exclusion (`@click.option(cls=MutuallyExclusiveOption)`), typed parameter conversion, and composable help text groups — all of which the CLI requires (§8). Using `argparse` would require reimplementing these capabilities manually. The CLI is one of three primary delivery surfaces (G3) and is installed by default — there is no use case for installing the package without CLI capability.

`orjson` (required)

Field	Value
Version constraint	`>=3.9`
Purpose	High-performance JSON serialization, native dataclass support without `dataclasses.asdict()` overhead (§6.9)
Imported by	`core/serializer.py`
Fallback	The serializer retains a `json.dumps()` fallback path behind a runtime check (`if orjson is not None:`). This fallback ensures the tool remains functional if `orjson` is uninstalled or cannot be imported, but it is not a supported configuration — the fallback exists for resilience, not as a design target.
Why required	JSON output is the tool's sole product. `orjson` eliminates the `dataclasses.asdict()` deep-copy overhead (which can take 1–2 seconds and double memory usage for 10,000-entry trees) and provides 5–10× faster serialization. Pre-built wheels are available for all supported platforms (Windows, Linux, macOS on x86-64 and ARM64).

`pyexiftool` (required)

Field	Value
Version constraint	`>=0.5`
Purpose	Persistent exiftool process for batch metadata extraction via `-stay_open` mode, with inherent Unicode filename safety via stdin pipe protocol (DEV-16, §6.6)
Imported by	`core/exif.py`
Fallback	The exif module retains a `subprocess.run()` + argfile fallback backend (see §6.6, Backend selection logic). This fallback is used when `pyexiftool` cannot maintain a stable connection to the exiftool process, but it is not a supported primary configuration.
Why required	Exiftool invocation is the dominant per-file cost in the indexing pipeline (§17.5). The `-stay_open` batch mode reduces per-file cost from 200–500 ms to 20–50 ms — a ~10× improvement that is the single largest performance optimization available. Additionally, the stdin pipe protocol inherits the Unicode safety guarantees of exiftool's argfile interface (FAQ §18), resolving the special-character argument-passing risks that motivated the original's Base64 encoding pipeline.

`tqdm` (required)

Field	Value
Version constraint	`>=4.65`
Purpose	Progress bar display for interactive CLI sessions (§11.6)
Imported by	`cli/main.py`
Why required	Real-time progress feedback is a baseline expectation for a CLI tool that processes potentially thousands of files. The log-line milestone approach is retained as a fallback for non-TTY environments (piped output, CI), but the `tqdm` progress bar is the standard interactive experience. `tqdm` is a pure-Python package with no compilation requirements and minimal footprint.

Per-package details — optional dependencies

`customtkinter` (extra: `gui`)

Field	Value
Version constraint	`>=5.2`
Purpose	Desktop GUI widget toolkit (§10)
Imported by	`gui/app.py` exclusively
Import guard	`ImportError` caught at the GUI entry point, with a message directing the user to `pip install shruggie-indexer[gui]`.
Transitive dependencies	`customtkinter` depends on `darkdetect` and `packaging`. These are installed automatically as transitive dependencies and are not interacted with directly by any `shruggie-indexer` module.

`rich` (extra: `dev`)

Field	Value
Version constraint	`>=13.0`
Purpose	Colorized log output via `rich.logging.RichHandler` as an optional enhancement to the CLI's stderr log stream (§11.1)
Imported by	`cli/main.py`
Import guard	`try: from rich.logging import RichHandler` / `except ImportError: RichHandler = None`. The CLI uses `RichHandler` as the stderr handler if available, falling back to the standard `logging.StreamHandler` otherwise.
Why optional	Visual enhancement, not functional requirement. The standard `StreamHandler` with the format string defined in §11.1 produces perfectly adequate log output. `rich` adds syntax highlighting, automatic log-level coloring, and improved traceback formatting, which are valuable during development but not required for production use.

`jsonschema` (extra: `dev`)

Field	Value
Version constraint	`>=4.17`
Purpose	Draft-07 JSON Schema validation for output conformance tests (§5.12, §14.4)
Imported by	Test modules only (`tests/`)
Import guard	None — `jsonschema` is a test dependency, not a runtime dependency. It is imported unconditionally in test modules. The test suite SHOULD fail with a clear `ImportError` if `jsonschema` is not installed, directing the developer to `pip install shruggie-indexer[dev]`.

`pydantic` (extra: `dev`)

Field	Value
Version constraint	`>=2.0`
Purpose	Optional runtime type validation models for consumers who ingest index output from untrusted sources (§5.12). Also used in test modules for strict schema validation.
Imported by	`models/schema.py` (behind `TYPE_CHECKING` and a runtime import guard)
Import guard	The Pydantic models in `models/schema.py` are defined behind a conditional block: `if TYPE_CHECKING or _PYDANTIC_AVAILABLE:`. The `_PYDANTIC_AVAILABLE` flag is set via `try: import pydantic` at the top of the module. Core engine modules never import the Pydantic models — they use the stdlib `dataclass` definitions exclusively.

`pytest`, **`pytest-cov`**, **`ruff`** (extra: `dev`)

These are development-only tools that are not imported by any runtime module. They are included in the `dev` extra for contributor convenience:

Package	Version constraint	Purpose
`pytest`	`>=7.0`	Test runner (§14)
`pytest-cov`	`>=4.0`	Coverage reporting for test runs
`ruff`	`>=0.3`	Linter and formatter, configured in `pyproject.toml` (§13.2)

12.4. Eliminated Original Dependencies

The original `MakeIndex` and its dependency tree consumed two external binaries, eight top-level pslib functions, six global variables, and approximately 60 nested sub-functions. The Python implementation eliminates the majority of these through the standard library, architectural consolidation, or deliberate scope reduction. This subsection provides the complete elimination manifest — every original dependency, its purpose, and what replaces it (or why it is dropped).

Eliminated external binaries

Original binary	Original purpose	Replacement	Deviation
`jq`	JSON parsing and key filtering of exiftool output (invoked in `GetFileExifRun` and `MetaFileRead-Data-ReadJson`)	`json.loads()` for parsing; dict comprehension for key filtering. Zero-dependency, cross-platform, no subprocess overhead.	DEV-06
`certutil`	Base64 encoding of binary sidecar file contents (invoked in `MetaFileRead-Data-Base64Encode`). Windows-only system utility.	`base64.b64encode()` from the standard library. Cross-platform, no subprocess overhead, no temporary file needed.	DEV-05 (part of the broader Base64 pipeline elimination)

Eliminated pslib functions

Original function	Original purpose	Replacement	Deviation
`Base64DecodeString`	Decodes Base64-encoded exiftool argument strings at runtime. Uses an OpsCode dispatch pattern across four encoding/URL-decode combinations.	Eliminated entirely. Exiftool arguments are defined as plain Python string lists — no encoding, no decoding, no dispatch.	DEV-05
`Date2UnixTime`	Converts formatted date strings to Unix timestamps via a three-stage pipeline: format-code resolution (calling `Date2FormatCode` externally, then falling back to an internal digit-counting heuristic via `Date2UnixTimeSquash` → `Date2UnixTimeCountDigits` → `Date2UnixTimeFormatCode`), then `[DateTimeOffset]::ParseExact().ToUnixTimeMilliseconds()`.	`int(os.stat_result.st_mtime * 1000)` for Unix milliseconds. `datetime.fromtimestamp().isoformat()` for ISO 8601 strings. Both derived directly from `os.stat()` float values — no string formatting, no reparsing, no format-code guessing.	DEV-07
`Date2FormatCode`	Analyzes date string structure and returns a .NET format code. Called by `Date2UnixTime` as its primary format-detection strategy.	Eliminated along with `Date2UnixTime`. The tool never converts date strings — it works with numeric timestamps from the filesystem directly.	DEV-07
`DirectoryId` (as a standalone function)	Generates directory identifiers by hashing directory name + parent name with four algorithms. Defines 7 internal sub-functions: `DirectoryId-GetName`, `DirectoryId-HashString`, `DirectoryId-HashString-Md5`, `-Sha1`, `-Sha256`, `-Sha512`, `DirectoryId-ParentName`, `DirectoryId-ResolvePath`.	`core/hashing.hash_string()` for all string hashing. `core/paths.extract_components()` for name/parent extraction. `core/hashing.compute_directory_id()` for the two-layer hash+concatenate+hash identity algorithm. The identity algorithm is preserved; the seven sub-functions are replaced by two shared utility functions.	DEV-01
`FileId` (as a standalone function)	Generates file identifiers by hashing file content (or name for symlinks) with up to four algorithms. Defines 10 internal sub-functions: `FileId-GetName`, `FileId-HashMd5`, `-HashMd5-String`, `-HashSha1`, `-HashSha1-String`, `-HashSha256`, `-HashSha256-String`, `-HashSha512`, `-HashSha512-String`, `FileId-ResolvePath`.	`core/hashing.hash_file()` for content hashing with single-pass multi-algorithm computation. `core/hashing.hash_string()` for name hashing. `core/paths.resolve_path()` for path resolution. The ten sub-functions are replaced by two shared utility functions.	DEV-01, DEV-02

Original function	Original purpose	Replacement	Deviation
`MetaFileRead` (as a standalone function)	Reads and parses sidecar metadata files. Defines 16 internal sub-functions for type detection, parent resolution, data reading (JSON, text, binary, link), and hashing. Depends on `certutil`, `jq`, `Lnk2Path`, `UrlFile2Url`, and `ValidateIsJson`.	`core/sidecar.py` reimplements the behavioral contract using stdlib: `json.loads()` replaces `jq`, `base64.b64encode()` replaces `certutil`, `hashlib` replaces the internal hash functions. `Lnk2Path` and `UrlFile2Url` are not ported — `.lnk` and `.url` are Windows-specific shortcut formats that are treated as opaque binary data in the cross-platform implementation (Base64-encoded when encountered as sidecar content).	DEV-05, DEV-06
`TempOpen`	Creates a temporary file with a UUID-based name in a hardcoded pslib temp directory (`C:\bin\pslib\temp`).	Eliminated. The only consumer was the exiftool argument-file pipeline, which is itself eliminated (DEV-05). If temporary files are needed in the future, `tempfile.NamedTemporaryFile` is the stdlib replacement.	DEV-05
`TempClose`	Deletes a temporary file created by `TempOpen`. Includes a `ForceAll` mode for batch cleanup of the pslib temp directory.	Eliminated along with `TempOpen`. Python's `tempfile` context managers handle cleanup automatically via `__exit__`.	DEV-05
`Vbs`	Centralized structured logging function: severity normalization, colorized `Write-Host` output, manual call-stack compression, session ID embedding, monthly log file rotation, log directory bootstrapping. Called by every function in the dependency tree.	Python's `logging` standard library module. Named loggers, hierarchical filtering, pluggable formatters and handlers, and automatic caller identification replace 100% of the `Vbs` implementation. See §11.	DEV-08

Eliminated pslib helper functions (not called directly by MakeIndex, DEV-13)

Original function	Original purpose	Status in port
`ValidateIsLink`	Listed as a dependency in the `MakeIndex` docstring but never actually invoked. `FileId` and `DirectoryId` perform symlink detection inline via `ReparsePoint` attribute check.	Not ported. Dead code in the original. Symlink detection uses `Path.is_symlink()`.
`ValidateIsFile`	Validates that a path references an existing file. Called by `MetaFileRead`.	Not ported as a standalone function. Replaced by `Path.is_file()` calls inline where needed.
`ValidateIsJson`	Validates whether a file contains valid JSON. Called by `MetaFileRead-Data`.	Not ported as a standalone function. Replaced by a try/except around `json.loads()` — the Pythonic pattern for JSON validation.
`Lnk2Path`	Resolves Windows `.lnk` shortcut files to their target paths. Called by `MetaFileRead-Data-ReadLink`.	Not ported. `.lnk` parsing requires either a Windows COM interface or a third-party library (`pylnk3`). In the cross-platform implementation, `.lnk` files encountered as sidecar content are treated as opaque binary data and Base64-encoded. A post-MVP enhancement could add `.lnk` resolution on Windows via an optional dependency.
`UrlFile2Url`	Extracts the URL from Windows `.url` internet shortcut files. Called by `MetaFileRead-Data-ReadLink`.	Not ported as a standalone function. `.url` files are simple INI-format text files; the URL is extracted with a regex or `configparser` inline in the sidecar reader. This is a trivial operation that does not warrant a separate function.
`UpdateFunctionStack`	Maintains a colon-delimited call-stack string for `Vbs` logging (e.g., `"MakeIndex:MakeObject:GetFileExif"`). Called by every internal sub-function.	Not ported. Python's `logging` framework provides automatic caller identification via `%(name)s`, `%(funcName)s`, and `%(lineno)s` format tokens. Manual call-stack bookkeeping is unnecessary.
`VariableStringify`	Converts PowerShell variables to string representations, handling `$null` and empty values. Called by `MakeObject`.	Not ported. Python's `str()`, `repr()`, and the `or` pattern (`value or default`) cover all cases handled by this function.

Eliminated global state

Original global variable	Original purpose	Replacement

Original global variable	Original purpose	Replacement
`$global:MetadataFileParser`	Configuration object governing metadata file parsing: exiftool exclusion lists, sidecar suffix patterns, type identification regexes, extension group classifications.	`IndexerConfig` dataclass (§7.1) — loaded from compiled defaults and optional TOML configuration files, passed as an explicit parameter through the call chain. No global state.
`$global:ExiftoolRejectList`	Runtime copy of `$MetadataFileParser.Exiftool.Exclude`, promoted to global scope for access by deeply nested sub-functions.	`config.exiftool_exclude_extensions` field on `IndexerConfig`, threaded explicitly to `core/exif.py`.
`$global:MetaSuffixInclude`, `$global:MetaSuffixIncludeString`	Runtime copies of sidecar include patterns, promoted to global scope.	`config.sidecar_include_patterns` field on `IndexerConfig`, threaded explicitly to `core/sidecar.py`.
`$global:MetaSuffixExclude`, `$global:MetaSuffixExcludeString`	Runtime copies of sidecar exclude patterns, promoted to global scope.	`config.sidecar_exclude_patterns` field on `IndexerConfig`, threaded explicitly to `core/sidecar.py`.
`$global:DeleteQueue`	Accumulates sidecar file paths for batch deletion when `MetaMergeDelete` is active. Initialized as empty array, populated during traversal, drained after indexing completes.	A local `list[Path]` managed by `core/entry.py`'s orchestrator function, returned to the caller as part of the result. No global mutation. See §6.8.
`$Sep`	Directory separator character. Used in path construction for renamed files.	`pathlib`'s `/` operator and `os.sep`. Manual separator handling is eliminated entirely (§6.2).
`$D_PSLIB_TEMP`	Hardcoded temp directory path (`C:\bin\pslib\temp`).	Eliminated. No temporary files are used in the core pipeline (DEV-05).
`$D_PSLIB_LOGS`	Hardcoded log directory path (`C:\bin\pslib\logs`).	Eliminated. The tool does not write log files by default (§11.1, Principle 3).
`$LibSessionID`	Session GUID generated at pslib script load time, embedded in every `Vbs` log entry.	`uuid.uuid4().hex[:8]` generated per invocation in the CLI/GUI entry point and injected via the `SessionFilter` (§11.4). Scoped to the invocation, not global.

Summary of elimination impact

The dependency elimination is substantial in both breadth and depth:

Category	Original count	Ported	Eliminated	Elimination rate
External binaries	3 (`exiftool`, `jq`, `certutil`)	1 (`exiftool`)	2	67%
Top-level pslib functions	8	0 (all absorbed into core modules)	8	100%
Nested sub-functions across all dependencies	~60	0 (replaced by ~10 shared utility functions)	~60	100%
Global variables	9	0	9	100%

The net effect is a dependency tree that is narrower (one external binary instead of three), shallower (no nested sub-function hierarchies), explicit (all dependencies declared in `pyproject.toml` or imported at module level), and stateless (no global variable mutation).

12.5. Dependency Verification at Runtime

The tool verifies dependency availability at runtime rather than failing silently or producing confusing errors deep in the call stack. Verification follows two patterns: proactive probing for the external binary, and import-guarded fallback for optional Python packages.

Exiftool binary probing

The `core/exif.py` module probes for `exiftool` availability exactly once per process lifetime using `shutil.which()`. The result is cached via `functools.lru_cache` to avoid repeated filesystem lookups:

```
# core/exif.py (illustrative — not the exact implementation)

import shutil
```

```
import functools
import logging

logger = logging.getLogger(__name__)

@functools.lru_cache(maxsize=1)
def _exiftool_available() -> bool:
    """Check whether exiftool is on PATH. Cached for process lifetime."""
    available = shutil.which("exiftool") is not None
    if not available:
        logger.warning(
            "exiftool not found on PATH. "
            "Embedded metadata extraction will be skipped for all files. "
            "Install exiftool from https://exiftool.org/"
        )
    return available
```

The probe is invoked lazily — on the first call to the exiftool extraction function, not at module import time. This means that importing `shruggie_indexer` as a library does not trigger the probe, and consumers who never call metadata extraction never see the warning. The CLI and GUI entry points do not need to check exiftool availability explicitly — the probe fires automatically when the first eligible file is encountered during indexing.

> **Historical note:** The original's `GetFileExifRun` invoked exiftool directly for every eligible file without prior availability checking. If exiftool was missing, every invocation produced a separate error, resulting in N errors for N files. The probe-once approach emits one warning for the entire invocation and avoids spawning N doomed subprocesses.

Optional Python package import guards

Optional third-party packages follow the try/except import pattern. The pattern has two variants depending on the failure mode:

Silent fallback — for performance-tier packages (`orjson`, `pyexiftool`, `tqdm`, `rich`) where a stdlib equivalent exists:

```
# core/serializer.py
try:
    import orjson
except ImportError:
    orjson = None  # type: ignore[assignment]

def serialize_entry(entry: IndexEntry, ...) -> str:
    if orjson is not None:
        return orjson.dumps(entry_dict, option=orjson.OPT_INDENT_2).decode()
    return json.dumps(entry_dict, indent=2, ensure_ascii=False)
```

The consumer never knows or cares which serializer was used — the output is identical (modulo insignificant whitespace differences). The selection is logged at `DEBUG` level for diagnostic purposes.

Hard failure with guidance — for surface-specific packages (`customtkinter` for the GUI) where no fallback exists, or as a resilience guard for required packages (`click`) that should always be installed:

```
# __main__.py
def main() -> None:
    try:
        from shruggie_indexer.cli.main import main as cli_main
    except ImportError:
        print(
            "The CLI requires the 'click' package.\n"
            "Install it with: pip install shruggie-indexer",
            file=sys.stderr,
        )
        sys.exit(1)
    cli_main()
```

The error message is specific, actionable, and includes the exact `pip install` command needed. This is a deliberate UX choice: a raw `ImportError` traceback pointing at `import click` conveys the same information but requires the user to diagnose the missing package themselves.

Dependency verification summary

Dependency	Verification method	Timing	Failure mode

Dependency	Verification method	Timing	Failure mode
`exiftool`	`shutil.which()`, cached	Lazy (first exif extraction call)	Warning + graceful degradation (null metadata)
`click`	`try: import`	CLI entry point invocation	Hard error with install instructions
`customtkinter`	`try: import`	GUI entry point invocation	Hard error with install instructions
`orjson`	`try: import`	Module load of `core/serializer.py`	Silent fallback to `json.dumps()`
`pyexiftool`	`try: import`	Module load of `core/exif.py`	Silent fallback to `subprocess.run()`
`tqdm`	`try: import`	CLI progress callback setup	Silent fallback to log-line milestones
`rich`	`try: import`	CLI logging configuration	Silent fallback to `logging.StreamHandler`
`pydantic`	`try: import`	Module load of `models/schema.py`	Pydantic models unavailable; stdlib dataclasses used
`jsonschema`	Direct import (test only)	Test module load	`ImportError` — developer installs `[dev]` extra

13. Packaging and Distribution

This section defines how `shruggie-indexer` is packaged, built, versioned, and distributed — from the `pyproject.toml` configuration that governs metadata and dependency declarations, through the entry points that connect the installed package to its three delivery surfaces (CLI, GUI, library), to the PyInstaller-based standalone executable builds and the GitHub Releases workflow that produces downloadable artifacts. It is the normative reference for every field in `pyproject.toml`, every entry point registration, and every artifact that a release produces.

The packaging conventions follow those established by `shruggie-feedtools` (§1.5, External References). Where this section does not explicitly define a convention — such as the exact ruff rule set or the pytest marker registration syntax — the `shruggie-feedtools` `pyproject.toml` is the normative reference for project scaffolding. Where the indexer's needs diverge from feedtools (additional extras, a second PyInstaller target for the GUI), the divergence is documented explicitly.

Key constraint (reiterated from §2.1): This project is not published to PyPI. End users download pre-built executables from GitHub Releases. The `pip install` workflow — including editable installs, extras, and the `[project]` metadata table — serves contributors setting up a local development environment, the CI pipeline building release artifacts, and hypothetical future library consumers who install directly from the GitHub repository URL. The `pyproject.toml` is therefore structured to support both `pip install -e ".[dev,cli,gui]"` for contributors and `pyinstaller` for release builds, but it does not include PyPI-specific fields (classifiers, project URLs) that would only matter for a published package.

13.1. Package Metadata

The `[project]` table in `pyproject.toml` declares the package identity, authorship, licensing, and compatibility metadata. These fields are consumed by the build backend (`hatchling`), by `pip install` for dependency resolution, and by the `--version` flag for version display.

Field	Value	Notes
`name`	`"shruggie-indexer"`	The distribution name. Hyphens are normalized to underscores for the import name (`shruggie_indexer`).
`description`	`"Filesystem indexer with hash-based identity, metadata extraction, and structured JSON output"`	Single-line summary.
`readme`	`"README.md"`	Points to the repository root README.
`license`	`"Apache-2.0"`	SPDX license identifier, referencing the `LICENSE` file at the repository root (§3.1).
`requires-python`	`">=3.12"`	Matches the Python version floor established in §2.5. The `>=` constraint (not `==`) allows any 3.12+ interpreter.
`authors`	`[{name = "William Thompson"}]`	Single author for the MVP.
`keywords`	`["indexer", "filesystem", "metadata", "exif", "hashing"]`	Discovery keywords — relevant if the package is ever published, harmless otherwise.
`dynamic`	`["version"]`	The version string is read from `src/shruggie_indexer/_version.py` by hatchling's version plugin (§13.6). It is not hardcoded in `pyproject.toml`.

Fields deliberately omitted:

- `classifiers` — PyPI trove classifiers are not included because the package is not published to PyPI. If the project is ever published, classifiers should be added at that time to reflect the license, supported Python versions, operating systems, and development status.
- `project-urls` — Same rationale. URLs for the repository, documentation, and issue tracker can be added if the project is published.

