Digital Audio Assets Protocol

Three-Part Module Manual v1.0

In-Session Manifest | Package Preparation | Package Authority Integration

Prepared as a three-stage DAAP software flow.
Each module remains a distinct stage, but no single module constitutes a complete DAAP asset by itself.

Combined scope	Stage 1: In-Session Manifest Utility | Stage 2: Package Prep Utility | Stage 3: Package Authority Integrator
Prepared by	Nquist LLC | 2026

Table of Contents

The file listings in each section reflect the current codebase structure. Where helpful, files are identified both by their standalone filename and by their repository package path so their role in the module and their location in the codebase can be understood together.

Section I: In-Session Manifest Module

User Manual | REAPER Lua Edition

Version 1.0.0

Application File	DAAP_InSession_Manifest_UI_v0.2.0.lua
Environment	REAPER + ReaScript + ReaImGui
Current Framework	Lua / ReaImGui
Planned Future Form	C++ / JUCE plugin

Section I

Stage 1 of the DAAP software flow

This stage captures the approved session-facing structural snapshot inside the DAW before external packaging begins.

Application File Set

Role	Codebase file name(s)
Primary application file	DAAP_InSession_Manifest_UI_v0.2.0.lua
Project-local supporting code files	None in the current uploaded in-session codebase. The utility is implemented as a single Lua script.
External runtime dependencies	REAPER + ReaScript + ReaImGui (runtime dependencies, not project-local code files)

Prepared for users working inside REAPER who need to inspect and capture the current state of a session before later packaging or protocol preparation.

GitHub Quick Start

This page is intended to get a REAPER user from repository download to a running in-session utility with the fewest possible steps.

> **WARNING.** This application will not open unless ReaImGui is installed through ReaPack.

> **NOTE.** This manual covers the in-session REAPER Lua utility only. It does not cover the separate external packaging-side application.

Download and Install

1. Open the official GitHub repository page. https://github.com/Nquist-LLC/In-Session-Manifest-Utility-for-the-Digital-Audio-Assets-Protocol-DAAP-
2. Click Code and choose Download ZIP if you want the full repository as a compressed folder or clone the repository if you are using Git.
3. Extract the downloaded folder to a location you can access easily from REAPER.
4. Locate the in-session REAPER script file: DAAP_InSession_Manifest_UI_v0.2.0.lua.
5. Open REAPER.
6. Go to Actions > Load ReaScript.
7. Select the Lua file and load it into REAPER.
8. Run the script from the Actions list, or assign it to a toolbar button, shortcut, or custom action.

Required Components

- REAPER
- ReaPack
- ReaImGui
- The Lua script file itself

Table of Contents

1. About This Application

The DAAP In-Session Manifest Utility is a REAPER-based inspection tool that runs inside the current session and displays a structured view of the session state. It is designed to help users review what is in the project at the time of capture, including track structure, plugin chains, parameter snapshots, and routing relationships.

Beyond inspection, this utility functions as the entry point into the DAAP framework, serving as the initial layer where session data is observed, organized, and prepared for formalization into a DAAP-compliant asset. In this role, it acts as the gateway between a working DAW session and the standardized asset structure required for secure distribution, playback, and downstream interoperability within the Digital Delivery Triad. By grounding the asset in the actual session state, rather than a flattened export, it ensures that the integrity, attribution, and structural relationships of the project are preserved at the moment of capture.

This application is currently written in Lua and uses ReaImGui to present a dockable panel in REAPER. It is intended to later evolve into an official C++ / JUCE-based plugin, but for now it is used as a REAPER script

2. What the Application Does

> **NOTE.** The displayed session data represents a cached snapshot. It is not a frame-by-frame live monitor of REAPER state.

When opened, the utility scans the active REAPER project and builds a cached snapshot of the session. That snapshot includes:

- project identity
- track count
- track names and GUIDs
- item counts
- track channel counts
- mute, solo, and phase states
- FX chains
- plugin status
- plugin preset names when available
- parameter snapshots up to the configured cap
- sends
- receives
- hardware outputs
- project-wide plugin inventory
- analytics summaries
- master track information when included

The snapshot model is intentional. It gives the user a stable capture of the session rather than a constantly changing live feed, so the screen reflects a defined session state until the user chooses to rescan it.

3. What the Application Does Not Do

This version is intentionally limited. It does not:

- package a final DAAP asset
- validate stems against an external folder
- compare session structure to exported media on disk
- prepare C2PA structures
- prepare DDEX delivery material
- perform cryptographic signing
- modify the REAPER session
- write changes back into tracks, plugins, or routing

It is an inspection and capture surface, not the downstream packaging application.