13.2. pyproject.toml Configuration

The complete `pyproject.toml` is the single configuration file for the build system, package metadata, dependency declarations, entry points, and tool settings. The following is the canonical content — an implementer SHOULD produce a file equivalent to this, though field ordering within tables may vary.

```toml
# ─── Build system ─────────────────────────────────────────────────────

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

# ─── Package metadata ─────────────────────────────────────────────────

[project]
name = "shruggie-indexer"
description = "Filesystem indexer with hash-based identity, metadata extraction, and structured JSON output"
readme = "README.md"
license = "Apache-2.0"
requires-python = ">=3.12"
authors = [{name = "William Thompson"}]
keywords = ["indexer", "filesystem", "metadata", "exif", "hashing"]
dynamic = ["version"]
dependencies = [
    "click>=8.1",
    "orjson>=3.9",
    "pyexiftool>=0.5",
    "tqdm>=4.65",
]

[project.optional-dependencies]
gui = ["customtkinter>=5.2"]
dev = [
    "pytest>=7.0",
    "pytest-cov>=4.0",
    "jsonschema>=4.17",
    "pydantic>=2.0",
    "ruff>=0.3",
    "rich>=13.0",
]
all = ["shruggie-indexer[gui]"]

# ─── Entry points ─────────────────────────────────────────────────────

[project.scripts]
shruggie-indexer = "shruggie_indexer.cli.main:main"

[project.gui-scripts]
shruggie-indexer-gui = "shruggie_indexer.gui.app:main"

# ─── Hatchling configuration ──────────────────────────────────────────

[tool.hatch.version]
path = "src/shruggie_indexer/_version.py"

[tool.hatch.build.targets.wheel]
packages = ["src/shruggie_indexer"]

# ─── Pytest ───────────────────────────────────────────────────────────

[tool.pytest.ini_options]
testpaths = ["tests"]
markers = [
    "slow: marks tests as slow (deselect with '-m \"not slow\"')",
    "platform_windows: marks tests that only run on Windows",
    "platform_linux: marks tests that only run on Linux",
    "platform_macos: marks tests that only run on macOS",
    "requires_exiftool: marks tests that require exiftool on PATH",
]

# ─── Ruff ─────────────────────────────────────────────────────────────

[tool.ruff]
target-version = "py312"
line-length = 100
src = ["src"]
```

```toml
[tool.ruff.lint]
select = [
    "E",    # pycodestyle errors
    "W",    # pycodestyle warnings
    "F",    # pyflakes
    "I",    # isort
    "N",    # pep8-naming
    "UP",   # pyupgrade
    "B",    # flake8-bugbear
    "SIM",  # flake8-simplify
    "TCH",  # flake8-type-checking
    "RUF",  # ruff-specific rules
]

[tool.ruff.lint.isort]
known-first-party = ["shruggie_indexer"]

# ─── PyInstaller (reference only — actual builds use .spec files) ──────────

[tool.pyinstaller]
# This table is not consumed by pyinstaller directly.
# It is included as a documentation aid; the build scripts
# (scripts/build.ps1, scripts/build.sh) invoke pyinstaller
# with explicit arguments or .spec files. See §13.4.
```

Notable design decisions

Four required runtime dependencies. The `[project.dependencies]` list declares `click`, `orjson`, `pyexiftool`, and `tqdm` (§12.3). A bare `pip install shruggie-indexer` installs all four. This is the implementation of design goal G5 (§2.3): the dependency set is small and deliberately chosen — each package replaces functionality that would otherwise require significant reimplementation, degrade correctness, or sacrifice order-of-magnitude performance. The GUI package (`customtkinter`) remains optional.

`[project.scripts]` vs. `[project.gui-scripts]`. The CLI entry point is registered under `[project.scripts]`, which creates a platform-appropriate console script wrapper (`shruggie-indexer` on Linux/macOS, `shruggie-indexer.exe` on Windows). The GUI entry point is registered under `[project.gui-scripts]`, which on Windows creates a wrapper that does not allocate a console window — this prevents the "flash of black console window" that would occur if a GUI application were launched from a `[project.scripts]` entry point. On Linux and macOS, `[project.gui-scripts]` behaves identically to `[project.scripts]`. The distinction matters only for the `pip install` development workflow; the PyInstaller-built standalone executables handle console/no-console via their own `--windowed` flag (§13.4).

`[tool.hatch.version]` path. The version string is read from `src/shruggie_indexer/_version.py` by hatchling's version plugin. This is a single-source-of-truth pattern: the version is defined in exactly one place (the `_version.py` file) and read by `pyproject.toml` (via hatchling), by `__init__.py` (via import), and by the CLI `--version` flag (via the same import). See §13.6 for the full version management strategy.

Ruff configuration scope. The ruff rule set is deliberately conservative for the MVP — it enables the most universally beneficial lint rules without imposing subjective style preferences (no `D` docstring enforcement, no `ANN` annotation enforcement, no `PT` pytest style rules). The rule set can be expanded incrementally as the codebase matures. The `target-version = "py312"` setting ensures ruff applies pyupgrade transformations appropriate to the Python 3.12 floor.

`[tool.pyinstaller]` as documentation. The `pyproject.toml` includes a `[tool.pyinstaller]` comment block as a breadcrumb for implementers. PyInstaller does not natively read configuration from `pyproject.toml` — it uses `.spec` files or command-line arguments. The actual build configuration lives in the build scripts (§3.5) and the `.spec` files described in §13.4.

13.3. Entry Points and Console Scripts

The package registers two entry points — one for the CLI and one for the GUI — plus the `python -m` module execution path. All three routes converge on the same core library.

CLI entry point: `shruggie-indexer`

Registration	**`[project.scripts]` in `pyproject.toml`**
Entry point string	`shruggie-indexer = "shruggie_indexer.cli.main:main"`
Invocation	`shruggie-indexer [OPTIONS] [TARGET]`
Requires	`click` (required runtime dependency, installed automatically)
Failure without dependency	`ImportError` caught in `__main__.py`; prints install instructions to stderr; exits with code 1

When `pip install -e .` is executed, pip creates a wrapper script named `shruggie-indexer` (or `shruggie-indexer.exe` on Windows) in the virtual environment's `bin/` (or `Scripts/`) directory. This wrapper imports `shruggie_indexer.cli.main` and calls its `main()` function. The wrapper is a platform-

native console script — on Linux/macOS it is a small Python script with a shebang line pointing to the venv interpreter; on Windows it is a `.exe` launcher generated by pip.

GUI entry point: `shruggie-indexer-gui`

Registration	`[project.gui-scripts]` **in** `pyproject.toml`
Entry point string	`shruggie-indexer-gui = "shruggie_indexer.gui.app:main"`
Invocation	`shruggie-indexer-gui` (no arguments for the MVP)
Requires	`customtkinter` (installed via the `gui` extra)
Failure without dependency	`ImportError` caught at the GUI entry point; prints install instructions to stderr; exits with code 1

The `[project.gui-scripts]` registration is functionally identical to `[project.scripts]` on Linux and macOS. On Windows, it creates a `shruggie-indexer-gui.exe` wrapper that suppresses console window creation — the GUI application launches without a visible terminal window. This is the standard mechanism for Python GUI applications on Windows and requires no special handling in the source code.

Module execution: `python -m shruggie_indexer`

Mechanism	`__main__.py` **in the top-level package**
Invocation	`python -m shruggie_indexer [OPTIONS] [TARGET]`
Behavior	Identical to the `shruggie-indexer` console script
Implementation	Imports and calls `cli.main.main()` with the same `ImportError` guard

The `__main__.py` file (§3.2) provides the `python -m` execution path. It contains no logic beyond importing and calling the CLI entry point:

```
# src/shruggie_indexer/__main__.py

import sys

def main() -> None:
    try:
        from shruggie_indexer.cli.main import main as cli_main
    except ImportError:
        print(
            "The CLI requires the 'click' package.\n"
            "Install it with: pip install shruggie-indexer",
            file=sys.stderr,
        )
        sys.exit(1)
    cli_main()

if __name__ == "__main__":
    main()
```

This path exists for two reasons: it provides a universal invocation mechanism that works even if the console script wrapper was not installed (e.g., when the package is installed in a non-standard way), and it enables `python -m shruggie_indexer` as a fallback for environments where modifying `PATH` to include the venv's `bin/` directory is inconvenient.

Entry point summary

Invocation	**Requires extras**	**Target function**	**Console window**
`shruggie-indexer`	`cli`	`shruggie_indexer.cli.main:main`	Yes
`shruggie-indexer-gui`	`gui`	`shruggie_indexer.gui.app:main`	No (Windows); Yes (Linux/macOS)
`python -m shruggie_indexer`	`cli`	`shruggie_indexer.cli.main:main`	Yes

13.4. Standalone Executable Builds

End users do not install `shruggie-indexer` via pip. They download standalone executables from GitHub Releases — pre-built binaries that bundle the Python interpreter, all required dependencies, and the application code into a single distributable artifact. The build tool is PyInstaller.

Build targets

Each release produces two executables per platform: one for the CLI and one for the GUI. The two are built from separate PyInstaller configurations because they have different entry points, dependency sets, and windowing requirements.

Target	Entry module	PyInstaller mode	Console window	Output filename (Windows)	Output filename (Linux/macOS)
CLI	`src/shruggie_indexer/cli/main.py`	`--onefile --console`	Yes	`shruggie-indexer.exe`	`shruggie-indexer`
GUI	`src/shruggie_indexer/gui/app.py`	`--onefile --windowed`	No	`shruggie-indexer-gui.exe`	`shruggie-indexer-gui`

`--onefile` mode. Both targets use PyInstaller's one-file bundle mode, which produces a single executable that extracts itself to a temporary directory at runtime. This is the simplest distribution format — the user downloads one file and runs it. The alternative `--onedir` mode (which produces a directory of files) is available as a fallback if one-file extraction causes issues on specific platforms, but `--onefile` is the default for releases.

`--windowed` vs. `--console`. The GUI target uses `--windowed` (aliased as `--noconsole` and `-w`) to suppress console window creation on Windows. The CLI target uses `--console` (the default) to ensure stdin/stdout/stderr are connected to the terminal. On Linux and macOS, `--windowed` has no behavioral effect — both targets produce standard ELF/Mach-O executables.

PyInstaller spec files

The build scripts (§3.5) invoke PyInstaller using `.spec` files rather than raw command-line arguments. Spec files provide reproducible, version-controlled build configurations and allow platform-conditional logic (e.g., including platform-specific hidden imports or data files).

The spec files live at the repository root alongside `pyproject.toml`:

```
shruggie-indexer/
├── shruggie-indexer-cli.spec
├── shruggie-indexer-gui.spec
├── pyproject.toml
└── ...
```

CLI spec file (`shruggie-indexer-cli.spec`):

```
# shruggie-indexer-cli.spec
# PyInstaller spec file for the CLI executable.

import sys
from pathlib import Path

block_cipher = None
src_dir = Path("src")

a = Analysis(
    [str(src_dir / "shruggie_indexer" / "cli" / "main.py")],
    pathex=[str(src_dir)],
    binaries=[],
    datas=[],
    hiddenimports=["shruggie_indexer"],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=["customtkinter", "tkinter", "_tkinter"],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
    noarchive=False,
)

pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)

exe = EXE(
    pyz,
    a.scripts,
    a.binaries,
    a.datas,
    [],
    name="shruggie-indexer",
    debug=False,
    bootloader_ignore_signals=False,
```

```
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=True,
    disable_windowed_traceback=False,
    argv_emulation=False,
    target_arch=None,
    codesign_identity=None,
    entitlements_file=None,
)
```

GUI spec file (`shruggie-indexer-gui.spec`):

```
# shruggie-indexer-gui.spec
# PyInstaller spec file for the GUI executable.

import sys
from pathlib import Path

block_cipher = None
src_dir = Path("src")

a = Analysis(
    [str(src_dir / "shruggie_indexer" / "gui" / "app.py")],
    pathex=[str(src_dir)],
    binaries=[],
    datas=[],
    hiddenimports=["shruggie_indexer", "customtkinter"],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=["click"],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
    noarchive=False,
)

pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)

exe = EXE(
    pyz,
    a.scripts,
    a.binaries,
    a.datas,
    [],
    name="shruggie-indexer-gui",
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=False,          # <-- windowed mode for GUI
    disable_windowed_traceback=False,
    argv_emulation=False,
    target_arch=None,
    codesign_identity=None,
    entitlements_file=None,
)
```

Key spec file decisions

`excludes` **lists.** Each spec file excludes packages that the other target requires but the current target does not. The CLI spec excludes `customtkinter`, `tkinter`, and `_tkinter` — the GUI toolkit and its underlying C extension are substantial (several MB) and are never imported by the CLI. The GUI spec excludes `click` — the GUI does not use the CLI's argument parser. These exclusions reduce bundle size and eliminate false-positive hidden-import detection.

`hiddenimports`**.** PyInstaller's static analysis cannot always detect dynamic imports (e.g., the `try: import orjson` pattern in the serializer, or the `pyexiftool` batch mode backend). The `hiddenimports` list explicitly declares packages that PyInstaller should include even if they are not statically visible.

The lists shown above are the minimum set and should include the four required runtime dependencies (`click`, `orjson`, `pyexiftool`, `tqdm`). For optional packages (e.g., `customtkinter` for the GUI build), add entries if installed in the build environment. Packages listed in `hiddenimports` that are not installed in the build environment are silently skipped — they do not cause build failures.

`datas` **list.** Both spec files declare an empty `datas` list. The indexer has no bundled data files (no templates, no asset images, no embedded configuration files). If a future enhancement requires bundled data (e.g., a default configuration file or GUI icon), the `datas` list is the correct mechanism for including it.

UPX compression. Both spec files enable UPX compression (`upx=True`). UPX reduces executable size by 30–50% with minimal startup time impact. If UPX is not installed in the build environment, PyInstaller silently skips compression — the build succeeds with a larger executable. The build scripts (§3.5) SHOULD log whether UPX was available for the build.

Build invocation

The build scripts (`scripts/build.ps1` and `scripts/build.sh`, §3.5) invoke PyInstaller against both spec files:

```
# scripts/build.sh (illustrative excerpt)

#!/usr/bin/env bash
set -euo pipefail

echo "Building CLI executable..."
pyinstaller shruggie-indexer-cli.spec --distpath dist/ --workpath build/cli --clean

echo "Building GUI executable..."
pyinstaller shruggie-indexer-gui.spec --distpath dist/ --workpath build/gui --clean

echo "Build complete. Artifacts in dist/"
ls -lh dist/
```

Both executables are output to the `dist/` directory. The `--workpath` argument separates the intermediate build artifacts for each target to avoid collisions. The `--clean` flag ensures a fresh build each time, preventing stale cached analysis from causing silent inclusion/exclusion errors.

The build scripts MUST be runnable from the repository root. They assume the virtual environment is active and that PyInstaller is installed (`pip install pyinstaller`). PyInstaller is not declared as a project dependency — it is a build-time tool installed in the CI pipeline or by the developer running a local build.

Exiftool and standalone builds

The standalone executables do NOT bundle `exiftool`. Exiftool is a separate binary with its own installation and licensing requirements (§12.1). Users who want metadata extraction must install exiftool independently and ensure it is on their system `PATH`. The executables degrade gracefully when exiftool is absent (§4.5, §12.5) — all indexing operations except embedded metadata extraction function normally.

This is a deliberate decision, not an oversight. Bundling exiftool would complicate licensing (exiftool is GPL-licensed, while shruggie-indexer is Apache 2.0), increase the bundle size significantly (~25 MB for the Perl distribution), and create a maintenance burden for tracking exiftool version updates. The user's existing exiftool installation is the correct integration point.

13.5. Release Artifact Inventory

Each release produces a fixed set of artifacts, uploaded to GitHub Releases. The artifact set is per-platform — each platform's CI runner produces its own set.

Artifact	Platform	Description
`shruggie-indexer-{version}-windows-x64.exe`	Windows	CLI standalone executable
`shruggie-indexer-gui-{version}-windows-x64.exe`	Windows	GUI standalone executable
`shruggie-indexer-{version}-linux-x64`	Linux	CLI standalone executable
`shruggie-indexer-gui-{version}-linux-x64`	Linux	GUI standalone executable
`shruggie-indexer-{version}-macos-x64`	macOS (Intel)	CLI standalone executable
`shruggie-indexer-gui-{version}-macos-x64`	macOS (Intel)	GUI standalone executable
`shruggie-indexer-{version}-macos-arm64`	macOS (Apple Silicon)	CLI standalone executable
`shruggie-indexer-gui-{version}-macos-arm64`	macOS (Apple Silicon)	GUI standalone executable

The `{version}` placeholder is the version string from `_version.py` (e.g., `0.1.0`). Filenames use hyphens as separators and include the platform and architecture to disambiguate downloads on the releases page.

macOS dual-architecture builds

macOS requires two architecture variants: `x64` (Intel) and `arm64` (Apple Silicon). PyInstaller produces native executables for the architecture of the host Python interpreter — a build on an Intel Mac produces an x64 binary, a build on an Apple Silicon Mac produces an arm64 binary. Cross-compilation (building arm64 on x64 or vice versa) is not reliably supported by PyInstaller.

The CI pipeline (§13.5.1) handles this by running macOS builds on two separate runner types, one for each architecture. If GitHub Actions does not provide both runner types, the alternative is to build a universal binary using `lipo` after producing both single-architecture builds — but this approach is fragile and should only be used if separate runners are unavailable.

> **Note:** ARM64 macOS builds are a release-time consideration, not an MVP blocker. The v0.1.0 release MAY ship with x64-only macOS builds if arm64 CI runners are not available, with arm64 support added in a subsequent release. The tool runs correctly on Apple Silicon via Rosetta 2 in the interim.

Artifacts NOT included in releases

The release does NOT include:

- **Source distributions (`.tar.gz`, `.whl`).** The project is not published to PyPI (§2.1). Source installs are done via `pip install -e .` from a git clone, not from a distribution archive.
- **Checksum files.** GitHub Releases provides its own SHA-256 checksums for uploaded artifacts. Separate `.sha256` sidecar files are redundant.
- **Platform-specific installers (`.msi`, `.dmg`, `.deb`).** Standalone executables are the distribution format. Platform-specific installers are a potential post-MVP enhancement for improved user experience (e.g., Start Menu integration on Windows, Applications folder placement on macOS), but they add significant build complexity and are not required for the MVP.

13.5.1. GitHub Actions Release Pipeline

The release pipeline is defined in `.github/workflows/release.yml` (§3.1). It triggers on version tag pushes and produces the full artifact inventory described above.

Trigger

```yaml
on:
  push:
    tags:
      - "v*"
```

The pipeline triggers when a tag matching `v*` (e.g., `v0.1.0`, `v0.2.0-rc1`) is pushed to the repository. It does not trigger on branch pushes or pull requests — those are handled by a separate CI workflow (not specified in this document, as it is not a packaging concern).

Matrix strategy

The pipeline uses a matrix strategy to build across platforms:

```yaml
strategy:
  matrix:
    include:
      - os: windows-latest
        platform_suffix: windows-x64
      - os: ubuntu-latest
        platform_suffix: linux-x64
      - os: macos-13
        platform_suffix: macos-x64
      - os: macos-latest
        platform_suffix: macos-arm64
```

Each matrix entry runs on a GitHub Actions runner for the target platform. The `platform_suffix` value is interpolated into artifact filenames.

Pipeline stages

The pipeline executes the following stages on each matrix runner:

Stage 1 — Checkout and environment setup. Checks out the repository at the tag commit. Installs Python 3.12 using `actions/setup-python`. Creates a virtual environment and installs the package with all extras: `pip install -e ".[gui,dev]"`. Installs PyInstaller: `pip install pyinstaller`.

Stage 2 — Test. Runs the test suite (`pytest tests/ -m "not requires_exiftool"`) to verify that the codebase is healthy before building release artifacts. The `requires_exiftool` marker is excluded because exiftool may not be installed on all CI runners. If tests fail, the pipeline aborts — no artifacts are produced.

Stage 3 — Build. Invokes the build scripts (`scripts/build.sh` or `scripts/build.ps1`) to produce both the CLI and GUI executables via PyInstaller. The build scripts output to `dist/`.

Stage 4 — Rename artifacts. Renames the executables from their generic names (`shruggie-indexer`, `shruggie-indexer-gui`) to the versioned, platform-tagged names defined in the artifact inventory (e.g., `shruggie-indexer-0.1.0-linux-x64`). The version string is extracted from the git tag.

Stage 5 — Upload. Uploads the renamed artifacts using `actions/upload-artifact` for cross-job sharing.

Stage 6 — Release (runs once, after all matrix jobs complete). A separate job that runs after all matrix builds succeed. It downloads all uploaded artifacts and creates a GitHub Release associated with the triggering tag. The release body includes a changelog summary (manually authored or extracted from a `CHANGELOG.md` file if present) and the list of downloadable artifacts.

Pipeline design principles

Build and test on the target platform. Each platform's artifacts are built on a runner matching that platform. Cross-compilation is not used. This ensures that PyInstaller's dependency detection, binary bundling, and executable format are all native to the target.

Fail fast. If any matrix job fails (test failure, build failure, upload failure), the release job does not execute. A partial release — with artifacts for some platforms but not others — is never created.

Reproducibility. The pipeline pins the Python version (`3.12`), uses `pip install` with the version constraints from `pyproject.toml`, and builds from the exact commit referenced by the tag. Two runs of the pipeline on the same tag SHOULD produce functionally identical artifacts (byte-identical output is not guaranteed due to PyInstaller's use of timestamps and random identifiers in the bootloader).

13.6. Version Management

The version string is defined in a single location and consumed by all components that need it.

Single source of truth

The version is defined in `src/shruggie_indexer/_version.py`:

```
# src/shruggie_indexer/_version.py
__version__ = "0.1.0"
```

This is the only place the version string is written. All other version consumers read from this file:

Consumer	Mechanism
`pyproject.toml`	`[tool.hatch.version]` reads `__version__` from the file path `src/shruggie_indexer/_version.py`. Hatchling parses the file and extracts the version string at build time.
`__init__.py`	`from shruggie_indexer._version import __version__` makes the version available as `shruggie_indexer.__version__` for library consumers.
CLI `--version` flag	`@click.version_option(version=__version__)` reads the imported `__version__` attribute.
GUI window title	The GUI's `__init__` method formats the window title as `f"Shruggie Indexer v{__version__}"` using the imported attribute.
PyInstaller artifacts	The build scripts extract the version from `_version.py` (via a shell `grep`/`sed` or Python one-liner) to construct versioned artifact filenames.

Versioning scheme

The project uses semantic versioning (`MAJOR.MINOR.PATCH`):

- **MAJOR** — Incremented for breaking changes to the public API (§9.1) or the output schema (§5). The output schema and the public API are the two stability boundaries.
- **MINOR** — Incremented for backward-compatible feature additions (new CLI flags, new configuration options, new metadata types, schema additions that do not break existing consumers).
- **PATCH** — Incremented for bug fixes, documentation corrections, and internal refactoring that does not change observable behavior.

Pre-release versions use the format `MAJOR.MINOR.PATCH-rcN` (e.g., `0.1.0-rc1`) for release candidates. Pre-release versions are valid PEP 440 versions when expressed as `0.1.0rc1` (no hyphen) — the `_version.py` file uses PEP 440 format, and the git tag uses the hyphenated form for readability.

For the MVP release cycle: the project starts at `0.1.0`. During the `0.x.y` series, minor version bumps MAY include breaking changes to the public API (§9.1). Once the project reaches `1.0.0`, semantic versioning guarantees apply in full.

Version bump procedure

Version changes are a manual, deliberate action — not automated by CI. The procedure for releasing a new version is:

1. Update `__version__` in `src/shruggie_indexer/_version.py` to the new version string.
2. Commit the change with a message following the pattern: `release: v{version}`.
3. Create a git tag: `git tag v{version}`.
4. Push the commit and tag: `git push && git push --tags`.
5. The GitHub Actions release pipeline (§13.5.1) triggers on the tag push and produces the release artifacts.

The version bump is a single-file change. There is no need to update `pyproject.toml`, `__init__.py`, or any other file — they all derive the version from `_version.py`. This eliminates the class of bugs where version strings fall out of sync across multiple files.

14. Testing

This section defines the testing strategy, test categories, coverage expectations, and execution requirements for `shruggie-indexer`. It is the normative reference for what must be tested, how tests are organized, what constitutes a passing test suite, and how the test infrastructure integrates with the CI pipeline (§13.5.1) and the cross-platform build matrix.

The test directory layout and fixture structure are defined in §3.4. The pytest configuration (markers, testpaths, strict mode) is defined in `pyproject.toml` (§13.2). The test dependencies (`pytest`, `pytest-cov`, `jsonschema`, `pydantic`) are declared in the `dev` extra (§12.3). This section defines the behavioral content of those tests — what each test category verifies, what invariants it enforces, and what the expected inputs and outputs are.

Testing philosophy: Every behavioral contract defined in this specification SHOULD have a corresponding test. An AI implementation agent building a module from this specification SHOULD be able to derive the test cases from the module's behavioral contract without additional guidance.

> **Historical note:** The original `MakeIndex` has no tests of any kind — no unit tests, no integration tests, no schema conformance checks, no assertion of expected output. Correctness was validated by the author's manual inspection of output files.

> **Scope clarification:** This section describes what the tests verify and what inputs they use. It does not prescribe the exact `assert` statements or implementation details of each test function — those are derived by the implementer from the behavioral contracts in §5–§11. The section provides enough structure for an implementer to produce a complete test suite without ambiguity about coverage scope.

14.1. Testing Strategy

Test categories

The test suite is organized into four categories, each targeting a different level of abstraction:

Category	Directory	Scope	External dependencies	Mocked components
Unit	`tests/unit/`	Individual functions and modules in isolation	None	Filesystem (via `tmp_path`), exiftool (via captured responses), configuration (via fixture configs)
Integration	`tests/integration/`	Full indexing pipeline from target path to validated JSON output	Filesystem (real); exiftool (optional, skippable via marker)	None — exercises the real code path end-to-end
Conformance	`tests/conformance/`	Output structure against the canonical v2 JSON Schema	`jsonschema` package	None — validates actual serializer output
Platform	`tests/platform/`	Behaviors that vary by operating system	Filesystem (real, platform-specific)	None — exercises platform-specific code paths

Tests are not organized by source module (no `tests/core/test_hashing.py` mirroring `src/shruggie_indexer/core/hashing.py`). Instead, they are organized by test type, with a flat file layout within each category. This grouping is described in §3.4 and is driven by CI utility: the pipeline can run `pytest tests/unit/` on all platforms, `pytest tests/platform/ -m platform_linux` only on Linux, and `pytest tests/conformance/` only after a successful unit pass.

Test execution

All tests are runnable with a bare `pytest` invocation from the repository root:

```
# Run all tests
pytest

# Run only unit tests
pytest tests/unit/

# Run tests excluding those that need exiftool
pytest -m "not requires_exiftool"
```

```
# Run with coverage
pytest --cov=shruggie_indexer --cov-report=term-missing
```

The `pyproject.toml [tool.pytest.ini_options]` section (§13.2) registers the following markers:

Marker	Purpose
`slow`	Tests that take more than a few seconds (large directory trees, hashing large files). Deselectable with `-m "not slow"` for rapid iteration.
`platform_windows`	Tests that only execute on Windows. Skipped on other platforms via `pytest.mark.skipif`.
`platform_linux`	Tests that only execute on Linux.
`platform_macos`	Tests that only execute on macOS.
`requires_exiftool`	Tests that require `exiftool` to be present on `PATH`. Skipped when exiftool is not installed.

The `--strict-markers` option ensures that any marker typo (e.g., `@pytest.mark.requiers_exiftool`) causes a test collection error rather than silently creating a new marker. This catches a common class of CI bugs where a misspelled marker causes a test to run unconditionally instead of being skipped.

Fixture infrastructure

The `tests/conftest.py` file provides shared fixtures consumed across all test categories:

`tmp_path` (built-in). pytest's built-in `tmp_path` fixture provides a unique temporary directory per test. Unit and integration tests create their filesystem fixtures inside `tmp_path` to ensure isolation — no test depends on the state left by a previous test, and no test writes to the real filesystem outside its temporary directory.

`sample_file` fixture. Creates a temporary file with configurable name, content, size, and extension inside `tmp_path`. Returns the `Path` to the file. This is the primary input fixture for single-file unit tests.

`sample_tree` fixture. Creates a temporary directory hierarchy with configurable depth, breadth, and file contents. Used by traversal, recursive indexing, and integration tests. The fixture accepts a dictionary-based tree specification:

```python
# Illustrative — not the exact implementation.
@pytest.fixture
def sample_tree(tmp_path):
    def _make_tree(spec: dict, root: Path | None = None) -> Path:
        base = root or tmp_path
        for name, content in spec.items():
            path = base / name
            if isinstance(content, dict):
                path.mkdir(parents=True, exist_ok=True)
                _make_tree(content, path)
            elif isinstance(content, bytes):
                path.write_bytes(content)
            else:
                path.write_text(str(content), encoding="utf-8")
        return base
    return _make_tree
```

`default_config` fixture. Returns an `IndexerConfig` constructed from compiled defaults (§7.2) with no user overrides. Used by unit tests that need a configuration object but are not testing configuration behavior.

`mock_exiftool` fixture. Patches `subprocess.run` in `core/exif.py` to return pre-captured JSON responses from `tests/fixtures/exiftool_responses/`. Each response file is named after the input file type (e.g., `jpeg_response.json`, `png_response.json`). The fixture returns a context manager that sets up and tears down the mock. Unit tests that exercise EXIF-related code paths use this fixture; integration tests that need real exiftool invocation use the `requires_exiftool` marker instead.

`exiftool_available` fixture. A session-scoped fixture that checks `shutil.which("exiftool")` once and exposes the result as a boolean. Tests marked with `@pytest.mark.requires_exiftool` skip when this fixture returns `False`.

14.2. Unit Test Coverage

Unit tests exercise individual functions and modules in isolation. Each test file in `tests/unit/` corresponds to one source module, and each test function validates a single behavioral expectation defined in §6 (Core Operations), §7 (Configuration), or §5 (Output Schema models).

test_traversal.py

Exercises `core/traversal.list_children()` (§6.1).

Test case	Input	Expected behavior
Empty directory	`tmp_path` with no children	Returns `([], [])` — two empty lists.
Files only	Directory containing three files, no subdirectories	Returns `([file1, file2, file3], [])` sorted lexicographically.
Directories only	Directory containing two subdirectories, no files	Returns `([], [dir1, dir2])` sorted lexicographically.
Mixed content	Directory with files and subdirectories	Files and directories returned in separate sorted lists.
Exclusion filtering	Directory containing a file matching `filesystem_excludes` (e.g., `desktop.ini`, `Thumbs.db`)	Excluded items are absent from both returned lists.
Glob exclusion	Directory containing an item matching `filesystem_exclude_globs` (e.g., `.git/`)	Glob-matched items are absent.
Symlink classification	Directory containing a symlink to a file and a symlink to a directory	Symlinks appear in the appropriate list based on `follow_symlinks=False` behavior of `os.scandir()`.
Sort order stability	Directory with files named `B.txt`, `a.txt`, `C.txt`	Case-insensitive sort: `a.txt`, `B.txt`, `C.txt`.
Permission error on child	Directory where one child file is unreadable	Unreadable item is excluded with a logged warning; other items returned normally.

test_paths.py

Exercises `core/paths.py` functions (§6.2).

Test case	Input	Expected behavior
Resolve absolute path	An absolute `Path`	Returns the same path (resolved, no symlinks followed).
Resolve relative path	A relative `Path`	Returns the absolute form relative to `cwd`.
Extract components	A path like `/home/user/photos/sunset.jpg`	`name="sunset.jpg"`, `stem="sunset"`, `suffix="jpg"` (lowercase, no dot), `parent_name="photos"`.
Extension lowercasing	`FILE.JPG`	`suffix="jpg"`.
No extension	A file named `Makefile`	`suffix=None`.
Multi-dot extension	`archive.tar.gz`	`suffix="gz"` (only the final extension).
Extension validation pass	`"jpg"` against the default regex	Validation passes.
Extension validation fail	`"thisextensionistoolong"` against the default regex	Validation fails; returns `None`.
Root-level parent	`/file.txt` on Unix, `C:\file.txt` on Windows	`parent_name` is empty string.
Sidecar path construction (file)	`Path("/photos/sunset.jpg")`	Returns `/photos/sunset.jpg_meta2.json`.
Sidecar path construction (dir)	`Path("/photos/vacation")`	Returns `/photos/vacation/_directorymeta2.json`.

test_hashing.py

Exercises `core/hashing.py` functions (§6.3).

Test case	Input	Expected behavior
Hash known file content	A file with content `b"hello world"`	MD5, SHA1, SHA256, SHA512 match pre-computed reference values. Hashes are uppercase hexadecimal.
Hash empty file	A zero-byte file	Returns the well-known empty-input hash for each algorithm.
Hash string	The string `"sunset.jpg"`	MD5 and SHA256 match pre-computed values. Input is UTF-8 encoded before hashing.
Hash empty string	`""`	Returns the well-known empty-string hash for each algorithm.
Multi-algorithm single pass	A 1 MB file with `algorithms=("md5", "sha1", "sha256", "sha512")`	All four digests returned. File is read once (verifiable by mocking `open()` to count reads).

Test case	Input	Expected behavior
Default algorithms	`hash_file(path)` with no explicit `algorithms` argument	Returns a `HashSet` with `md5` and `sha256` populated.
SHA-512 optional	`hash_file(path, algorithms=("md5", "sha256"))`	Returned `HashSet` has `sha512=None`.
Directory identity (two-layer)	`hash_directory_id("vacation", "photos")`	Result matches `hash_string(hash_string("vacation").md5 + hash_string("photos").md5)`. The two-layer scheme is validated against a manual step-through.
Null hash constant	Requesting the null hash for an algorithm	Returns the hash of `b"0"` for the given algorithm — the well-known null-hash sentinel.
ID prefix — file	A file entry	Identity string starts with `y`.
ID prefix — directory	A directory entry	Identity string starts with `x`.
ID prefix — generated metadata	A generated metadata entry	Identity string starts with `z`.
HashSet uppercase	Any hash computation	All hex strings contain only `0-9A-F`, never lowercase `a-f`.

test_timestamps.py

Exercises `core/timestamps.py` (§6.5).

Test case	Input	Expected behavior
Mtime extraction	`os.stat()` of a known file	`modified.unix` is `int(st_mtime * 1000)`. `modified.iso` is a valid ISO 8601 string matching the same instant.
Atime extraction	Same file	`accessed.unix` and `accessed.iso` consistent with `st_atime`.
Creation time (Windows/macOS)	File on a platform with `st_birthtime` or `st_ctime` as creation time	`created.unix` and `created.iso` populated.
Creation time (Linux fallback)	File on Linux where `st_birthtime` is unavailable	`created` is `None` (not an error).
ISO 8601 format	Any timestamp	ISO string matches `YYYY-MM-DDTHH:MM:SS` format (with optional fractional seconds and timezone).
Unix milliseconds	A file with `st_mtime = 1700000000.123`	`unix` value is `1700000000123` (integer milliseconds, not seconds).

test_exif.py

Exercises `core/exif.py` (§6.6). Uses `mock_exiftool` fixture for most tests.

Test case	Input	Expected behavior
Successful extraction	A JPEG file with mock exiftool returning valid JSON	Returns parsed metadata dict.
Exiftool not found	`shutil.which("exiftool")` patched to return `None`	Returns `None`. Warning logged once (not per file).
Excluded file type	A `.zip` file when `.zip` is in `exiftool_exclude_extensions`	Returns `None`. Debug-level log emitted.
Exiftool error	Mock returning non-zero returncode	Returns `None`. Warning logged with stderr content.
Malformed JSON output	Mock returning invalid JSON on stdout	Returns `None`. Warning logged.
Key filtering	Mock returning JSON with keys in the exclusion list	Excluded keys absent from returned dict.
Timeout	Mock raising `subprocess.TimeoutExpired`	Returns `None`. Warning logged.

test_sidecar.py

Exercises `core/sidecar.py` (§6.7). Uses static fixture files from `tests/fixtures/sidecar_samples/`.

Test case	Input	Expected behavior

Test case	Input	Expected behavior
JSON sidecar discovery	Parent file with a matching `*_meta.json` sidecar alongside it	Sidecar discovered and classified as `JsonMetadata`.
Description sidecar	A `.description` file alongside a parent	Content read as plain text; `MetadataEntry` created with correct type and origin.
Hash sidecar	A `.md5` file alongside a parent	Content parsed as hash value.
Binary sidecar (thumbnail)	A `.jpg` file matching the thumbnail regex	Content Base64-encoded. `transforms` list includes the encoding operation.
No sidecars present	A parent file with no matching sidecar files	Returns empty list.
Malformed JSON sidecar	A `_meta.json` file containing invalid JSON	Warning logged. Sidecar skipped or content treated as raw text (field-level error, §4.5).
Multiple sidecars per parent	A parent file with three different sidecar types	All three discovered and returned as separate `MetadataEntry` objects.
MetaMergeDelete queueing	Config with `meta_merge_delete=True`; valid sidecar found	Sidecar path appended to the delete queue.
Sidecar for directory	`_directorymeta.json` inside a directory	Discovered and classified correctly.

test_entry.py

Exercises `core/entry.py` — `build_file_entry()`, `build_directory_entry()`, and `index_path()` (§6.8).

Test case	Input	Expected behavior
Single file entry	A text file in `tmp_path`	Returns `IndexEntry` with `type="file"`, populated hashes, timestamps, name, and size.
Directory entry (flat)	A directory with three files, `recursive=False`	Returns `IndexEntry` with `type="directory"` and `items` containing three child entries. No subdirectory recursion.
Directory entry (recursive)	A directory with nested subdirectories	Returns `IndexEntry` with `items` nested to match the filesystem hierarchy.
Symlink file entry	A symlink to a file	`is_link=True`. Hashes computed from name (not content).
Empty directory	A directory with no children	Returns `IndexEntry` with `items=[]`.
Item-level error handling	A file whose content raises `PermissionError` during hashing	Item skipped; warning logged. Remaining items processed normally.
Field-level error handling	A file where exiftool fails but hashing succeeds	`IndexEntry` produced with `null` metadata; hashes and timestamps populated.
ID algorithm selection	Config with `id_algorithm="sha256"`	`id` field is `"y" + sha256_hash`, not MD5.

test_serializer.py

Exercises `core/serializer.py` (§6.9).

Test case	Input	Expected behavior
Basic serialization	A fully-populated `IndexEntry`	Valid JSON output. `schema_version` is the first key. All required fields present.
Null optional fields	An `IndexEntry` with `metadata=None`	`metadata` key present in JSON with value `null`.
SHA-512 omission	An `IndexEntry` where `HashSet.sha512` is `None`	`sha512` key absent from the `hashes` object (not present as `null`).
Nested items	A directory `IndexEntry` with child entries	`items` array contains correctly serialized child objects.
Orjson fallback	When `orjson` is not importable	Falls back to `json.dumps()`; output is semantically identical.
Ensure ASCII false	Filenames with non-ASCII characters (e.g., `日本語.txt`)	Non-ASCII characters preserved in output, not escaped to `\uXXXX`.
Indent formatting	Default serialization	Output is indented (2-space) for readability.

test_rename.py

Exercises `core/rename.py` (§6.10).

Test case	Input	Expected behavior
Successful rename	A file, its computed `storage_name`, rename enabled	File renamed to `storage_name` on disk. Old path no longer exists.
Dry run	Same file, `dry_run=True`	File NOT renamed. Log message indicates what would have been renamed.
Name collision	A file whose `storage_name` already exists at the target path	Rename skipped. Warning logged. Original file untouched.
Cross-filesystem rename	(Platform-dependent) A file where `os.rename()` would fail across mount points	`shutil.move()` fallback succeeds.

test_schema.py

Exercises `models/schema.py` — the dataclass definitions and optional Pydantic models (§5).

Test case	Input	Expected behavior
IndexEntry construction	All required fields provided	Object created successfully. All fields accessible.
IndexEntry missing required field	Required field omitted	`TypeError` raised (frozen dataclass constructor enforcement).
HashSet uppercase invariant	A `HashSet` constructed with lowercase hex	Implementer decides: either the constructor normalizes to uppercase, or a validation check rejects lowercase. The test asserts whichever contract the implementation defines.
Pydantic model validation (if available)	Valid JSON parsed via `IndexEntry.model_validate_json()`	Pydantic model constructed without errors.
Pydantic model rejection (if available)	JSON with wrong types or missing required fields	Pydantic `ValidationError` raised with descriptive message.

test_config.py

Exercises `config/loader.py` and `config/types.py` (§7).

Test case	Input	Expected behavior
Default config	`load_config()` with no config file present	Returns `IndexerConfig` with all default values from §7.2.
TOML file loading	A valid TOML config file in `tmp_path`	Overridden values applied; non-overridden values retain defaults.
Invalid TOML syntax	A TOML file with syntax errors	`ConfigurationError` raised (or equivalent). Clear error message including the file path and parse error.
Unknown keys in TOML	A TOML file with keys not defined in `IndexerConfig`	Unknown keys ignored (not an error). Logged at debug level.
CLI override merging	A TOML file setting `recursive=False`, CLI override setting `recursive=True`	CLI override wins: `config.recursive == True`.
Frozen immutability	Attempting to set a field on a constructed `IndexerConfig`	`FrozenInstanceError` raised.
Sidecar pattern configuration	A TOML file adding a new sidecar type regex	New pattern appears in `config.sidecar_include_patterns`.
Exiftool exclusion extension	A TOML file adding `.xyz` to the exclusion list	`.xyz` present in `config.exiftool_exclude_extensions`.

14.3. Integration Tests

Integration tests exercise the full indexing pipeline end-to-end — from a filesystem path to validated JSON output — without mocking the core engine. The distinction from unit tests is that integration tests validate the wiring between modules (does `index_path()` correctly thread configuration through traversal, hashing, timestamps, exif, sidecar, serialization?), while unit tests validate individual module behavior in isolation.

Integration tests create real filesystem structures in `tmp_path` and call the public API or the CLI to process them. Output is captured and validated for structural correctness, field presence, and value accuracy.

test_single_file.py

Exercises the single-file indexing path: `index_path()` called on a file target.

Test case	Input	Expected behavior
Text file	A `.txt` file with known content	Output `IndexEntry` has `type="file"`, correct `name.text`, non-null `hashes.content` with verifiable digests, valid timestamps, `items` absent or `null`.
Binary file	A `.bin` file with random bytes	Hashes computed from actual content. Size matches file length.
Zero-byte file	An empty file	Hashes are the well-known empty-input values. `size.bytes == 0`.
File with sidecar	A `.jpg` file with a `_meta.json` sidecar alongside it, `extract_exif=False`, `meta_merge=True`	`metadata` array includes a sidecar-origin entry with the sidecar's content.
File with exiftool	A real JPEG file, `extract_exif=True`	`metadata` array includes a generated-origin `exiftool.json_metadata` entry. (Requires `requires_exiftool` marker.)
Symlink to file	A symlink pointing to a real file	`is_link=True`. Name hashes used for identity (not content hashes).

test_directory_flat.py

Exercises flat (non-recursive) directory indexing.

Test case	Input	Expected behavior
Directory with files	A directory containing three files	Root entry has `type="directory"`. `items` contains three file entries. No subdirectory entries in `items`.
Directory with subdirectories	A directory containing files and one subdirectory, `recursive=False`	`items` includes the subdirectory as a directory entry with its own identity and timestamps, but the subdirectory's `items` is absent (not recursed into).
Empty directory	A directory with no children	Root entry has `items=[]`.

test_directory_recursive.py

Exercises recursive directory indexing.

Test case	Input	Expected behavior
Two-level tree	A directory with files and one subdirectory containing files	Root `items` includes the subdirectory entry. That subdirectory entry's `items` includes its child files.
Deep nesting	A 5-level deep directory tree	All levels present in the nested `items` structure.
Large flat directory	A directory with 100 files	All 100 files present in `items`. Marked `@pytest.mark.slow` if execution exceeds a few seconds.
Mixed exclusions	A tree containing both included and excluded items (e.g., `desktop.ini`, `.git/`)	Excluded items absent from `items` at all levels.

test_output_modes.py

Exercises the three output routing modes (§8.3, §8.9).

Test case	Input	Expected behavior
Stdout output	`index_path()` with `output_stdout=True`	JSON written to the captured stdout stream. Valid JSON parseable by `json.loads()`.
File output	Config with `output_file` pointing to a path in `tmp_path`	JSON written to the specified file. File exists after invocation. Content is valid JSON.

Test case	Input	Expected behavior
In-place output	Config with `output_inplace=True` on a directory	`_meta2.json` sidecar files created alongside each indexed file. `_directorymeta2.json` created in each indexed directory.
Stdout suppression	`output_stdout=False`, `output_file` set	No output on stdout; output written to file only.
Combined modes	`output_stdout=True` and `output_file` set	Output appears on both stdout and in the file. Both are valid JSON.

test_cli.py

Exercises the CLI interface (§8) by invoking `click`'s test runner or via `subprocess.run()`.

Test case	Input	Expected behavior
Default invocation	`shruggie-indexer` with no arguments (CWD is `tmp_path`)	Indexes the current directory recursively. Exits with code 0. JSON on stdout.
`--help`	`shruggie-indexer --help`	Exits with code 0. Output contains usage text matching §8.1.
`--version`	`shruggie-indexer --version`	Exits with code 0. Output contains the version string from `_version.py`.
File target	`shruggie-indexer path/to/file.txt`	Indexes the single file. Output is one `IndexEntry`.
Directory target	`shruggie-indexer path/to/dir/`	Indexes the directory recursively.
`--no-recursive`	`shruggie-indexer --no-recursive path/to/dir/`	Flat traversal. Output contains only immediate children.
`--outfile`	`shruggie-indexer -o output.json path/`	Output written to `output.json`. Stdout is empty.
`--meta`	`shruggie-indexer --meta path/to/file.jpg`	Metadata extraction attempted. (Skipped if exiftool unavailable.)
`--id-type sha256`	`shruggie-indexer --id-type sha256 path/to/file.txt`	`id` field uses SHA-256, not MD5. `id_algorithm` field is `"sha256"`.
Invalid target	`shruggie-indexer /nonexistent/path`	Exits with code 3 (`TARGET_ERROR`). Error message on stderr.
Invalid flag combination	`shruggie-indexer --meta-merge-delete` (without `--outfile` or `--inplace`)	Exits with code 2 (`CONFIGURATION_ERROR`). Error message explains the safety requirement.
`-v` verbosity	`shruggie-indexer -v path/`	INFO-level log messages appear on stderr.
`-q` quiet mode	`shruggie-indexer -q path/`	No log output on stderr (except fatal errors).
Exit code 0	Successful single-file index	`result.returncode == 0`.
Exit code 1	Directory with one unreadable file	`result.returncode == 1` (partial failure). Output still produced for readable files.

14.4. Output Schema Conformance Tests

Conformance tests validate that the JSON output produced by the indexer structurally matches the canonical v2 JSON Schema definition at `schemas.shruggie.tech/data/shruggie-indexer-v2.schema.json`. These tests use the `jsonschema` package (§12.3) to perform Draft-07 validation against actual serializer output.

Conformance tests are architecturally distinct from unit and integration tests: they do not test implementation logic. They test whether the output artifact — the final JSON bytes — conforms to the published contract. A conformance failure means the serializer is producing output that an external consumer would reject as invalid.

Schema loading

The `tests/conformance/test_v2_schema.py` module loads the canonical schema once per test session. The schema SHOULD be loaded from a local copy committed to `tests/fixtures/` (for offline reproducibility) and validated periodically against the published URL to detect drift. The test module SHOULD NOT fetch the schema from the network on every test run — this makes the test suite dependent on network availability and introduces latency.

```
# Illustrative — not the exact implementation.
import json
import jsonschema
import pytest

SCHEMA_PATH = Path(__file__).parent.parent / "fixtures" / "shruggie-indexer-v2.schema.json"
```

```python
@pytest.fixture(scope="session")
def v2_schema():
    return json.loads(SCHEMA_PATH.read_text(encoding="utf-8"))

def validate_entry(entry_json: str, schema: dict) -> None:
    """Validate a JSON string against the v2 schema. Raises on failure."""
    instance = json.loads(entry_json)
    jsonschema.validate(instance=instance, schema=schema)
```

Conformance test cases

Test case	Input	Validation
Single file entry	Index a text file, serialize to JSON	`jsonschema.validate()` passes against the v2 schema.
Directory entry (flat)	Index a flat directory	Schema validation passes on the root entry and each child in `items`.
Directory entry (recursive)	Index a recursive directory tree	Schema validation passes at every level of the nested structure.
Entry with metadata	Index a file with sidecar metadata, `meta_merge=True`	Schema validation passes; `metadata` array entries conform to `MetadataEntry` definition.
Entry with exiftool metadata	Index a JPEG with `extract_exif=True` (requires exiftool)	Schema validation passes; generated `MetadataEntry` conforms.
Symlink entry	Index a symlink	Schema validation passes; `is_link=True` is valid.
All field types exercised	A purpose-built fixture that exercises every optional field and every sub-object type	Full schema coverage — every `$ref` in the schema is exercised at least once.
Schema version discriminator	Any valid entry	`entry["schema_version"] == 2`.
No additional properties	Any valid entry	Validation with `additionalProperties: false` passes — no unexpected keys at any nesting level.

Serialization invariant checks

Beyond schema validation, conformance tests verify the serialization invariants defined in §5.12:

Invariant	Test
Required fields always present	Parse the JSON output and verify that every field listed in the schema's `required` array is present as a key (even if the value is `null`).
SHA-512 omission when not computed	When `compute_sha512=False`, verify that `sha512` key does NOT appear in any `hashes` object.
Sidecar-only fields present for sidecars	For `MetadataEntry` objects with `origin="sidecar"`, verify that `file_system`, `size`, and `timestamps` are present.
Generated-only fields absent for generated entries	For `MetadataEntry` objects with `origin="generated"`, verify that `file_system`, `size`, and `timestamps` are absent.

14.5. Cross-Platform Test Matrix

The test suite runs on three platforms in the CI pipeline (§13.5.1). Most tests are platform-agnostic and run identically everywhere. Platform-specific tests are isolated in `tests/platform/` and conditionally executed via markers.

CI matrix

Platform	Runner	Python version	Test scope
Windows x64	`windows-latest`	3.12	`tests/unit/`, `tests/integration/`, `tests/conformance/`, `tests/platform/` (with `platform_windows` marker)
Linux x64	`ubuntu-latest`	3.12	`tests/unit/`, `tests/integration/`, `tests/conformance/`, `tests/platform/` (with `platform_linux` marker)
macOS x64	`macos-13`	3.12	`tests/unit/`, `tests/integration/`, `tests/conformance/`, `tests/platform/` (with `platform_macos` marker)

Platform	Runner	Python version	Test scope
macOS ARM64	`macos-latest`	3.12	Same as macOS x64. Validates ARM64 behavior.

All runners install the package with `pip install -e ".[dev,cli,gui]"` and run `pytest` with the `requires_exiftool` marker excluded (unless exiftool is pre-installed on the runner). The CI pipeline does not install exiftool — exiftool-dependent integration tests are validated during local development and optionally in a dedicated CI job that installs exiftool.

Platform-specific test targets

`tests/platform/test_timestamps_platform.py`

Test case	Platform	Expected behavior
Creation time via `st_birthtime`	macOS	`created` timestamp populated from `st_birthtime`.
Creation time via `st_ctime`	Windows	`created` timestamp populated from `st_ctime` (which is creation time on NTFS).
Creation time fallback	Linux (ext4)	`created` is `None` — `st_birthtime` not available, `st_ctime` is change time, not creation time.
Timestamp precision	All platforms	Unix millisecond value and ISO string represent the same instant within 1-second tolerance.

`tests/platform/test_symlinks_platform.py`

Test case	Platform	Expected behavior
File symlink detection	All	`is_link=True` for symlinks created with `Path.symlink_to()`.
Directory symlink detection	All	`is_link=True` for directory symlinks.
Dangling symlink handling	All	Item-level error: warning logged, item either skipped or produced with degraded fields.
Symlink hashing fallback	All	Symlinked files use name-based hashing (not content hashing).
Symlink creation (Windows)	Windows	Test uses `os.symlink()` with appropriate privileges. Skipped if symlink creation fails (non-admin user without Developer Mode).

Platform-conditional skip pattern

Platform-specific tests use `pytest.mark.skipif` with platform detection:

```
import platform
import pytest

is_windows = platform.system() == "Windows"
is_linux = platform.system() == "Linux"
is_macos = platform.system() == "Darwin"

@pytest.mark.platform_windows
@pytest.mark.skipif(not is_windows, reason="Windows-specific test")
def test_creation_time_windows(sample_file):
    ...
```

The double-marker pattern (`@pytest.mark.platform_windows` plus `@pytest.mark.skipif`) enables both marker-based filtering (`-m platform_windows`) and automatic skipping on non-matching platforms. The `skipif` ensures the test never executes on the wrong platform, while the marker enables positive selection (`pytest -m platform_windows` runs only Windows tests).

14.6. Backward Compatibility Validation

Backward compatibility tests validate that the tool produces output semantically equivalent to the original `MakeIndex` implementation for the same input paths — accounting for the documented v1-to-v2 schema restructuring (§5) and the sixteen intentional deviations (§2.6). These tests are not schema conformance tests (those validate structure); they are semantic equivalence tests (these validate that the indexer computes the correct values).

Reference data approach

The test suite includes a set of pre-computed reference entries — known-good `IndexEntry` values for specific inputs. These reference entries are created by running the original `MakeIndex` on a set of controlled input files, manually converting the v1 output to v2 field structure, and committing the result as fixture

data in `tests/fixtures/`.

Reference data files follow a naming convention: `tests/fixtures/reference/{test_name}.v2.json`. Each file contains a complete v2 `IndexEntry` with all fields populated as the indexer should produce them.

What backward compatibility validates

Validation target	How it is tested
Hash identity equivalence	For a file with known content, the tool's `id` field matches the reference. This validates that hashing (content encoding, algorithm selection, hex formatting, prefix application) produces the same identity as the original.
Name hash equivalence	For a file with a known name, the tool's `name.hashes` match the reference. This validates that string hashing (UTF-8 encoding, case handling) matches the original.
Directory identity equivalence	For a directory with a known name and parent, the tool's `id` matches the reference. This validates the two-layer `hash(hash(name) + hash(parent))` scheme.
Timestamp equivalence	For a file with known timestamps, the tool's `timestamps` values match the reference within platform precision limits (±1 second for ISO, ±1000 for Unix milliseconds).
Sidecar discovery equivalence	For a file with known sidecar files alongside it, the tool discovers and classifies the same sidecars as the original.
Sidecar content equivalence	For a JSON sidecar with known content, the tool's parsed metadata matches the reference.

Intentional deviation exclusions

The sixteen intentional deviations (DEV-01 through DEV-16, §2.6) produce expected differences from the original's output. Backward compatibility tests explicitly account for these:

Deviation	Impact on backward compatibility test
DEV-02 (multi-algorithm single-pass hashing)	Reference data includes SHA-256 (and optionally SHA-512) values. SHA-1 is dropped. Tests validate the populated values against independently computed hashes, not against the original's outputs.
DEV-07 (direct timestamp derivation)	Timestamps derived from `os.stat()` floats rather than formatted strings. Tests allow a ±1 second tolerance window for ISO timestamps and ±1000 for Unix millisecond values.
DEV-09 (computed null-hash constants)	The tool computes null hashes at module load time. Reference data reflects the computed values (`hash(b"0")`), not the original's hardcoded constants (which should be identical, but the test verifies this).

Tests that exercise unchanged behavior (hash identity, sidecar discovery, directory two-layer scheme) require exact matches. Tests that exercise deviated behavior use the deviation-specific validation rules above.

Fixture creation and maintenance

Reference fixtures are created once during the initial porting effort and committed to the repository. They are NOT regenerated on every test run. Updating reference fixtures requires:

1. Running the original `MakeIndex` on the controlled input files.
2. Converting v1 output to v2 structure using the field mapping from §5.11.
3. Applying the sixteen deviation adjustments.
4. Committing the updated fixtures with a description of what changed.

This is a manual process — automated reference generation would require running the original PowerShell implementation, which is not available in the CI environment and is not included in the repository (§1.2).

14.7. Performance Benchmarks

Performance benchmarks validate that the indexer operates within acceptable time and resource bounds for representative workloads. Benchmarks are not pass/fail tests — they produce timing data that is tracked over time to detect performance regressions. They are marked `@pytest.mark.slow` and are excluded from the default test run.

Benchmark scenarios

Scenario	Input	Measured metric	Baseline expectation

Scenario	Input	Measured metric	Baseline expectation
Single file hashing (small)	A 1 KB file, MD5 + SHA-256 (default algorithms)	Wall-clock time for `hash_file()`	< 10 ms. Hash computation for small files should be dominated by file-open overhead, not computation.
Single file hashing (large)	A 100 MB file, MD5 + SHA-256 (default algorithms)	Wall-clock time and throughput (MB/s)	Throughput within 50% of raw `hashlib` speed on the same file. Validates that the multi-algorithm single-pass approach does not introduce unexpected overhead.
Directory traversal (wide)	A directory with 10,000 files	Wall-clock time for `list_children()`	< 5 seconds. Single-pass `os.scandir()` enumeration should be I/O-bound.
Directory traversal (deep)	A 50-level deep directory chain	Wall-clock time for recursive `index_path()`	Completes without stack overflow. Python's default recursion limit (1000) is not exceeded for reasonable depths.
Full pipeline (small tree)	A directory tree with 100 files across 10 subdirectories	Wall-clock time for `index_path()` with `recursive=True`	< 5 seconds (excluding exiftool). Establishes a baseline for per-file overhead.
Full pipeline (medium tree)	A directory tree with 1,000 files across 100 subdirectories	Wall-clock time for `index_path()` with `recursive=True`	< 60 seconds (excluding exiftool). Linear scaling from the small-tree baseline.
Serialization (large output)	An `IndexEntry` tree with 1,000 entries	Wall-clock time for `serialize_entry()`	< 2 seconds with `json.dumps()`. Faster with `orjson` if available.
Exiftool invocation (batch)	10 JPEG files via `PyExifTool` batch mode	Wall-clock time per file	< 50 ms per file after process startup. Batch mode amortizes the exiftool startup cost across all files.
Exiftool invocation (fallback)	10 JPEG files via `subprocess.run()` (per-file mode)	Wall-clock time per file	< 500 ms per file. Dominated by exiftool startup time.

Benchmark implementation

Benchmarks use `time.perf_counter()` for wall-clock measurement. They are implemented as regular pytest test functions with timing logic:

```
# Illustrative — not the exact implementation.
import time
import pytest

@pytest.mark.slow
def test_benchmark_hash_large_file(tmp_path):
    """Benchmark: hash a 100 MB file with default algorithms (MD5 + SHA-256)."""
    large_file = tmp_path / "large.bin"
    large_file.write_bytes(b"\x00" * (100 * 1024 * 1024))

    start = time.perf_counter()
    result = hash_file(large_file, algorithms=("md5", "sha256"))
    elapsed = time.perf_counter() - start

    # Log the result for regression tracking
    throughput = 100 / elapsed  # MB/s
    print(f"\n  hash_file (100 MB, 2 algorithms): {elapsed:.3f}s ({throughput:.1f} MB/s)")

    # Soft assertion — failure logs a warning, does not fail the test
    assert result.md5 is not None  # Sanity check
```

Benchmarks do NOT have hard pass/fail thresholds in CI. Timing is machine-dependent and varies significantly between CI runners and local hardware. Instead, benchmarks produce timing output that is reviewed during development to detect regressions. A future enhancement (post-MVP) may integrate `pytest-benchmark` or a similar tool for structured performance tracking with statistical analysis.

Resource bounds

Beyond timing, two resource limits are validated:

Memory. The indexer processes files by streaming chunks (§6.3, §17.2) and does not load entire file contents into memory. A benchmark test that hashes a 1 GB file should not cause memory usage to spike to 1 GB. This is validated by monitoring `os.getpid()` memory via `resource.getrusage()` (Unix) or `psutil` (cross-platform, if available in the `dev` extra) before and after the operation.

Recursion depth. The recursive directory traversal must not exceed Python's default recursion limit (typically 1000 frames). A benchmark test that indexes a 100-level deep directory tree validates that the implementation does not use call-stack recursion that would fail at depth. If the implementation uses recursive function calls for tree traversal, the benchmark validates that the maximum practical depth (50–100 levels) is safe. Deeper trees are a pathological case documented in §16.5.

15. Platform Portability

This section defines the cross-platform design principles, platform-specific behaviors, and filesystem abstraction strategies that enable `shruggie-indexer` to run correctly on Windows, Linux, and macOS from a single codebase. It is the normative reference for how the indexer handles the platform differences that affect filesystem traversal, path manipulation, timestamp extraction, symlink detection, hashing, and output generation.

`shruggie-indexer` uses Python's cross-platform standard library to abstract away all OS-specific dependencies. This section documents the residual platform differences that Python's abstractions cannot fully hide — the behavioral variations in filesystem semantics that produce different observable output depending on which operating system the indexer runs on.

> **Historical note:** The original `MakeIndex` runs exclusively on Windows. It depends on PowerShell, .NET Framework types (`System.IO.Path`, `System.IO.FileAttributes`, `DateTimeOffset`), Windows system utilities (`certutil`), and Windows-specific filesystem assumptions (NTFS semantics, backslash separators, the `ReparsePoint` attribute for symlink detection, reliable creation time via `CreationTime`). All of these Windows-specific dependencies are eliminated in `shruggie-indexer`.

Design goal G2 (§2.3) states: *The tool MUST run on Windows, Linux, and macOS without platform-specific code branches in the core indexing engine.* This section specifies how that goal is achieved and where platform differences still surface in the output. §6 (Core Operations) defines what each operation does; this section defines how those operations behave differently across platforms and what the implementer must account for. §14.5 (Cross-Platform Test Matrix) defines how platform-specific behaviors are tested; this section defines what those tests are verifying.

15.1. Cross-Platform Design Principles

Five design principles govern the tool's approach to platform portability. These principles are not aspirational guidelines — they are hard constraints that the implementation MUST satisfy.

Principle P1 — No platform-conditional logic in the core engine. The `core/` and `config/` subpackages MUST NOT contain `if sys.platform == ...` or `if os.name == ...` branches. All platform variation is absorbed by Python standard library abstractions (`pathlib`, `os.stat`, `os.scandir`, `hashlib`, `subprocess`) or by the configuration system. When a platform behavior cannot be abstracted away — such as creation time availability — the code uses a uniform strategy (try/fallback) that works on all platforms without branching on the OS identity.

The rationale for this constraint is maintainability: platform-conditional branches in hot-path code are a persistent source of bugs that are only discoverable when running on the affected platform. Python's standard library already provides cross-platform abstractions for every operation the indexer performs. The implementation leverages these abstractions rather than reimplementing platform detection.

The one permitted exception is the `cli/main.py` and `gui/app.py` entry points, which MAY contain platform-conditional logic for presentation-layer concerns: console encoding setup, Windows console virtual terminal processing, macOS application bundle registration, and similar concerns that do not affect indexing behavior. These are cosmetic entry-point adjustments, not core-engine branches.

Principle P2 — Output determinism across platforms. For the same input file content, the indexer MUST produce identical values for `id`, `hashes`, `name`, `size`, `extension`, and `storage_name` regardless of the platform. These are the identity and content fields — they are derived from file bytes and name strings, which are platform-independent. If the same file is indexed on Windows and on Linux, its `id` MUST match.

Fields that are inherently platform-dependent — `timestamps`, `file_system.absolute`, `attributes.is_link` — are permitted to vary between platforms. The variation is documented and expected, not a compatibility bug. Consumers that require cross-platform comparability SHOULD use `id` (content identity) or `name.hashes` (name identity), not timestamps or absolute paths.

Principle P3 — Forward-slash normalization in output. All path strings written to the output JSON use forward-slash (`/`) separators, regardless of the host platform. This applies to `file_system.relative` and to any path components embedded in metadata entries. `file_system.absolute` retains the platform-native separator because it is a verbatim filesystem reference that consumers may use for local file access — converting it to forward slashes would make it unusable on Windows. The normalization strategy is:

Output field	Separator convention	Rationale
`file_system.relative`	Always `/`	Portable relative paths for cross-platform index consumption.
`file_system.absolute`	Platform-native (`\` on Windows, `/` on Linux/macOS)	Usable as a local filesystem reference on the originating platform.
`parent.name`	N/A (leaf name, no separators)	Single directory name component, separator-free.
`name.text`	N/A (leaf name, no separators)	Single filename component, separator-free.

The relative-path normalization is performed by `core/paths.py` using `PurePosixPath(relative_path).as_posix()` or equivalent string replacement (`str.replace(os.sep, "/")`). This is a deviation from the original, which produces Windows-native backslash paths in all output.

Principle P4 — Graceful degradation for unavailable features. When a platform does not support a feature required by a specific output field — such as creation time on older Linux kernels — the indexer populates the field with the best available approximation rather than `null`. The approximation is documented (§15.5) and the degradation is transparent: a debug-level log message is emitted on the first occurrence, and the output field is populated with the fallback value. No per-file warnings are produced for known platform limitations.

Principle P5 — Test once, verify everywhere. The test suite (§14) includes a `tests/platform/` category with tests that exercise platform-specific code paths. These tests use pytest markers (`@pytest.mark.platform_windows`, `@pytest.mark.platform_linux`, `@pytest.mark.platform_macos`) and are executed on the corresponding platform in CI. Platform tests verify that the abstractions described in this section produce correct results on each target OS. Unit tests in `tests/unit/` exercise core logic that is platform-independent and run on all platforms without markers.

15.2. Windows-Specific Considerations

Windows is the original `MakeIndex`'s native platform and the only platform it supports. `shruggie-indexer` produces equivalent output on Windows while also running correctly on Linux and macOS. This subsection documents the Windows-specific behaviors that the indexer accounts for.

Path separators and `pathlib`

Windows uses backslash (`\`) as the native directory separator. The original explicitly manages separators via the `$Sep` global variable, assigned as `[System.IO.Path]::DirectorySeparatorChar`, and used in string concatenation throughout the codebase (e.g., `$FileRenamedPath = -join("$FileParentDirectory","$Sep","$FileRenamed")`). This manual separator management is entirely eliminated — `pathlib.Path` uses the platform-correct separator for all path operations, and the `/` operator constructs paths without string concatenation:

```
# Port: pathlib handles separators automatically
target_path = parent_dir / storage_name
sidecar_path = item_path.parent / f"{item_path.name}_meta2.json"
```

No code in the implementation references `os.sep` directly for path construction. The only use of `os.sep` is in the output normalization described in Principle P3, where `file_system.relative` paths are converted from platform-native separators to forward slashes.

Long path support

Windows has two path length regimes:

Regime	Maximum length	Applies when
Legacy (Win32 API)	260 characters (`MAX_PATH`)	Default on Windows 10 before 1607; applications that do not declare long-path awareness.
Extended	32,767 characters	Windows 10 1607+ with the `LongPathsEnabled` registry key set, or paths prefixed with `\\?\`.

The original checks for the 260-character limit in `Base64DecodeString` (line 635) but does not implement general long-path handling. Python 3.6+ on Windows automatically uses the extended-length path prefix (`\\?\`) for paths exceeding 260 characters when the application manifest declares long-path awareness. CPython 3.6+ includes this manifest.

The indexer does not implement its own path-length management. It relies on Python's built-in long-path support, which is transparent to the application code. If a user encounters path-length errors on older Windows configurations where `LongPathsEnabled` is not set, the resolution is a Windows configuration change, not a code change in the indexer. A diagnostic message SHOULD be added to the error handler for `OSError` with `winerror 206` (filename or extension too long) that suggests enabling long paths.

UNC paths

Universal Naming Convention paths (`\\server\share\path`) are valid input targets on Windows. `pathlib.PureWindowsPath` handles UNC paths correctly — `Path("\\\\server\\share\\folder").parts` returns `("\\\\server\\share\\", "folder")`, and `Path.resolve()` preserves the UNC prefix.

The indexer does not special-case UNC paths. They flow through `resolve_path()` (§6.2) and `extract_components()` like any other path. The only behavioral difference is that the `parent_path` for a root-level item on a UNC share will be the share root (`\\server\share`) rather than a drive letter.

UNC paths are not applicable on Linux or macOS. CIFS/SMB mounts on those platforms appear as local paths (e.g., `/mnt/share/folder`), and the mount abstraction is transparent to the indexer.

Case preservation, case insensitivity

NTFS is case-preserving but case-insensitive by default. A file created as `Photo.JPG` retains that casing in directory listings, but `photo.jpg`, `PHOTO.JPG`, and `Photo.JPG` all resolve to the same file. This affects two operations:

Filesystem exclusion matching (§6.1). The exclusion filter uses case-insensitive comparison (`entry.name.lower() in excludes_set`) on all platforms. This is correct for Windows (where the filesystem is case-insensitive) and conservative for Linux (where it filters slightly more aggressively than the filesystem requires). The minor over-filtering on Linux — excluding a file literally named `$RECYCLE.BIN` on a case-sensitive filesystem — is harmless and simplifies the implementation.

Sorting order (§6.1). The traversal sort key (`lambda e: e.name.lower()`) produces consistent ordering across platforms. On Windows, the OS already returns entries in case-insensitive order from most NTFS directories; the explicit sort ensures the same order on case-sensitive filesystems.

Rename collision detection (§6.10). `storage_name` values are lowercase hex strings, so case-insensitivity does not affect rename collisions — the hash output is already normalized to a single case. The collision check (`os.stat()` inode comparison) works correctly on both case-sensitive and case-insensitive filesystems.

Console encoding

Windows console applications default to the OEM code page (e.g., CP437 or CP850) for stdout, which cannot represent the full Unicode range. The CLI entry point (`cli/main.py`) SHOULD set the console output encoding to UTF-8 at startup:

```
# Illustrative — not the exact implementation.
import sys, os
if sys.platform == "win32":
    sys.stdout.reconfigure(encoding="utf-8")
    sys.stderr.reconfigure(encoding="utf-8")
    # Enable ANSI escape sequences for colored output on Windows 10+
    os.system("")  # Triggers VT processing mode
```

This is one of the permitted entry-point platform branches (Principle P1 exception). It does not affect core engine behavior — it ensures that filenames containing non-ASCII characters are displayed correctly when output is piped to the console.

Windows Defender and real-time scanning

On Windows systems with real-time antivirus scanning enabled (Windows Defender, third-party AV), the file open operations during hashing may trigger per-file scanning. For large directory trees, this can significantly increase indexing time. The indexer cannot control or bypass real-time scanning. If users report unexpectedly slow performance on Windows, the documentation SHOULD mention AV scanning as a potential cause and suggest excluding the target directory from real-time scanning during indexing.

This is a documentation concern, not a code concern. The indexer does not implement AV-avoidance strategies.

15.3. Linux and macOS Considerations

Linux and macOS share POSIX filesystem semantics but differ from each other in several ways that affect the indexer.

Linux: filesystem diversity

Linux supports dozens of filesystem types. The filesystems most likely to be encountered by the indexer — and their relevant behavioral differences — are:

Filesystem	Creation time (`st_birthtime`)	Access time	Max filename length	Case sensitivity
ext4	Available on kernel 4.11+ via `statx`; Python 3.12+ exposes as `st_birthtime`	Configurable: `noatime`, `relatime` (default), `strictatime`	255 bytes	Case-sensitive
XFS	Available on kernel 4.11+ via `statx`	Configurable	255 bytes	Case-sensitive
Btrfs	Available on kernel 4.11+ via `statx`	Configurable	255 bytes	Case-sensitive
tmpfs	Not available	Configurable	255 bytes	Case-sensitive
NFS	Depends on server filesystem	Depends on server	Depends on server	Depends on server
FAT32/exFAT	Not available	Not reliable	255 characters (LFN)	Case-insensitive

The indexer does not detect or adapt to the underlying filesystem type. It uses the uniform `os.stat()` / `os.lstat()` interface and relies on the kernel to provide whatever timestamp precision and attribute support the filesystem offers. When `st_birthtime` is unavailable, the `st_ctime` fallback (§15.5) activates transparently.

Access time caveat. The `relatime` mount option, which is the default on most Linux distributions since around 2009, only updates `atime` when the previous access time is older than the modification time. This means `accessed` timestamps in the index output may not reflect the most recent access. The indexer reports whatever `os.stat()` provides without attempting to validate accuracy. This is documented in §5.7 and is not a portability bug.

macOS: HFS+ and APFS

macOS uses APFS (on SSDs since macOS 10.13) or HFS+ (on HDDs and older systems). Both filesystems are case-preserving and, by default, case-insensitive — the same behavior as NTFS on Windows.

Filesystem	Creation time	Max filename	Case sensitivity	Unicode normalization
APFS	`st_birthtime` always available	255 UTF-8 bytes	Case-insensitive (default)	Preserves original form (NFD or NFC)
HFS+	`st_birthtime` always available	255 UTF-16 code units	Case-insensitive (default)	Normalizes to NFD on storage

Unicode normalization (HFS+). HFS+ normalizes filenames to Unicode NFD (decomposed form) on storage. A file created as `café.txt` (NFC, single `é` code point U+00E9) is stored as `café.txt` (NFD, `e` + combining acute accent U+0301). This normalization is visible to `os.scandir()` — the returned `DirEntry.name` reflects the filesystem's stored form, not the form originally used at creation.

APFS does not normalize — it preserves whatever form was used at creation. Most macOS tools use NFC, so APFS filenames are typically NFC.

This normalization difference affects name hashing: `hash_string("café")` would produce different results depending on whether the name is NFC or NFD, because the UTF-8 byte sequences differ. To ensure cross-platform hash determinism, `hash_string()` applies `unicodedata.normalize('NFC', value)` unconditionally on all platforms before encoding to UTF-8 and hashing (DEV-15). This guarantees that a file named `café.txt` produces identical identity hashes regardless of whether the filesystem returned the name in NFC (Windows, typical APFS) or NFD (HFS+). The NFC normalization is applied on all platforms — not just macOS — because APFS preserves whatever form was used at creation, meaning NFD filenames can appear on any macOS volume.

> **Note:** This is a deliberate break from the "hash what the filesystem returns" approach. The original never needed to address cross-platform filename equivalence because it ran exclusively on Windows (which always uses NFC). The tool prioritizes cross-platform hash determinism: the same logical filename MUST produce the same identity on all supported platforms. The tradeoff is that re-indexing the same file on an HFS+ volume will produce an identity hash based on the NFC normalization of the filename rather than the NFD form stored on disk — but this is the correct behavior, because the hashed value represents the *logical* filename, not its filesystem-specific encoding.

macOS: extended attributes and quarantine

macOS assigns extended attributes (`com.apple.quarantine`, `com.apple.metadata:*`) to downloaded files and files with Finder tags. These attributes are not part of the standard `os.stat()` result and are not read by the indexer. They do not affect any indexed fields.

The `.DS_Store` file, which macOS Finder creates in every directory it visits, is excluded by the default filesystem exclusion set (§6.1, §7.2). Similarly, `.Spotlight-V100`, `.Trashes`, `.fseventsd`, `.TemporaryItems`, and `.DocumentRevisions-V100` are excluded by default. These are all macOS system artifacts that should not appear in index output.

macOS and Linux: file permissions

The indexer requires read permission on every file it hashes and every directory it traverses. On Linux and macOS, files may have restrictive permissions that prevent the indexer (running as the current user) from reading them. The error handling strategy (§4.5) applies: a `PermissionError` during `hash_file()` or `os.scandir()` is treated as an item-level error — the item is included in the output with degraded fields (`null` hashes, empty `items` list for directories), and a warning is logged.

On Windows, file permissions are managed through ACLs rather than POSIX mode bits, but the behavioral outcome is the same: a `PermissionError` is raised by the OS if the current user lacks read access, and the same error handling applies.

15.4. Filesystem Behavior Differences

This subsection consolidates the filesystem behaviors that vary across platforms and affect indexer output. Each behavior is described in terms of its observable effect on the output schema fields, not in terms of internal implementation details (which are covered in §6).

Path separator in `file_system.absolute`

Platform	Example `file_system.absolute` value
Windows	`"C:\\Users\\alice\\photos\\sunset.jpg"`
Linux	`"/home/alice/photos/sunset.jpg"`
macOS	`"/Users/alice/photos/sunset.jpg"`

The `file_system.relative` field always uses forward slashes, regardless of platform (Principle P3). The `file_system.absolute` field uses the platform-native separator because it serves as a local reference.

File size consistency

`os.stat().st_size` returns the logical file size in bytes on all platforms. This value is consistent across platforms for the same file content — a 1,024-byte file reports `st_size = 1024` on Windows, Linux, and macOS. The `size.bytes` output field is platform-independent.

Sparse files are a minor exception: `st_size` reports the logical size (including sparse regions), not the physical disk allocation. The indexer reports the logical size, which is the same across platforms. Content hashing of sparse files reads the logical content (including zero-filled sparse regions), so hash values are also consistent.

Filename length limits

Platform/FS	Max filename length	Max path length
NTFS	255 UTF-16 code units	32,767 characters (extended)
ext4	255 bytes (UTF-8)	No fixed limit (kernel `PATH_MAX` = 4,096 bytes)
APFS	255 UTF-8 bytes	No fixed limit
HFS+	255 UTF-16 code units	No fixed limit

The indexer does not validate filename lengths. If the filesystem accepts a filename, the indexer can process it. Filenames that exceed the target filesystem's limit when writing sidecar files (the original name plus `_meta2.json` suffix, or `_directorymeta2.json`) will produce an `OSError` that is handled as a file-level error (§4.5).

Hidden files and dot-files

On Linux and macOS, files and directories whose names begin with a dot (`.`) are conventionally hidden. On Windows, hidden status is an NTFS attribute (`FILE_ATTRIBUTE_HIDDEN`) independent of the filename.

The indexer does not distinguish between hidden and visible files. All files and directories within the target path are indexed, including dot-files on Linux/macOS and hidden files on Windows. The filesystem exclusion filters (§6.1) handle specific system artifacts (`.DS_Store`, `Thumbs.db`, etc.) by name, not by hidden status.

The `Get-ChildItem -Force` flag used in the original includes hidden files; `os.scandir()` includes all entries by default. The behavior is equivalent.

Atomic rename guarantees

`Path.rename()` delegates to the platform's rename system call:

Platform	System call	Atomicity	Cross-device support
Windows	`MoveFileExW`	Atomic on same volume (NTFS)	Fails with `OSError` if source and target are on different volumes.
Linux	`rename(2)`	Atomic on same filesystem	Fails with `EXDEV` if source and target are on different filesystems.
macOS	`rename(2)`	Atomic on same filesystem	Fails with `EXDEV` if source and target are on different filesystems.

The rename operation in §6.10 always targets the same directory as the source file (`item_path.parent / storage_name`), so the source and target are guaranteed to be on the same filesystem. Cross-device rename failures are not expected in normal operation. The `shutil.move()` fallback exists as a safety net but should never be reached during standard rename operations.

On Windows, `Path.rename()` will fail if the target file already exists and is a different file (unlike POSIX, where `rename(2)` atomically replaces the target). The collision detection in §6.10 — which checks for target existence before calling `rename()` — handles this Windows-specific behavior correctly: if the target exists and is a different inode, a `RenameError` is raised; if it is the same inode (already renamed), the operation is a no-op.

> **Historical note:** The original uses PowerShell's `Move-Item`, which delegates to `MoveFileExW` on Windows. `Path.rename()` uses the same underlying system call on Windows and the POSIX `rename(2)` on Linux/macOS. The behavioral difference — Windows `rename` failing on existing targets vs. POSIX `rename` atomically replacing them — is handled by the collision detection layer, making the observable behavior identical across platforms.

15.5. Creation Time Portability

Creation time is the single most significant cross-platform behavioral difference that surfaces in the indexer's output. The original relies on .NET's `CreationTime` property, which is always available on Windows. The indexer must handle platforms where true creation time may not be available.

Platform availability matrix

Platform	Python attribute	Source	Reliability
Windows (NTFS)	`st_birthtime` (Python 3.12+)	NTFS `$STANDARD_INFORMATION.CreationTime`	Always available. This is the true file creation time.
Windows (NTFS)	`st_ctime`	Same as `st_birthtime` on Windows	Always available. On Windows, Python maps `st_ctime` to the NTFS creation time, not the inode change time.
macOS (APFS/HFS+)	`st_birthtime`	Filesystem creation timestamp	Always available. macOS has supported `st_birthtime` in its `stat` structure since OS X 10.6.

Platform	Python attribute	Source	Reliability
Linux (ext4/XFS/Btrfs, kernel 4.11+)	`st_birthtime` (Python 3.12+)	`statx` system call with `STATX_BTIME`	Available when the kernel supports `statx` and the filesystem records birth time. Python 3.12 added `st_birthtime` on Linux via `statx`.
Linux (older kernels or tmpfs/NFS)	Not available	—	`st_birthtime` raises `AttributeError`. Fallback to `st_ctime` is required.

Resolution strategy

The implementation uses a two-tier approach defined in §6.5:

```
# Illustrative — not the exact implementation.
def _get_creation_time(stat_result: os.stat_result) -> float:
    try:
        return stat_result.st_birthtime
    except AttributeError:
        return stat_result.st_ctime
```

This pattern is platform-independent — it does not check `sys.platform`. It simply attempts `st_birthtime` and falls back to `st_ctime` if the attribute does not exist. The fallback is silent at the per-file level; a debug-level log message is emitted on the first fallback occurrence per invocation to inform the user that creation times are approximate.

Semantic difference of the fallback

When `st_ctime` is used as the creation time fallback, the value's meaning differs by platform:

Platform	`st_ctime` meaning	Relationship to creation time
Windows	NTFS creation time	Identical — `st_ctime` IS the creation time on Windows. This is not a fallback in practice; it is a second path to the same value.
Linux	Inode change time (ctime)	Different — ctime updates when file metadata changes (permissions, ownership, hard link count), not when the file is created. For files that have never had metadata changes, `ctime` ≈ `mtime` ≈ creation time. For files that have been `chmod`ed or `chown`ed, `ctime` may be more recent than the actual creation time.
macOS	Inode change time (ctime)	Same as Linux, but the fallback is rarely reached because `st_birthtime` is available on all macOS filesystems.

The practical impact is limited: most files processed by the indexer are media files and documents that are created once and rarely have their metadata changed. For these files, `st_ctime` on Linux is a good approximation of creation time. The deviation is most visible for files that have undergone `chmod`, `chown`, or hard link operations — these will show a `created` timestamp that reflects the most recent metadata change, not the original creation.

Output implications

The `timestamps.created` field in the v2 schema (§5.7) always contains a value — it is never `null` due to platform limitations. The value is either the true creation time (from `st_birthtime`) or the best available approximation (from `st_ctime`). The output schema does not include a field indicating which source was used, because distinguishing the two would require consumers to handle the ambiguity and provides limited actionable value.

If a future version requires explicit provenance tracking for creation times, a `timestamps.created_source` field (with values like `"birthtime"` or `"ctime_fallback"`) could be added to the schema without breaking backward compatibility.

Interaction with backward compatibility testing

Backward compatibility tests (§14.6) that validate timestamp equivalence between the tool and the original MUST account for the creation-time difference. The original always uses .NET `CreationTime` (true creation time on Windows NTFS). When the tool runs on Linux without `st_birthtime` support, the `created` timestamp may differ from the reference value. The test tolerance (±1 second for ISO strings, ±1000 for Unix milliseconds) applies to the numerical precision of the timestamp, not to the semantic difference between creation time and ctime.

Platform-specific tests in `tests/platform/` SHOULD include a test that verifies:

1. On Windows: `timestamps.created` matches the NTFS creation time.

2. On macOS: `timestamps.created` uses `st_birthtime`.
3. On Linux (kernel 4.11+, ext4): `timestamps.created` uses `st_birthtime` when available.
4. On Linux (fallback): `timestamps.created` uses `st_ctime` and a debug log message is emitted on the first occurrence.

15.6. Symlink and Reparse Point Handling

Symlink semantics differ significantly between Windows and POSIX systems. The original detects symlinks by checking the `ReparsePoint` attribute in the NTFS file attribute bitmask — a Windows-specific mechanism. The indexer uses `Path.is_symlink()`, which delegates to the appropriate platform mechanism. This subsection documents the platform-specific behaviors that affect symlink handling.

Platform mechanisms

Platform	Creation mechanism	Detection API	Privilege required
Windows	`mklink` (cmd), `New-Item -ItemType SymbolicLink` (PowerShell), `os.symlink()` (Python)	`Path.is_symlink()` → checks `FILE_ATTRIBUTE_REPARSE_POINT` via `GetFileAttributesW`	Creating symlinks requires either administrator privileges or Developer Mode enabled (Windows 10 1703+). Reading/detecting symlinks requires no special privileges.
Linux	`ln -s`, `os.symlink()`	`Path.is_symlink()` → `lstat()` checks `S_IFLNK` in `st_mode`	No special privileges for creation or detection.
macOS	`ln -s`, `os.symlink()`	`Path.is_symlink()` → `lstat()` checks `S_IFLNK` in `st_mode`	No special privileges for creation or detection.

The `Path.is_symlink()` call is the correct cross-platform abstraction. It returns `True` for symbolic links on all platforms and is the only symlink detection mechanism used by the indexer.

Windows: junctions vs. symlinks

Windows has two types of reparse points that behave like symlinks:

Type	Target	Created by	Detected by `is_symlink()`
Symbolic link (symlink)	File or directory, absolute or relative	`mklink /D` (directory), `mklink` (file), `os.symlink()`	Yes — `Path.is_symlink()` returns `True`.
Junction (mount point)	Directory only, absolute path only	`mklink /J`, `os.path.join()` with junction semantics	Yes — `Path.is_symlink()` returns `True` on Python 3.12+.

Both junction points and symbolic links have the `FILE_ATTRIBUTE_REPARSE_POINT` attribute set, and Python's `Path.is_symlink()` detects both as of Python 3.12. The original's `Attributes -band ReparsePoint` check also detects both types. The indexer's behavior matches the original for both junctions and symlinks.

The indexer does not distinguish between junctions and symlinks in the output — both set `attributes.is_link = true` and trigger the same behavioral changes (name hashing instead of content hashing, skipped EXIF extraction, `os.lstat()` for timestamps). This is correct behavior: both types redirect to a different filesystem location, and hashing the link target's content would produce an identity that depends on the target rather than the link itself.

Symlink targets across filesystems

A symlink can point to a target on a different filesystem or a different volume. The original does not explicitly handle this case — it simply switches to name hashing when the `ReparsePoint` attribute is detected, regardless of the target's location.

The indexer preserves this behavior. When `is_symlink()` returns `True`, content hashing is replaced by name hashing, and the link target is not followed for any content-dependent operation (hashing, EXIF extraction). The only operation that follows the link is `Path.resolve()`, which resolves the symlink to its target for the `file_system.absolute` field. If the symlink target does not exist (dangling symlink), `Path.resolve(strict=False)` returns the normalized path without verifying existence, and the entry is populated with degraded fields as documented in §6.4.

Symlink traversal safety

The indexer does not follow symlinks during directory traversal. The `list_children()` function (§6.1) uses `os.scandir()` with `follow_symlinks=False` for entry classification. A symlink to a directory appears in the traversal results but is not descended into — it is processed as a single entry with `is_link = True` and an empty `items` list. This prevents symlink loops (where directory A symlinks to B and B symlinks to A) from causing infinite recursion.

This is consistent with the original's behavior. The original's `Get-ChildItem -Force` does not follow symlinks into directories by default — it lists the symlink as an entry without descending into it.

> **Historical note:** The original does not explicitly document or test its symlink traversal behavior — the non-descent is a side effect of `Get-ChildItem`'s default behavior rather than a deliberate design decision. The explicit `follow_symlinks=False` parameter makes the safety behavior intentional and testable.

Symlink metadata sources

When processing a symlink, the timestamps module (§6.5) uses `os.lstat()` instead of `os.stat()`. The distinction:

Call	Returns metadata for
`os.stat(path)`	The symlink's target. If the target is another symlink, follows the chain to the final target. Raises `FileNotFoundError` for dangling symlinks.
`os.lstat(path)`	The symlink itself. Never follows the link. Always succeeds for an existing symlink, even if the target is missing.

On all three platforms, `os.lstat()` returns the symlink's own modification time, access time, and (where available) creation time. These timestamps reflect when the symlink was created or modified, not when the target was created or modified.

The `size` field for symlinks varies by platform:

Platform	`os.lstat().st_size` for a symlink
Windows	0 (reparse points report zero size)
Linux	Length of the target path string in bytes
macOS	Length of the target path string in bytes

This difference is inherent to the platform's symlink implementation and is documented here for completeness. Consumers SHOULD NOT rely on the `size` field for symlinks — it does not represent the size of the target file.

Creating test symlinks across platforms

The platform-specific tests in `tests/platform/` need to create symlinks for validation. On Windows, symlink creation requires either administrator privileges or Developer Mode. The test infrastructure uses `os.symlink()` with a try/except for `OSError` — if symlink creation fails due to insufficient privileges, the test is skipped with `pytest.skip("Symlink creation requires elevated privileges on Windows")`. This is handled by the test fixture, not by repeating the privilege check in every test function.

On Linux and macOS, `os.symlink()` works without special privileges. Symlink creation for tests is straightforward and does not require any special handling.

16. Security and Safety

This section defines the security boundaries and safety mechanisms that protect the filesystem, the user's data, and the indexer's own output integrity during operation. The indexer is not a network-facing service and does not process untrusted input in the traditional web-security sense — its primary threat model involves unintended filesystem mutation, resource exhaustion, and data loss through operational error. The safeguards described here are engineering guardrails, not defense-in-depth against a hostile actor.

The indexer performs five categories of operation that carry safety implications: traversal across symlink boundaries (§16.1), path resolution from user-supplied and filesystem-derived strings (§16.2), temporary file creation during output writes (§16.3), destructive deletion of sidecar files during MetaMergeDelete (§16.4), and resource consumption when processing large files or deeply nested directory trees (§16.5). Each subsection specifies the threat, the mitigation, and the relationship to the behavioral contracts defined in earlier sections.

§6 (Core Operations) defines what each operation does. §4.5 (Error Handling Strategy) defines the per-item isolation model. §15 (Platform Portability) defines how platform-specific filesystem behaviors affect safety-relevant operations. This section consolidates the safety-specific aspects of those behaviors into a single normative reference for implementers and auditors.

16.1. Symlink Traversal Safety

Threat

Symbolic links introduce three categories of safety risk during filesystem traversal:

Infinite recursion. A directory symlink that targets an ancestor of itself creates a cycle. If the traversal follows the symlink, it enters an unbounded loop: `A/ → B/ → symlink-to-A/ → B/ → symlink-to-A/ → ...`. On the original Windows-only platform, this risk was limited to NTFS junctions and developer-mode symlinks. On Linux and macOS, symlinks are ubiquitous and cycles are trivially constructible.

Scope escape. A symlink inside the target directory may point to a location outside the intended indexing scope — a parent directory, a different filesystem, a sensitive system directory, or a network mount. Following the symlink would cause the indexer to read (and hash) file content that the user did not intend to include in the index.

Dangling references. A symlink whose target has been deleted or moved causes `stat()`, `open()`, and `resolve()` to fail with `FileNotFoundError` or `OSError`. If the traversal does not anticipate this, the failure may propagate beyond the item-level error boundary and abort the entire operation.

Mitigation: non-following traversal

The indexer does not follow symlinks during directory traversal. This is the single, comprehensive mitigation for all three risks.

`list_children()` (§6.1) enumerates directory contents using `os.scandir()` with `follow_symlinks=False` for entry classification. A symlink to a directory appears in the traversal results as a single item — it is processed as an `IndexEntry` with `attributes.is_link = True` and an empty `items` list. The traversal does not descend into the symlink target. A symlink to a file likewise appears as a single item with `is_link = True` and is processed using name hashing rather than content hashing (§6.3), since reading the target's content would follow the link.

This behavior is consistent with the original's `Get-ChildItem -Force` semantics, which do not follow symlinks into directories by default. The difference is intentionality: the original's non-following behavior is an emergent side effect of PowerShell's default `Get-ChildItem` behavior, while the explicit `follow_symlinks=False` parameter is a deliberate, testable design decision (§15.6).

Symlink processing boundaries

When a symlink is encountered during traversal, the following operations are affected:

Operation	Symlink behavior	Rationale
Content hashing (`hash_file`)	Replaced by name hashing (`hash_string`)	Reading the target's content would follow the link. Name hashing produces an identity that depends on the link, not its target.
EXIF extraction (`extract_exif`)	Skipped — returns `None`	`exiftool` would follow the link to read the target file. Metadata from the target does not belong to the link.
Timestamp extraction	Uses `os.lstat()` instead of `os.stat()`	`lstat()` reads the symlink's own metadata without following the link. `stat()` would return the target's metadata (or raise `FileNotFoundError` for dangling links).
Path resolution (`resolve_path`)	Uses `Path.resolve(strict=False)` for the `file_system.absolute` field	`strict=False` normalizes the path without requiring the target to exist, handling dangling symlinks gracefully.
Directory descent	Not performed	The symlink entry has an empty `items` list regardless of whether the target is a directory.

Dangling symlink handling

When `is_symlink()` returns `True` and the symlink target does not exist, the item is processed with degraded fields. The specific degradation:

- `hashes`: Populated from name hashing (which does not require the target to exist). Not `null`.
- `id`: Derived from the name hash. Valid and deterministic.
- `size.bytes`: The value from `os.lstat().st_size` — the size of the symlink itself (0 on Windows, target-path-length on POSIX). Not the size of the (nonexistent) target.
- `timestamps`: From `os.lstat()`. Reflects the symlink's own filesystem metadata.
- `metadata`: Empty list — no EXIF extraction, no sidecar discovery (the item is a symlink).

A debug-level log message is emitted noting the dangling symlink. The entry is included in the output with no `null` fields — dangling symlinks are a normal condition, not an error.

Testing symlink safety

The test suite (§14) includes dedicated symlink safety tests in `tests/platform/`:

- **Cycle detection.** Create `A/ → B/ → link-to-A`. Index `A/` recursively. Verify that the traversal terminates, that the symlink entry has an empty `items` list, and that no infinite recursion occurs.
- **Scope boundary.** Create a symlink inside the target directory that points outside it. Verify that the external target is not hashed or traversed.
- **Dangling symlink.** Create a symlink whose target does not exist. Verify that the entry is populated with name hashes and `lstat()` timestamps, and that no exception propagates.

These tests use pytest markers for platform-conditional execution (§14.5) and handle the Windows symlink privilege requirement documented in §15.6.

16.2. Path Validation and Sanitization

Threat

The indexer processes paths from two sources: user-supplied target paths (from CLI arguments, GUI input, or API parameters) and filesystem-derived paths (from `os.scandir()` during traversal). Both sources can produce paths that are malformed, adversarial, or problematic for specific platforms.

Specific risks include path traversal components (`..`) that escape the intended scope, embedded null bytes that truncate strings in C-level filesystem calls, excessively long paths that exceed platform limits, and filenames containing characters that are legal on the source filesystem but illegal on the output filesystem (relevant when indexes are consumed cross-platform).

Target path validation (Stage 2)

The target path supplied by the user undergoes validation during Stage 2 of the processing pipeline (§4.1). The validation sequence:

1. **Resolution.** `resolve_path()` (§6.2) calls `Path.resolve(strict=True)`, which resolves symlinks, collapses `.` and `..` components, and produces a canonical absolute path. The `..` components are fully resolved by the OS before the indexer sees the path — there is no string-level `..` manipulation that an adversarial input could exploit.

2. **Existence check.** After resolution, `resolved.exists()` verifies that the path refers to an actual filesystem object. If the path does not exist, a `TargetError` is raised and the process exits with code 3 (§8.10). No traversal or file reading occurs.

3. **Type classification.** `resolved.is_file()` and `resolved.is_dir()` classify the target. If the target is neither a file nor a directory (e.g., a character device, a named pipe, or a socket), an `IndexerError` is raised. The indexer processes only regular files, directories, and symlinks to those types.

4. **Null byte rejection.** Python 3 raises `ValueError: embedded null character` when a `Path` object is constructed from a string containing `\x00`. This rejection happens before any filesystem call, preventing null-byte injection into OS-level path APIs. The implementation does not need to add explicit null-byte checking — Python's `pathlib` and `os` modules enforce this invariant automatically.

Filesystem-derived path handling

Paths returned by `os.scandir()` during traversal are filesystem-authoritative — they reflect the actual names stored by the filesystem and do not contain traversal components. No sanitization is applied to these paths because they originate from the filesystem kernel, not from user input. The entry builder (§6.8) processes them through the same `extract_components()` function (§6.2), which uses `pathlib` properties rather than string splitting, avoiding injection risks from filenames containing path separator characters.

Output path sanitization

The indexer writes output files to two categories of destination, each with its own path construction rules:

Aggregate output file (`--outfile`). The path is user-specified and resolved via `Path.resolve()` before writing. The configuration validation (§7.1, rule 3) rejects output paths that fall inside the target directory when `--inplace` is also active, preventing the output file from being indexed on subsequent runs. Beyond this conflict check, no further sanitization is applied — the user controls the output path.

In-place sidecar files (`--inplace`). Sidecar paths are constructed by `paths.build_sidecar_path()` (§6.2), which appends a fixed suffix (`_meta2.json` or `_directorymeta2.json`) to the item's existing path. The construction uses `pathlib` path arithmetic (`item_path.parent / sidecar_name`), not string concatenation. The sidecar filename is derived deterministically from the item filename, so there is no opportunity for path injection — a filename like `../../etc/passwd` on the filesystem would produce a sidecar named `../../etc/passwd_meta2.json` in the same directory (not a traversal to `/etc/`), because `pathlib`'s `/` operator joins relative to the parent, not relative to the working directory.

Rename path safety

The `rename_item()` function (§6.10) constructs the target path via `paths.build_storage_path()`, which joins the item's parent directory with its `storage_name`. Since `storage_name` is a hex string derived from a cryptographic hash (`y` + MD5/SHA256 hexdigest + optional extension), it cannot contain path separators, traversal components, or special characters. The only characters in a `storage_name` are `[a-zA-Z0-9.]`, making it safe for all target filesystems.

Collision detection (§6.10) verifies that the rename target does not already exist as a different file before executing the rename. This prevents accidental data loss from hash collisions (astronomically unlikely but guarded against at negligible cost) and from repeated runs where some files have already been renamed.

16.3. Temporary File Handling

Original approach and its problems

The original `MakeIndex` creates temporary files via `TempOpen` and deletes them via `TempClose` for a single purpose: passing exiftool arguments through an intermediary argfile. The `TempOpen` function writes to a fixed directory (`$D_PSLIB_TEMP = C:\bin\pslib\temp`) using a UUID-based naming scheme, and `TempClose` deletes the file by path. The `TempOpen` docstring warns that failing to call `TempClose` will leave orphaned temp files, and `TempClose -ForceAll` exists as a bulk cleanup mechanism — an acknowledgment that the manual open/close protocol is leak-prone.

This pattern has three safety problems. First, the fixed temp directory path (`C:\bin\pslib\temp`) is outside the standard OS temp location and requires manual creation — if the directory does not exist, `TempOpen` fails. Second, the manual open-then-close protocol has no automatic cleanup guarantee: an exception between `TempOpen` and `TempClose` leaks a temp file. Third, the entire pattern is unnecessary — subprocess argument passing does not require an intermediary file.

Port approach: elimination

The implementation eliminates the original's temporary file creation pattern for the exiftool use case. The primary backend uses `PyExifTool`'s `-stay_open` batch mode (§6.6, DEV-05, DEV-16), which communicates via stdin/stdout pipes — no temporary files involved. The subprocess fallback path uses `tempfile`-managed argfiles with automatic cleanup via context managers. In both cases, the original's `-@` argfile switch with manual `TempOpen`/`TempClose` lifecycle and Base64 encoding/decoding pipeline are eliminated.

Remaining temporary file scenarios

Two scenarios in the implementation may still involve temporary file creation:

Atomic file writes. When writing the aggregate output file (`--outfile`), the serializer SHOULD use an atomic write pattern: write to a temporary file in the same directory as the target, then rename the temporary file to the final path. This prevents partial writes from producing a corrupt output file if the process is interrupted during serialization. The implementation uses Python's `tempfile.NamedTemporaryFile(dir=target_dir, delete=False)` to create the temporary file in the correct directory (ensuring the rename is atomic on the same filesystem), writes the complete JSON content, calls `os.replace()` to atomically swap the file into place, and cleans up the temporary file in a `finally` block if the rename fails.

The `tempfile` module creates files with restrictive permissions (mode `0o600` on POSIX) and uses OS-provided mechanisms for unique naming, avoiding the collision and permission issues of the original's manual UUID scheme.

Exiftool batch mode (primary). The `PyExifTool` batch mode (§6.6, DEV-16) maintains a persistent exiftool process communicating via stdin/stdout pipes rather than argfiles. No temporary files are involved — the batch protocol is entirely in-memory. If the batch backend is unavailable, the subprocess fallback uses `tempfile`-based argfiles (see §12.2).

Cleanup guarantee

For any scenario where temporary files are created, the implementation uses context managers or `try`/`finally` blocks to guarantee cleanup:

```
# Illustrative — not the exact implementation.
import tempfile, os

def atomic_write(target_path: Path, content: str) -> None:
    target_dir = target_path.parent
    fd = None
    tmp_path = None
    try:
        fd = tempfile.NamedTemporaryFile(
            mode="w",
            encoding="utf-8",
            dir=target_dir,
            suffix=".tmp",
            prefix=".shruggie-indexer-",
            delete=False,
        )
        tmp_path = Path(fd.name)
        fd.write(content)
        fd.flush()
        os.fsync(fd.fileno())
        fd.close()
        fd = None
        os.replace(tmp_path, target_path)
        tmp_path = None  # rename succeeded — nothing to clean up
    finally:
        if fd is not None:
            fd.close()
        if tmp_path is not None and tmp_path.exists():
            tmp_path.unlink(missing_ok=True)
```

The `finally` block ensures that the temporary file is removed even if serialization fails, the process receives a signal, or an unexpected exception occurs. The `missing_ok=True` parameter to `unlink()` prevents a secondary exception if the file was already removed.

> **Historical note:** The original's `TempOpen`/`TempClose` protocol relies on manual discipline to avoid leaks. The context-manager approach makes cleanup automatic and exception-safe, consistent with Python's resource management conventions.

16.4. Metadata Merge-Delete Safeguards

Threat

MetaMergeDelete is the only destructive data operation in the indexer. When active, it deletes sidecar metadata files (`.info.json`, `.description`, thumbnails, subtitles, etc.) from the filesystem after their content has been merged into the parent item's `metadata` array. If the merged content is not persisted to a durable output — because no output file was specified, because the output file write failed, or because the process was interrupted before the output was committed — the sidecar data is irrecoverably lost. The original labels this risk explicitly: the `$MMDSafe` variable gates the entire operation on whether a persistent output mechanism is active.

Safeguard 1: Configuration-time output requirement

The configuration loader (§7.1, validation rule 1) enforces a hard prerequisite: if `meta_merge_delete` is `True`, at least one of `output_file` or `output_inplace` MUST also be `True`. This validation runs during Stage 1 — before any file is read, hashed, or modified. If the prerequisite is not met, the process terminates with exit code 2 (`CONFIGURATION_ERROR`) and a diagnostic message:

```
Error: --meta-merge-delete requires --outfile or --inplace to ensure
sidecar content is preserved before deletion.
```

This prevents the most straightforward data-loss scenario: a user running `shruggie-indexer --meta-merge-delete --stdout` (stdout-only output, no persistent file), where the sidecar content would exist only in the transient stdout stream.

The original enforces this via the `$MMDSafe` variable (lines ~9354–9427 in `MakeIndex`). The tool's enforcement is equivalent but implemented as a declarative validation rule rather than an imperative flag check embedded in the output-routing logic.

Safeguard 2: Deferred deletion

Sidecar file deletion is deferred to Stage 6 of the processing pipeline (§4.1, §4.4) — after all indexing is complete and all output has been written. During Stage 3–4 traversal, the sidecar module (§6.7) appends each successfully merged sidecar's path to the delete queue (a `list[Path]` owned by the top-level orchestrator). The actual `Path.unlink()` calls happen only after the traversal loop exits and the serializer has finished writing all output.

This temporal separation provides an interruption safety window: if the process is killed during traversal (Ctrl+C, `SIGTERM`, system crash), no sidecar files have been deleted yet. The partially-written in-place sidecar files may be incomplete, but the original sidecar source files remain intact on disk. The user can re-run the indexer to produce a complete index without data loss.

The deferral order is:

1. All items traversed, all `IndexEntry` objects constructed, all in-place sidecar files written.
2. Aggregate output file written (if `--outfile` is active).
3. Delete queue drained: each sidecar path is unlinked.

Step 3 only executes if steps 1 and 2 complete without fatal error. If an unrecoverable error occurs during traversal or output writing, the delete queue is not drained and the sidecar files remain.

Safeguard 3: Per-file deletion error isolation

When draining the delete queue, each `Path.unlink()` call is wrapped in a `try`/`except`. If a single sidecar file cannot be deleted (permission denied, file already removed, filesystem error), the failure is logged as a warning and the queue continues with the next entry. A deletion failure for one sidecar does not prevent deletion of the remaining sidecars, and does not change the exit code from `SUCCESS` to a failure code — the indexing itself completed successfully; the deletion failure is a post-processing anomaly.

```python
# Illustrative — not the exact implementation.
def drain_delete_queue(queue: list[Path]) -> int:
    failed = 0
    for sidecar_path in queue:
        try:
            sidecar_path.unlink()
            logger.debug("Deleted sidecar: %s", sidecar_path)
        except OSError as exc:
            logger.warning("Failed to delete sidecar %s: %s", sidecar_path, exc)
            failed += 1
    return failed
```

If any deletions fail, the count is included in the final status log line so the user is aware that cleanup was incomplete.

Safeguard 4: v2 schema provenance for reversal

The v2 output schema (§5.10) enriches sidecar `MetadataEntry` objects with filesystem provenance fields (`file_system`, `size`, `timestamps`) that the v1 schema lacked. These fields record the sidecar file's original relative path, byte size, and modification timestamps at the time of merging. This provenance data serves as a reversal manifest: a future `revert` operation (§6.10) can reconstruct deleted sidecar files by writing the `data` field content back to the original path, restoring the file size, and setting the timestamps.

Without the v2 provenance fields, MetaMergeDelete would be a one-way operation — the merged data would be present in the parent entry's `metadata` array, but the information needed to reconstruct the original files (path, size, timestamps) would be lost. The v2 schema's enrichment is a safety feature, not merely a schema improvement.

The reversibility guarantee is structural: the provenance fields are populated for every sidecar entry with `origin: "sidecar"`, regardless of whether MetaMergeDelete is active. Even when MetaMerge (without Delete) is used, the provenance data is recorded, allowing a future Delete operation to be applied to an existing index without re-indexing.

Safeguard 5: Dry-run mode interaction

When `--dry-run` is active alongside `--rename`, the rename operation is simulated without executing (§6.10). However, `--dry-run` does NOT interact with MetaMergeDelete — there is no `--dry-run` equivalent for sidecar deletion. If a user requests `--meta-merge-delete --dry-run`, the `--dry-run` flag

applies only to renames; sidecar deletion still occurs if the configuration validation passes.

This is a deliberate design constraint, not an oversight. Simulating MetaMergeDelete without actually deleting would produce the same output as MetaMerge (without Delete) — there is no observable difference in a dry-run. If a user wants to preview the effect of MetaMergeDelete, they should run with `--meta-merge` first (which merges without deleting), inspect the output to verify correctness, and then run with `--meta-merge-delete` to commit the deletion. The documentation SHOULD make this workflow explicit.

16.5. Large File and Deep Recursion Handling

Large file hashing

The primary resource risk during file hashing is memory consumption. A naïve implementation that reads an entire file into memory before hashing would fail catastrophically for files larger than available RAM. The `hash_file()` function (§6.3) reads files in fixed-size chunks (64 KB default, §17.2) and feeds each chunk to all active hash objects. Peak memory usage is bounded by the chunk size plus the hash objects' internal state — approximately 70 KB regardless of file size. A 100 GB video file and a 1 KB text file consume the same amount of memory during hashing.

The chunk-based approach also provides natural interruption points. If a `cancel_event` is checked between chunks (the GUI entry point may implement this for responsiveness), cancellation latency is bounded by the time to read one chunk, not the time to read the entire file.

No file size limit is enforced by the indexer. The practical ceiling is the filesystem's maximum file size (16 TB for NTFS, 16 TiB for ext4, 8 EiB for APFS). Files approaching these limits will take proportionally longer to hash, but the indexer will not run out of memory, crash, or produce incorrect results.

The `exiftool` invocation (§6.6) includes the `-api largefilesupport=1` argument in the default argument list (§7.4), which enables exiftool's large-file handling for files exceeding 2 GB. Without this flag, exiftool truncates metadata extraction at the 2 GB boundary for certain file formats.

Base64-encoded sidecar size

When the sidecar module (§6.7) reads a binary sidecar file (thumbnails, screenshots, torrent files) for Base64 encoding, the entire file is read into memory via `path.read_bytes()` and then Base64-encoded. Base64 encoding expands the data by approximately 33%, so a 100 MB thumbnail would consume roughly 133 MB of memory during encoding and remain in memory as part of the `MetadataEntry.data` field throughout the entry's lifetime.

For typical sidecar files — thumbnails are usually under 1 MB, torrent files under 10 MB — this is not a concern. For pathological cases where a user has multi-hundred-megabyte binary sidecars, memory consumption could become significant. The indexer does not implement streaming Base64 encoding for sidecar files because the encoded data must be held in memory as a JSON-serializable string regardless of how it was encoded. The mitigation is documentation: the user-facing documentation SHOULD note that very large binary sidecars contribute directly to memory usage and output file size, and that excluding them via the metadata exclusion configuration (§7.5) is appropriate when memory constraints are tight.

Deep directory recursion

Python's default recursion limit is 1,000 frames. The recursive traversal in `build_directory_entry()` (§6.8) adds one Python call frame per level of directory nesting. A directory tree nested 900 levels deep would approach the default recursion limit, and a 1,000-level tree would hit it, producing a `RecursionError`.

Directory trees deeper than a few hundred levels are virtually nonexistent in real-world filesystems. The deepest common nesting — `node_modules` dependency trees — rarely exceeds 30–50 levels. Nevertheless, the implementation SHOULD handle the edge case gracefully rather than crashing with an unhandled `RecursionError`.

Two mitigation strategies are available:

Strategy A — Increase the recursion limit. At the start of the indexing operation, call `sys.setrecursionlimit(max(sys.getrecursionlimit(), 10_000))`. This raises the ceiling to 10,000 levels, which exceeds any realistic directory depth. The cost is a trivially larger thread stack allocation.

Strategy B — Iterative traversal. Replace the recursive `build_directory_entry()` call with an iterative depth-first traversal using an explicit stack. This eliminates the recursion limit entirely and is the more robust approach, but it adds implementation complexity (the explicit stack must maintain the same parent–child assembly semantics as the recursive version).

The specification recommends **Strategy A** for the MVP, with a comment in the code noting that Strategy B is the long-term solution if the recursion limit proves insufficient. Strategy A is sufficient for all realistic workloads and requires a single line of code. Strategy B is a refactoring exercise that can be deferred without safety risk.

If a `RecursionError` does occur despite the raised limit, it is caught by the top-level exception handler (§8.10) and produces exit code 4 (`RUNTIME_ERROR`) with a diagnostic message suggesting that the user reduce directory depth or report the issue.

Large directory enumeration

A single directory containing a very large number of entries — tens or hundreds of thousands of files — does not pose a recursion risk but does affect memory usage and performance. `os.scandir()` returns an iterator that yields `DirEntry` objects lazily, so the directory listing is not loaded entirely into memory at once. However, the `list_children()` function (§6.1) collects all entries into two sorted lists (files and directories), which does materialize the full listing.

For a directory with 100,000 entries, the materialized lists consume approximately 10–20 MB of memory (one `Path` object per entry). This is within acceptable bounds for any system that can run the indexer. For directories with millions of entries — rare but possible on large media servers or backup volumes — the memory consumption scales linearly and may become significant on constrained systems.

The indexer does not implement streaming or batched entry processing for single-directory enumeration. The sorted-list approach is required for deterministic output ordering (§6.1), and the memory overhead is proportional to the directory size, not the total tree size. A directory with 1,000,000 files but only 10 levels of nesting requires ~100 MB for the single largest directory listing, not for the entire tree.

JSON serialization memory

For recursive directory indexing, the complete `IndexEntry` tree is held in memory until serialization is complete (§6.9). The tree's memory footprint is proportional to the total number of items indexed multiplied by the average entry size. A rough estimate: each `IndexEntry` consumes 2–5 KB of memory (hash strings, paths, timestamps, metadata references), so a tree of 100,000 items occupies 200–500 MB.

For very large trees (millions of items), this memory footprint may exceed available RAM. The mitigation is the `--inplace` output mode, which writes each item's sidecar file immediately after construction and does not require the full tree to be held in memory simultaneously. When `--inplace` is the sole output mode (no `--stdout`, no `--outfile`), the entry builder can release child entries after writing their sidecars, reducing peak memory to the depth of the tree rather than its breadth.

This streaming release optimization is not required for MVP. The MVP implementation MAY hold the full tree in memory for all output modes. The optimization SHOULD be implemented in a performance pass if users report memory issues with large trees. The architectural separation between entry construction (§6.8) and serialization (§6.9) supports this optimization without structural changes — the serializer already writes in-place sidecars during traversal, so the change is to release the child reference after the write completes.

The `--outfile` and `--stdout` output modes inherently require the full tree in memory (the aggregate JSON document must be complete before serialization). For users indexing very large trees, the documentation SHOULD recommend `--inplace` as the memory-efficient alternative, with a post-processing step to aggregate the individual sidecar files if a single output document is needed.

17. Performance Considerations

This section defines the performance characteristics, optimization strategies, and resource consumption boundaries of `shruggie-indexer`. It is the normative reference for how the indexer manages computational cost across its five performance-sensitive operations: cryptographic hashing, file I/O, directory enumeration, JSON serialization, and external process invocation. Each subsection describes the optimization approach, the rationale behind the chosen strategy, and the bounds within which the optimization holds.

> **Historical note:** The original `MakeIndex` has no documented performance design — its performance characteristics are emergent side effects of implementation choices made for convenience (separate file reads per hash algorithm, piping through `jq`, writing Base64-encoded arguments to temporary files for exiftool). `shruggie-indexer` improves on three of these incidental costs significantly and maintains parity on the others.

This section makes the performance design explicit so that implementers do not inadvertently regress and so that future optimization work can target the areas with the highest payoff.

§6 (Core Operations) defines what each operation does. §14.7 (Performance Benchmarks) defines how performance is measured and what the baseline expectations are. §16.5 (Large File and Deep Recursion Handling) defines the resource consumption safety boundaries. This section sits between those references — it explains *why* the implementation is structured for performance, *how* the key optimizations work, and *where* the remaining performance bottlenecks lie.

17.1. Multi-Algorithm Hashing in a Single Pass

The problem

The indexer computes multiple cryptographic hash digests for every file: MD5 and SHA256 by default, with SHA512 as an opt-in (§6.3). A naïve implementation — computing each algorithm in a separate pass — reads the file from disk once per algorithm. For a 1 GB file with two algorithms, this means 2 GB of I/O. For three algorithms, 3 GB. File I/O is overwhelmingly the dominant cost of hashing for any file larger than a few kilobytes; the CPU time spent computing digests is negligible by comparison. The naïve multi-pass approach doubles or triples the I/O cost for zero additional information — every pass reads the same bytes.

This is exactly what the original does. `FileId` defines independent sub-functions for each algorithm (`FileId-HashMd5`, `FileId-HashSha256`, etc.), each of which opens the file via `[System.IO.File]::OpenRead()`, reads it to completion, computes the digest, and closes the file. When two algorithms are requested (the default), the file is opened, read, and closed twice. The .NET `CryptoStream` class supports chaining multiple hash transforms on a single stream, but the original does not use this capability.

The optimization

The indexer reads each file exactly once, regardless of how many hash algorithms are active. `hash_file()` (§6.3) creates one `hashlib` hash object per algorithm, reads the file in chunks, and feeds each chunk to every hash object before reading the next chunk:

```
# Illustrative — not the exact implementation.
def hash_file(path: Path, algorithms: tuple[str, ...] = ("md5", "sha256")) -> HashSet:
```

```
    hashers = {alg: hashlib.new(alg) for alg in algorithms}
    with open(path, "rb") as f:
        while chunk := f.read(CHUNK_SIZE):
            for h in hashers.values():
                h.update(chunk)
    return HashSet(**{alg: h.hexdigest().upper() for alg, h in hashers.items()})
```

The key insight is that `hashlib` hash objects accept incremental `update()` calls — they maintain internal state across multiple chunk submissions and produce the same digest as if the entire content had been submitted in a single call. This is a property of all Merkle–Damgård hash constructions (which MD5, SHA-1, SHA-256, and SHA-512 all are). The per-chunk CPU cost of calling `update()` on N hash objects is linear in N, but the per-chunk I/O cost is constant — the same bytes are in memory regardless of how many hash objects consume them.

Impact

For the default two-algorithm case (MD5 + SHA256), the single-pass approach halves total file I/O compared to the original. For the three-algorithm case (MD5 + SHA256 + SHA512), it reduces I/O to one-third. The absolute time savings depend on storage throughput: on an SSD reading at 500 MB/s, hashing a 1 GB file takes approximately 2 seconds in single-pass versus approximately 4 seconds in dual-pass. On spinning disks (100–150 MB/s) or network storage, the savings are proportionally larger because the I/O penalty of additional passes is more severe.

The CPU overhead of feeding each chunk to multiple hash objects is negligible. Python's `hashlib` delegates to OpenSSL's C implementations, which process a 64 KB chunk in microseconds. The overhead of iterating a 2- or 3-element dict per chunk is immeasurable relative to the `read()` system call latency.

When this matters — and when it does not

The single-pass optimization has no impact for string hashing (`hash_string()`) or directory identity computation (`hash_directory_id()`), both of which operate on short in-memory byte sequences that are hashed instantaneously. The optimization is meaningful only for `hash_file()` on files large enough that I/O time dominates — roughly, files larger than 100 KB. For smaller files, the cost is dominated by file-open/close overhead, and the difference between one pass and two is imperceptible. The benchmarks in §14.7 validate this expectation: the small-file benchmark (1 KB) measures < 10 ms regardless of algorithm count, while the large-file benchmark (100 MB) measures throughput that approaches raw `hashlib` speed.

17.2. Chunked File Reading

Chunk size selection

`hash_file()` reads files in fixed-size chunks of 65,536 bytes (64 KB). This value is not configurable — it is an implementation constant internal to `core/hashing.py`.

The chunk size represents a balance between two opposing costs. Smaller chunks increase the number of `read()` system calls per file, each of which carries kernel-transition overhead. Larger chunks increase the per-read memory allocation and, past the OS file cache page size, offer diminishing returns because the OS prefetcher is already streaming data ahead of the application's read position. Python's `hashlib` documentation recommends chunk sizes between 4 KB and 128 KB for stream hashing. Empirical testing on Linux (ext4, SSD) and Windows (NTFS, SSD) shows throughput plateaus above approximately 32 KB; 64 KB was chosen as a comfortable margin above the plateau that works well across all three target platforms.

The following data points informed the selection:

Chunk size	Approximate throughput (SHA256, SSD)	Notes
4 KB	~200 MB/s	Syscall overhead visible.
16 KB	~400 MB/s	Improving rapidly.
32 KB	~480 MB/s	Near plateau.
64 KB	~500 MB/s	At plateau. Chosen value.
128 KB	~500 MB/s	No further gain.
1 MB	~500 MB/s	No further gain; higher per-chunk memory.

These figures are illustrative and vary by platform, filesystem, and storage medium. The 64 KB value is not performance-critical in the sense that a different choice would produce a dramatically different outcome — any value between 32 KB and 256 KB produces comparable throughput. The value is fixed rather than configurable because exposing it as a tuning parameter invites cargo-cult optimization without measurable benefit and adds configuration complexity for no user-facing gain.

Memory bound

The chunk-based approach bounds peak memory consumption during hashing to approximately `CHUNK_SIZE` (64 KB) plus the internal state of the active hash objects (a few hundred bytes each). A 100 GB file and a 100-byte file consume the same peak memory during hashing. This property is the memory safety guarantee described in §16.5 — the hashing module cannot cause an out-of-memory condition regardless of file size.

The chunk boundary also provides natural interruption points. If the GUI entry point implements a cancellation check between chunks (§10.5), cancellation latency is bounded by the time to read one 64 KB chunk — under 1 millisecond on any local storage device. This responsiveness is not achievable with a whole-file-read approach.

17.3. Large Directory Tree Handling

Traversal performance

The traversal module (`core/traversal.py`, §6.1) enumerates directory contents using `os.scandir()` in a single pass. This is both a correctness improvement and a performance improvement over the original's dual-pass `Get-ChildItem` approach.

The original performs two separate `Get-ChildItem` calls per directory — one with the `-File` flag to retrieve files, one with the `-Directory` flag to retrieve directories. Each call independently enumerates the entire directory, applies its filter, and returns the matching subset. For a directory with 10,000 entries, this means two complete readdir traversals, two complete filter passes, and two result-set materializations. The data that distinguishes files from directories (the `d_type` field in POSIX `readdir`, the `dwFileAttributes` field in Windows `FindNextFile`) is available on both passes but only consulted on one.

`shruggie-indexer`'s `os.scandir()` reads the directory once. Each `DirEntry` object returned by `scandir` caches the file/directory classification from the underlying OS call, so `entry.is_file(follow_symlinks=False)` and `entry.is_dir(follow_symlinks=False)` resolve without an additional `stat()` call on platforms that provide `d_type` (Linux, macOS) or `dwFileAttributes` (Windows). Classification and separation into the files and directories lists happen in a single iteration.

For a directory with N entries, the original performs 2 × O(N) directory reads plus 2 × O(N) filter passes. The indexer performs 1 × O(N) directory read plus 1 × O(N) classification pass. On spinning disks where directory reads are seek-limited, or on network filesystems where each readdir round-trip has latency, the single-pass approach can be up to twice as fast for large directories.

Sorting cost

After enumeration, the indexer sorts both the file list and the directory list lexicographically by name (case-insensitive). The sort uses Python's Timsort (`sorted()` with a `key` function), which runs in O(N log N) for N entries. For 10,000 entries, this is approximately 130,000 comparisons — completed in milliseconds. For 100,000 entries, approximately 1.7 million comparisons — still well under one second. The sorting cost is negligible relative to the per-item entry construction cost (hashing, stat, potential exiftool invocation).

The original does not explicitly sort — `Get-ChildItem` returns entries in filesystem order (which on NTFS is B-tree sorted, but on ext4 and APFS is effectively directory-creation order). The explicit sort ensures deterministic output ordering across platforms, at a cost that is imperceptible for any realistic directory size.

Memory consumption during traversal

`list_children()` materializes the complete file and directory lists for each directory being processed. For a directory with N entries, this allocates approximately N `Path` objects — each consuming roughly 100–200 bytes of Python heap (the object header plus the string path data). A directory with 100,000 entries thus requires 10–20 MB of memory for the materialized lists.

This memory is allocated per-directory, not per-tree. When processing a recursive tree, only the listing for the *currently-active directory* is materialized at any given time. The file and directory lists for a parent directory are held in memory while child directories are being processed (because the parent's file list has already been iterated, but the directory list is being iterated), so the peak listing memory is proportional to the *maximum directory width at any single level along the current depth-first path*, not to the total entry count of the tree. For a tree with 1,000,000 entries distributed across 1,000 directories of 1,000 entries each, the peak listing memory is approximately 200 KB per directory × the number of directories on the current recursive stack — typically under 10 MB.

The §16.5 discussion of streaming release for `--inplace` mode applies to the *entry tree* held in memory after construction, not to the traversal listings. The traversal listings are transient and released as each directory finishes processing.

17.4. JSON Serialization for Large Output Trees

Serialization cost

The serializer (§6.9) converts a completed `IndexEntry` tree to JSON text via `dataclasses.asdict()` followed by `json.dumps()`. For small to medium trees (up to a few hundred entries), serialization time is negligible — a 100-entry tree serializes in under 100 milliseconds. For large trees, serialization can become a measurable fraction of total runtime because the standard library `json.dumps()` operates in pure Python for dict traversal, even though the string encoding is implemented in C.

The following estimates illustrate the cost scaling for pretty-printed JSON output (`indent=2`, `ensure_ascii=False`):

Tree size (entries)	Approximate output size	`json.dumps()` time	Notes
100	~400 KB	< 100 ms	Negligible.
1,000	~4 MB	~500 ms	Measurable but acceptable.
10,000	~40 MB	~5 seconds	Noticeable. Approaches the hashing cost of the tree.
100,000	~400 MB	~50 seconds	Significant. May exceed the hashing time for SSD-backed trees.

These estimates assume approximately 4 KB of JSON per entry (the average for entries with two hash algorithms, timestamps, path components, and no metadata). Entries with large EXIF metadata or Base64-encoded sidecar content will be proportionally larger.

The `orjson` acceleration path

The `orjson` package (a required dependency, §12.3) provides the primary serialization path via `orjson.dumps()`. `orjson` is a Rust-backed JSON library that serializes Python dicts 3–10× faster than `json.dumps()` for typical workloads. For a 10,000-entry tree, serialization drops from approximately 5 seconds to under 1 second.

Because `orjson` is a required dependency, the serializer uses it as the primary path with a `json.dumps()` fallback for resilience (§12.5):

```
# Illustrative — not the exact implementation.
import orjson

def serialize_entry(entry: IndexEntry, *, compact: bool = False) -> str:
    entry_dict = dataclasses.asdict(entry)
    option = 0 if compact else orjson.OPT_INDENT_2
    return orjson.dumps(entry_dict, option=option).decode("utf-8")
```

The `orjson` path returns bytes, which are decoded to a UTF-8 string for API compatibility. The decode cost is minor relative to the serialization savings. A `json.dumps()` fallback is retained as a defensive measure in case `orjson` is somehow unavailable at runtime.

For large trees (10,000+ entries) using `--stdout` or `--outfile` output modes, the `orjson` serialization path provides substantial acceleration automatically. Users of `--inplace` mode (which serializes one entry at a time) see proportionally less benefit because individual entry serialization is already fast.

`dataclasses.asdict()` overhead

The `dataclasses.asdict()` call is itself a non-trivial cost for large trees. It performs a recursive deep-copy of the entire entry tree, converting every dataclass instance to a plain dict and every list to a new list. For a 10,000-entry tree, this deep copy can take 1–2 seconds and temporarily doubles the memory footprint of the tree (the original dataclass tree and the dict copy coexist until `asdict()` returns and serialization begins).

If `orjson` is available, this overhead can be eliminated entirely: `orjson` serializes dataclasses natively, without requiring a `dataclasses.asdict()` conversion step. The implementation MAY bypass `asdict()` when `orjson` is the active serializer, passing the root `IndexEntry` directly to `orjson.dumps()` with `option=orjson.OPT_PASSTHROUGH_DATACLASS` (or equivalent). This requires that all fields on all dataclass types are `orjson`-serializable (which they are — the schema uses only strings, ints, floats, bools, lists, dicts, and `None`).

The `json.dumps()` fallback path cannot bypass `asdict()` because `json.dumps()` does not handle dataclass instances without a custom encoder. The overhead is accepted for the stdlib path; the `orjson` path eliminates it.

Compact vs. pretty-printed output

Pretty-printed output (`indent=2`) produces approximately 40% more bytes than compact output for typical `IndexEntry` data — the indentation whitespace and newlines add up across deeply nested structures. The serialization time difference between compact and pretty-printed is minimal (the formatter's whitespace insertion is trivial), but the I/O cost of writing a 40% larger file is not.

The `--compact` flag (§8.3) selects compact output. For large trees written to `--outfile`, compact output reduces both serialization time (less string concatenation) and file write time. For `--stdout` piped to a downstream consumer, compact output reduces pipe throughput requirements. The default is pretty-printed for human readability; the recommendation for automated pipelines processing large trees is `--compact`.

17.5. Exiftool Invocation Strategy

The dominant cost

For files that have embedded metadata, the `exiftool` invocation (§6.6) is overwhelmingly the most expensive per-file operation in the indexing pipeline. Hashing a 10 MB JPEG takes approximately 20–40 ms (limited by file read speed). Extracting EXIF metadata from the same file via `exiftool` takes 200–500 ms — an order of magnitude more. The cost is almost entirely `exiftool` process startup: the Perl interpreter loads, exiftool's module tree initializes, the file is read, metadata is extracted, and JSON output is produced. For a directory of 1,000 JPEG files, exiftool invocations alone can account for 3–8 minutes of total runtime, dwarfing the combined cost of hashing, stat calls, sidecar discovery, and serialization.

This cost profile is inherited from the original, which invokes exiftool once per file via `GetFileExifRun`. The original routes the invocation through a Base64-decoded argument file and a `jq` post-processing pipeline, adding further per-file overhead (temporary file creation, `certutil` invocation for Base64 decoding, `jq` process startup, and temporary file cleanup). `shruggie-indexer` eliminates the argument-file machinery and the `jq` pipeline (DEV-05, DEV-06) and uses `pyexiftool`'s `-stay_open` batch mode (DEV-16) as the primary backend, reducing per-file cost from 200–500 ms to 20–50 ms.

Batch invocation: the primary approach (DEV-16)

The primary exiftool backend uses the `pyexiftool` package (`>=0.5`, a required runtime dependency) to maintain a single persistent exiftool process for the duration of the indexing run. `pyexiftool` wraps exiftool's `-stay_open` mode: file paths and arguments are written to exiftool's stdin pipe one per line, and

JSON output is read from stdout for each file as it completes. The per-file cost drops from 200–500 ms (dominated by process startup) to 20–50 ms (dominated by actual metadata extraction).

```
# Illustrative — not the exact implementation.
import exiftool

with exiftool.ExifToolHelper(common_args=EXIFTOOL_COMMON_ARGS) as et:
    for path in eligible_files:
        try:
            metadata = et.get_metadata(str(path))
        except Exception:
            # Per-file error handling — see below
            ...
```

The stdin pipe protocol provides an additional benefit: it inherits the same Unicode safety guarantees as exiftool's `-@` argfile interface (documented in exiftool FAQ §18 and the WINDOWS UNICODE FILE NAMES section). File paths written to stdin are passed directly to exiftool's internal filename handler with `-charset filename=utf8`, bypassing shell escaping and OS-level command-line argument encoding entirely. This resolves the special-character argument-passing risks that motivated the original's Base64 encoding pipeline.

Error isolation in batch mode. The persistent-process model changes the error isolation characteristics compared to per-file `subprocess.run()`. The exif module wraps each per-file `get_metadata()` call in a try/except and distinguishes two failure categories:

1. **Recoverable per-file error** (non-zero exit with valid JSON output). `ExifToolHelper` raises `ExifToolExecuteError` when exiftool returns a non-zero exit code. The handler inspects the exception's stdout for valid JSON metadata. If present and non-trivial (keys beyond `SourceFile`), the metadata is parsed, filtered, and returned normally. The persistent process is **not** reset — a per-file non-zero exit (e.g., exit code 1 for "unknown file type") does not indicate process failure. This matches the original MakeIndex behavior, which captured stdout regardless of exit code.

2. **True process failure** (crash, broken pipe, timeout). If the exception does not contain recoverable stdout, the handler resets the `ExifToolHelper` by exiting and re-entering the context manager to spawn a fresh exiftool process. The failed file is logged as a field-level error and processing continues.

Both categories provide the same item-level error boundary defined in §4.5 — a single file's exiftool failure does not affect any other file. The distinction between categories avoids the performance penalty of unnecessary process restarts for the common "unknown file type" case. See §6.6 for the full error handling table.

Subprocess + argfile: the fallback approach

If `pyexiftool` cannot maintain a stable connection to the exiftool process (e.g., in constrained environments, or if the package is uninstalled), the exif module falls back to a per-file `subprocess.run()` invocation. Arguments are written to a temporary file via `tempfile.NamedTemporaryFile` and passed to exiftool via its `-@` argfile switch:

```
# Illustrative — not the exact implementation.
result = subprocess.run(
    ["exiftool", "-@", argfile_path],
    capture_output=True,
    text=True,
    timeout=30,
)
```

The fallback uses argfile-based argument passing (not direct command-line arguments) to preserve Unicode filename safety — the argfile mechanism respects `-charset filename=utf8` and avoids the shell escaping issues that affect direct argument passing on some platforms. The `timeout=30` parameter bounds worst-case latency for a single invocation.

This fallback exists for resilience. It is not a supported primary configuration — users who install the package normally will always have `pyexiftool` available. The fallback is retained because the cost of maintaining it is near-zero and it provides a safety net for edge cases.

Availability probe cost

The exiftool availability probe (`shutil.which("exiftool")`, §6.6) runs once per process lifetime. On most systems, `shutil.which()` resolves in under 1 millisecond by scanning the `PATH` directories. The cost is amortized over the entire invocation and is never repeated — the result is cached in a module-level variable.

The original has no availability probe (§4.5). It discovers exiftool's absence through per-file failure: each `GetFileExifRun` call spawns a subprocess that immediately fails, producing a per-file error and the associated overhead of process creation and error-string construction. For a directory of 10,000 files with no exiftool installed, the original spawns and fails 10,000 subprocesses. The probe-once approach converts this from O(N) failed subprocess invocations to a single `shutil.which()` call — a performance improvement that is also a usability improvement (one warning message instead of 10,000 error messages).

Extension exclusion as a performance gate

Before invoking exiftool, the module checks the file's extension against the exclusion list (§7.4). The default exclusion set (`csv`, `htm`, `html`, `json`, `tsv`, `xml`) targets file types where exiftool tends to dump the entire file content into the metadata output rather than extracting meaningful embedded metadata. Excluding these types avoids both the subprocess cost and the downstream cost of serializing and storing the bloated metadata output.

The exclusion check is a `frozenset` membership test — O(1) per file. For a mixed-content directory where 30–50% of files are non-media types (a common scenario in project directories that contain code, data, and media), the exclusion filter can eliminate a significant fraction of exiftool invocations without any loss of useful metadata. This is a carry-forward from the original's `$global:MetadataFileParser.Exiftool.Exclude` list, and it serves the same purpose — the tool preserves this gate exactly as-is because it is one of the original's better performance decisions.

Timeout as a safety bound

The 30-second per-file timeout (§6.6) serves as both a safety mechanism (§16.5) and a performance bound. In the batch backend (`pyexiftool`), the timeout is enforced at the `ExifToolHelper` level per `get_metadata()` call. In the subprocess fallback, it is the `timeout` parameter on `subprocess.run()`. Without a timeout, a pathological file — a multi-gigabyte video with deeply nested metadata structures, or a corrupted file that causes exiftool to enter an infinite analysis loop — could block the indexer indefinitely. The timeout ensures that no single file can consume more than 30 seconds of exiftool processing time. When the timeout fires, the file's metadata is recorded as `None` (the exiftool `MetadataEntry` is omitted from the `metadata` array), a warning is logged, and processing continues with the next file.

The 30-second value is generous for normal operations — typical EXIF extraction completes in under 1 second for even large media files — but it could be exceeded for very large video files with embedded subtitle tracks or chapter metadata. The timeout is currently not configurable. If users report legitimate timeout hits on files that exiftool can process (just slowly), the value MAY be exposed as a configuration parameter in a future update. For the MVP, 30 seconds provides a wide safety margin without requiring user tuning.

18. Future Considerations

This section catalogs enhancements, extensions, and architectural evolution paths that are explicitly deferred from the v0.1.0 (MVP) release. Every item listed here has been referenced from at least one normative section of this specification — this section consolidates those references into a single planning document, describes each item's scope and preconditions, and identifies the architectural constraints that govern how the item can be added without disrupting the existing system.

This section is informational, not normative. Nothing described here constitutes a commitment to implement. The items are organized by likelihood and dependency order: §18.1 covers concrete feature additions that the MVP architecture already supports structurally; §18.2 covers output schema evolution, which has broader compatibility implications; §18.3 covers a plugin or extension architecture, which would be the most significant structural change and is therefore the most speculative.

18.1. Potential Feature Additions

This subsection collects every post-MVP enhancement that has been identified during specification development, organized by the functional area of the codebase that would be affected. Each item includes the originating reference, a description of what the enhancement involves, the preconditions or dependencies that must be satisfied before implementation, and an assessment of the architectural impact — whether the enhancement fits cleanly within the existing module boundaries or requires structural changes.

18.1.1. v1-to-v2 Migration Utility

Originating references: §1.2 (Out of Scope), §2.3 (G7, NG2), §5.5, §5.13 (Backward Compatibility).

This is the most clearly defined post-MVP deliverable. The utility converts existing v1 index sidecar files (`_meta.json`, `_directorymeta.json`) to the v2 format (`_meta2.json`, `_directorymeta2.json`). The conversion is lossy in one direction: v1 fields dropped in v2 (`Encoding`, `BaseName`, `SHA1` hashes) are discarded. It is enriching in the other: v2 fields with no v1 equivalent (`schema_version`, `id_algorithm`, `type`, `mime_type`, `size.text`, `file_system.relative`, and all `MetadataEntry` provenance fields) are populated with computed or default values where possible and `null` where not.

The migration utility is a standalone command — likely a new CLI subcommand (`shruggie-indexer migrate`) or a separate script in the `scripts/` directory. It does NOT require modifications to the core indexing engine or the v2 dataclass definitions. The primary implementation challenge is correctly identifying v1 documents (by the absence of `schema_version` or the presence of v1-specific fields like `_id` with its `y`/`x` prefix) and mapping v1's flat PascalCase fields to v2's nested snake_case sub-objects.

Preconditions: The v2 schema and output pipeline must be stable. The utility should not ship until the MVP has been validated against the v2 schema in production-like usage, confirming that the v2 field semantics are correct and complete.

Architectural impact: Low. The utility consumes the existing `models/schema.py` dataclasses for v2 output construction and the `core/serializer.py` module for JSON output. It adds a new module (e.g., `tools/migrate.py` or `cli/commands/migrate.py`) with no changes to existing modules.

18.1.2. Rename Revert Operation

Originating references: §6.10 (File Rename and In-Place Write Operations).

The original's source comments include a "To-Do" note about adding a `Revert` parameter. The v2 schema's enriched `MetadataEntry` provenance fields (§5.10, principle P3) and the in-place sidecar files provide the data foundation for reversal: the sidecar file written alongside each renamed item records the original filename in `name.text`, allowing a revert operation to reconstruct the original path.

A `revert_rename()` function would read the sidecar's `name.text` field and rename the file back to its original name. The implementation is straightforward — iterate sidecar files in a target directory, parse each one, extract `name.text`, and rename the associated storage-named file back to its original name. The primary complexity is conflict detection: if the original filename already exists (because a different file now occupies that name), the revert must fail gracefully for that file.

Preconditions: The rename operation (§6.10) and in-place write mode (§6.9) must be stable and producing correct sidecar files with complete `name.text` provenance.

Architectural impact: Low. A new function in `core/rename.py` (or a new `core/revert.py` module), a new CLI flag (`--revert`), and corresponding API surface. No changes to existing modules beyond adding the new entry point.

18.1.3. Exiftool Batch Mode via PyExifTool

Status: Implemented in MVP (DEV-16). The `pyexiftool` package is a required runtime dependency. The batch mode implementation using `ExifToolHelper` and exiftool's `-stay_open` protocol is the primary exiftool backend in the MVP. See §6.6 and §17.5 for the full implementation details including error isolation, backend selection logic, and the subprocess+argfile fallback.

Remaining post-MVP work: The current implementation processes files one at a time through the batch pipe (synchronous per-file `get_metadata()` calls within the `ExifToolHelper` context). A further optimization could pre-batch file paths and process them in parallel query groups, or overlap exiftool I/O with hashing I/O using an async pipeline. These are incremental improvements within the existing `core/exif.py` module boundary and do not require structural changes.

18.1.4. CLI Graceful Interrupt Handling (SIGINT)

Originating references: §10.5 (GUI cancellation), §8.10 (Exit Codes).

The GUI defines a cancellation mechanism (§10.5) that interrupts the indexing loop between items and raises `IndexerCancelled`. The CLI does not currently support mid-operation cancellation — a `SIGINT` (Ctrl+C) terminates the process immediately, potentially leaving partially-written output files.

A graceful interrupt handler would register a `signal.signal(signal.SIGINT, ...)` handler that sets a cancellation flag checked at item boundaries in the traversal loop. When triggered, the handler would complete the current item, write partial output (for `--outfile` or `--inplace` modes), clean up any open file handles, and exit with a distinct exit code indicating interrupted operation.

Preconditions: The item-level error boundary (§4.5) already provides the check-point structure — the cancellation flag check is a natural addition to the existing per-item try/except boundary. The `--inplace` mode already writes incrementally, so partial output is inherently safe. The `--outfile` and `--stdout` modes require a decision about whether to write a partial tree or discard everything.

Architectural impact: Low to moderate. A signal handler in `cli/main.py`, a cancellation flag threading through the orchestrator, and a new exit code. The GUI's `IndexerCancelled` exception (§10.5) may be reusable for both surfaces.

18.1.5. Depth-Limited Recursion

Originating references: §9.2 (Core Functions architectural note).

The `recursive` parameter on `build_directory_entry()` is a boolean. A depth-limited variant (`max_depth=N`) would allow users to index only the top N levels of a directory tree, which is useful for large repository checkouts or deeply nested media libraries where only the top-level structure is of interest.

The implementation is straightforward: replace the boolean `recursive` parameter with an integer depth counter that decrements at each level. When the counter reaches zero, child directories are listed but not entered. The CLI would expose this as `--depth N` (or `--max-depth N`), with the default being unlimited (`None` or `-1`).

Preconditions: None beyond a stable recursive traversal implementation.

Architectural impact: Low. A parameter type change in `core/entry.py` and `core/traversal.py`, a new CLI flag, and a corresponding API parameter. The recursive call structure does not change — only the recursion guard condition.

18.1.6. Unicode Normalization Control

Originating references: §6.3 (DEV-15), §15.3 (macOS HFS+/APFS).

Status: Partially implemented in MVP. The MVP applies unconditional NFC normalization to all strings before hashing (DEV-15), ensuring cross-platform hash determinism. This resolves the original concern about HFS+ NFD filenames producing different hashes than NFC filenames on other platforms.

A future enhancement could add a `--no-normalize-unicode` CLI flag (or `normalize_unicode: false` config option) that disables NFC normalization, reverting to the "hash what the filesystem returns" behavior. This would be useful for users who need filesystem-level fidelity over cross-platform consistency — for example, forensic analysis of macOS HFS+ volumes where the NFD form is meaningful.

Preconditions: The NFC normalization implementation (DEV-15) must be stable and well-tested.

Architectural impact: Low. A conditional gate around the existing `unicodedata.normalize()` call in `core/hashing.hash_string()`, controlled by a config flag. The config system, CLI parser, and API surface each gain one additional option.

18.1.7. Windows `.lnk` Shortcut Resolution

Originating references: §12.4 (Eliminated Original Dependencies — `Lnk2Path`).

The original resolves Windows `.lnk` shortcut files to their target paths via the `Lnk2Path` function, which uses COM interop. The cross-platform port treats `.lnk` files encountered as sidecar content as opaque binary data and Base64-encodes them.

A post-MVP enhancement could add `.lnk` resolution on Windows using the optional `pylnk3` package (or COM interop via `win32com`). The resolution would extract the target path from the shortcut and store it as the sidecar's `data` field instead of the raw binary content. On non-Windows platforms, the fallback to Base64 encoding would remain.

Preconditions: A suitable `.lnk` parsing library must be identified. `pylnk3` is the most common pure-Python option. The enhancement should be gated by an import guard, similar to the `pyexiftool` pattern.

Architectural impact: Low. Changes confined to `core/sidecar.py` (or the metadata read logic for link-type sidecars). A new optional dependency declared in `pyproject.toml`.

18.1.8. Exiftool Runtime Version Checking

Originating references: §12.1 (Required External Binaries).

The tool requires exiftool ≥ 12.0 for the `-api requestall=3` and `-api largefilesupport=1` arguments but does not enforce version checking at runtime. An older exiftool will likely work for most files but may produce incomplete metadata.

A version check would invoke `exiftool -ver` once at startup (alongside the existing `shutil.which()` availability probe), parse the version string, and emit a warning if the version is below the minimum. The check cost is negligible — one additional subprocess invocation per process lifetime, completed in under 100 ms.

Preconditions: A stable exiftool availability probe (§17.5) must be in place. The version check should run immediately after the availability probe succeeds.

Architectural impact: Minimal. A few lines added to the exiftool availability probe in `core/exif.py`.

18.1.9. Configurable Exiftool Timeout

Originating references: §6.6, §17.5.

The 30-second exiftool timeout is currently a hardcoded constant. If users encounter legitimate timeout hits on files that exiftool can process (just slowly) — such as very large video files with deeply nested metadata — the timeout could be exposed as a configuration parameter.

Preconditions: User reports of legitimate timeout hits. Without evidence of real-world need, adding a configuration surface for this value is premature.

Architectural impact: Minimal. A new field in `IndexerConfig`, passed through to `subprocess.run(timeout=...)`.

18.1.10. Structured Performance Tracking

Originating references: §14.7 (Performance Benchmarks).

The MVP's benchmarks produce unstructured timing output reviewed manually. A post-MVP enhancement could integrate `pytest-benchmark` for structured performance tracking with statistical analysis, historical trend lines, and automated regression detection.

Preconditions: A stable benchmark suite and a CI pipeline that preserves benchmark results across runs.

Architectural impact: Minimal. A new dev dependency, benchmark fixture changes in `tests/benchmarks/`, and CI configuration updates. No changes to production code.

18.1.11. Platform-Specific Installers

Originating references: §13.5 (Release Artifact Inventory).

The MVP distributes standalone executables. Platform-specific installers (`.msi` for Windows, `.dmg` for macOS, `.deb`/`.rpm` for Linux) would improve user experience with features like Start Menu integration, Applications folder placement, and package manager updates.

Preconditions: A stable release pipeline and executable build process. Installer creation tools (`WiX` for `.msi`, `create-dmg` for `.dmg`, `fpm` for Linux packages) must be integrated into the GitHub Actions workflow.

Architectural impact: None to production code. Build pipeline additions only.

18.1.12. Formal Accessibility Improvements

Originating references: §10.7 (Keyboard Shortcuts and Accessibility).

The MVP makes reasonable accommodations for keyboard-only operation but does not target formal accessibility compliance (WCAG, Section 508). CustomTkinter inherits `tkinter`'s native accessibility support, which varies by platform — Windows provides the best screen reader integration via UI

Automation, while Linux/macOS support is limited.

Formal accessibility work would involve audit against WCAG guidelines, addition of ARIA-equivalent labels where CustomTkinter supports them, high-contrast theme variants, and screen reader testing on all three platforms.

Preconditions: A stable GUI implementation and user feedback indicating accessibility is a priority.

Architectural impact: Moderate within the GUI layer (`gui/`). No impact on the core engine or CLI.

18.1.13. End-User Documentation

Originating references: §3.6 (Documentation Artifacts), §3.7 (Documentation Site).

The documentation site infrastructure — MkDocs with Material for MkDocs, the `mkdocs.yml` configuration, the GitHub Actions deployment workflow, and the `docs/user/` stub pages — is created as part of the MVP repository scaffolding. This establishes the complete site navigation skeleton and deployment pipeline from day one.

Content authoring for user-facing documentation (`docs/user/`) is incremental and is not gated on the MVP release. Pages are populated as the CLI interface and configuration system stabilize. For the v0.1.0 release, user-facing documentation content is limited to the `README.md` at the repository root, while the `docs/user/` stub pages contain placeholder text directing readers to the README. A complete documentation set — authored incrementally post-MVP — would include an installation guide, quick-start tutorial, configuration reference, and changelog.

Preconditions for content authoring: A stable CLI interface and configuration system. Documentation written against an unstable interface creates maintenance burden. The documentation site infrastructure itself has no preconditions — it is deployable with stub content.

Architectural impact: Minimal. Adds `mkdocs.yml` at the repository root, a GitHub Actions workflow (`.github/workflows/docs.yml`), optional Python dependencies (`mkdocs`, `mkdocs-material`) in the `docs` extras group, and stub Markdown files in `docs/user/`. No impact on the core engine, CLI, or GUI.

18.1.14. Session ID in JSON Output

Originating references: §11.4 (Session Identifiers).

The session ID (a UUID4 generated per invocation) currently appears only in log output. A future schema version could include it in the JSON output metadata to link an index entry back to the invocation that produced it. This would support provenance tracking — given an index file, a user could correlate it with the log output from the run that created it.

This item is deliberately listed as a feature addition rather than a schema evolution item (§18.2) because it is a purely additive field that does not change existing semantics. It could be added as a new top-level field (e.g., `session_id`) in the v2 schema without breaking backward compatibility, or it could be deferred to a v3 schema if it accompanies other breaking changes.

Preconditions: Confirmation that the session ID is useful to downstream consumers. The field adds bytes to every output file for a benefit that may be niche.

Architectural impact: Minimal. A new field on the `IndexEntry` dataclass, populated by the orchestrator from the session context.

18.2. Schema Evolution

The v2 output schema includes a `schema_version` discriminator field (§5.3) whose express purpose is to enable schema evolution. The discriminator allows consumers to detect the schema version before parsing and to dispatch to version-specific parsing logic. This subsection describes the principles governing schema evolution, the known candidates for a future v3 schema, and the compatibility constraints that any schema change must satisfy.

18.2.1. Evolution Principles

Additive changes are non-breaking. A new optional field added to an existing object (e.g., a `session_id` field on `IndexEntry`, or a `created_source` field on `TimestampsObject`) does not break existing consumers. Consumers that do not recognize the field ignore it. The `schema_version` value does not need to change for purely additive fields — the v2 schema's `additionalProperties: false` constraint would need to be relaxed or the canonical JSON Schema updated to include the new field, but the discriminator value can remain `2` as long as no existing field's semantics change.

Structural changes require a version bump. Any change that renames a field, changes a field's type, removes a required field, or alters the semantic meaning of an existing field constitutes a breaking change and MUST increment `schema_version`. Consumers dispatch on `schema_version` and expect the fields listed for that version.

Deprecation before removal. If a v2 field is to be removed in v3, a transition period should be provided: the field is marked as deprecated in v2.x documentation, emitted but ignored by the tool, and finally removed in v3. The migration utility pattern established for v1-to-v2 (§18.1.1) should be replicated for any future version transition.

Schema-version-specific serialization. The serializer (§6.9) currently hardcodes `schema_version: 2`. If a future version supports emitting multiple schema versions (e.g., a `--schema-version 3` flag for early adopters), the serializer must dispatch to version-specific field sets and object structures. This is a non-trivial change and should be avoided unless there is a compelling reason to support concurrent version output.

18.2.2. Candidate v3 Additions

The following fields have been identified during specification development as candidates for future schema versions. None of these are committed — they are recorded here so that future development can evaluate them against actual user needs.

`timestamps.created_source` (§15.5). A string field on `TimestampsObject` indicating the provenance of the creation timestamp — `"birthtime"` when derived from `st_birthtime` (macOS, Windows) or `"ctime_fallback"` when derived from `st_ctime` (Linux, where `ctime` represents the last inode change, not creation). This field would resolve the ambiguity documented in §15.5 about what `timestamps.created` actually represents on different platforms.

`session_id` (§18.1.14). A UUID4 string linking the index entry to the invocation that produced it. See §18.1.14 for the full description. This is additive and could be added to v2 without a version bump.

`encoding` (§5.11). If encoding detection becomes a requirement, a new field with a Python-native structure (not the .NET-specific `System.Text.Encoding` serialization from v1) would be added. The structure might include `bom` (detected BOM, if any), `detected_encoding` (best-guess encoding name from `chardet` or similar), and `confidence` (detection confidence score). This would be a new top-level field — not a restoration of the v1 `Encoding` field, which is explicitly dropped without replacement.

`type` enum extension (§5.4). The `type` field currently uses a two-value string enum (`"file"`, `"directory"`). The enum was designed as extensible (§5.4) — a future version could add `"symlink"` as a distinct type rather than a boolean flag, allowing consumers to dispatch on item type without checking a separate symlink field. This would be a semantic change to the `type` field (expanding its value set) and should be handled carefully: existing consumers that switch on `type` and assume only two values would need updating.

18.2.3. Compatibility Strategy

The v2 schema's design already accommodates the most likely evolution paths:

The `schema_version` discriminator enables version detection. The `type` field uses a string enum rather than a boolean, enabling value-set expansion. The `MetadataEntry.origin` field uses a string enum (`"sidecar"`, `"generated"`) that can be extended with new origin types. The `HashSet` object's `sha512` field demonstrates the pattern for optional algorithm-specific fields — additional algorithms can be added as new optional properties without breaking consumers that expect only `md5` and `sha256`.

The v2 sidecar filename convention (`_meta2.json`, `_directorymeta2.json`) embeds the schema version in the filename. A v3 schema would use `_meta3.json` and `_directorymeta3.json`, allowing v2 and v3 sidecar files to coexist on disk. This is the same coexistence pattern used for the v1-to-v2 transition (§5.13).

18.3. Plugin or Extension Architecture

The MVP does not include a plugin system. All behavior is compiled into the package — sidecar parsers, hash algorithms, metadata extractors, and output formatters are all defined in the core modules and configured through the externalized configuration system (§7). This subsection evaluates whether a plugin architecture is warranted, what it would look like, and under what conditions it should be considered.

18.3.1. Current Extensibility Mechanisms

The MVP already provides meaningful extensibility through configuration rather than code:

The sidecar discovery system (§7.3) uses regex patterns defined in configuration. Adding a new sidecar type requires adding a new regex pattern and type definition to the configuration file — no code changes. The exiftool exclusion list (§7.4) is similarly configurable. The filesystem exclusion filters (§7.2) allow users to add platform-specific or project-specific directory exclusions. The extension validation regex (§7.5, DEV-14) is externalized for the same reason.

These configuration-driven extension points cover the most common customization needs: "I have a new metadata file pattern that the indexer doesn't recognize" and "I have files or directories that should be excluded." For these use cases, the configuration system is the correct mechanism — a plugin architecture would be overengineering.

18.3.2. Where Plugins Would Add Value

A plugin architecture would become valuable if users need to extend the indexer's behavior in ways that configuration cannot express:

Custom metadata extractors. The MVP extracts metadata via exiftool (for embedded EXIF/XMP data) and via the sidecar parser (for external metadata files). If users need to extract metadata from domain-specific formats that exiftool does not support — such as proprietary CAD file metadata, scientific data file headers (HDF5, NetCDF), or application-specific config files — a plugin interface for metadata extractors would allow them to register a callable that receives a file path and returns a `MetadataEntry` (or `None`).

Custom output formatters. The MVP outputs JSON exclusively. If downstream consumers need CSV, XML, SQLite, or protocol buffer output, a formatter plugin interface would allow custom serializers to be registered alongside the built-in JSON serializer.

Custom identity schemes. The MVP computes identity from content hashes (MD5, SHA256, optionally SHA512). Some use cases may require alternative identity schemes — e.g., BLAKE3 for performance, or content-addressable storage identifiers that combine hash with size. A hash algorithm plugin interface would allow new algorithms to be registered and included in `HashSet` output.

18.3.3. Recommended Approach

If a plugin architecture is pursued, the recommended approach is a lightweight entry-point-based system using Python's `importlib.metadata.entry_points()` (available in Python ≥ 3.12, the project's baseline). Plugins would be installed as separate Python packages that

declare entry points in their `pyproject.toml`:

```
# In a hypothetical shruggie-indexer-hdf5 plugin's pyproject.toml:
[project.entry-points."shruggie_indexer.extractors"]
hdf5 = "shruggie_indexer_hdf5:extract_hdf5_metadata"
```

The indexer would discover installed plugins at startup via `entry_points(group="shruggie_indexer.extractors")` and register them alongside the built-in extractors. This approach requires no changes to the indexer's core architecture — the discovery is additive, the plugin callable conforms to an interface defined by the indexer, and uninstalling the plugin removes the behavior.

This is the standard Python pattern for extensible applications (used by `pytest`, `setuptools`, `tox`, and many others). It does not require a custom plugin loader, a plugin directory, or a plugin configuration file — the Python packaging system handles discovery and installation.

18.3.4. When to Implement

A plugin architecture should NOT be implemented speculatively. The configuration-based extensibility in the MVP covers the known customization needs. The plugin architecture should be pursued only when concrete user requests demonstrate a need that configuration cannot satisfy — specifically, when users need to run custom Python code as part of the indexing pipeline, not just adjust parameters.

The architectural preparation for a future plugin system is minimal: the hub-and-spoke module design (§4.2) already isolates each functional area behind a defined interface (`extract_exif()`, `discover_sidecars()`, `hash_file()`, `serialize_entry()`). Converting any of these interfaces to a plugin dispatch point requires adding an entry-point discovery step at module initialization and iterating over registered plugins alongside the built-in implementation. The structural refactoring cost is low when the need materializes.