4. System Requirements

To use this application, you need:

- REAPER
- ReaPack
- ReaImGui
- the Lua script file itself

The script checks for ReaImGui at startup. If ReaImGui is missing, it shows a message telling the user to install it through Extensions > ReaPack > Browse packages… > search “ReaImGui.”

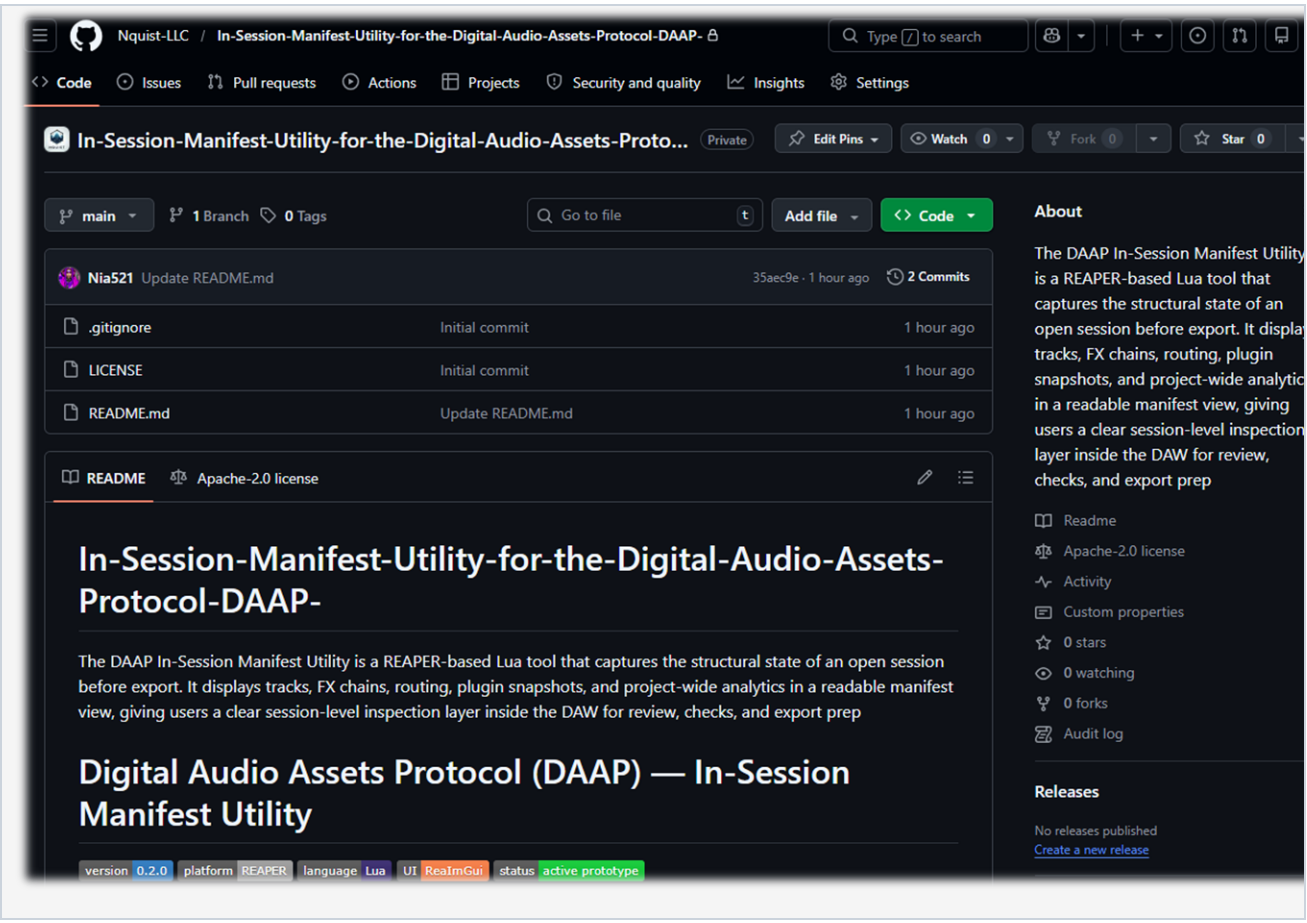
> **WARNING.** If ReaImGui is missing, the application will not open. Install it through ReaPack before troubleshooting anything else.

5. Downloading the Tool from GitHub

> **NOTE.** Replace the placeholder repository URL with the final public repository link before distribution.

Use the repository page as the authoritative source for the current script file, release notes, and installation assets. In the final release version of this manual, replace the placeholder repository URL with the actual public link.

A GitHub-facing front section is included at the beginning of this manual so new users can reach a working installation quickly before reading the full guide.



6. Installing ReaPack and ReaImGui

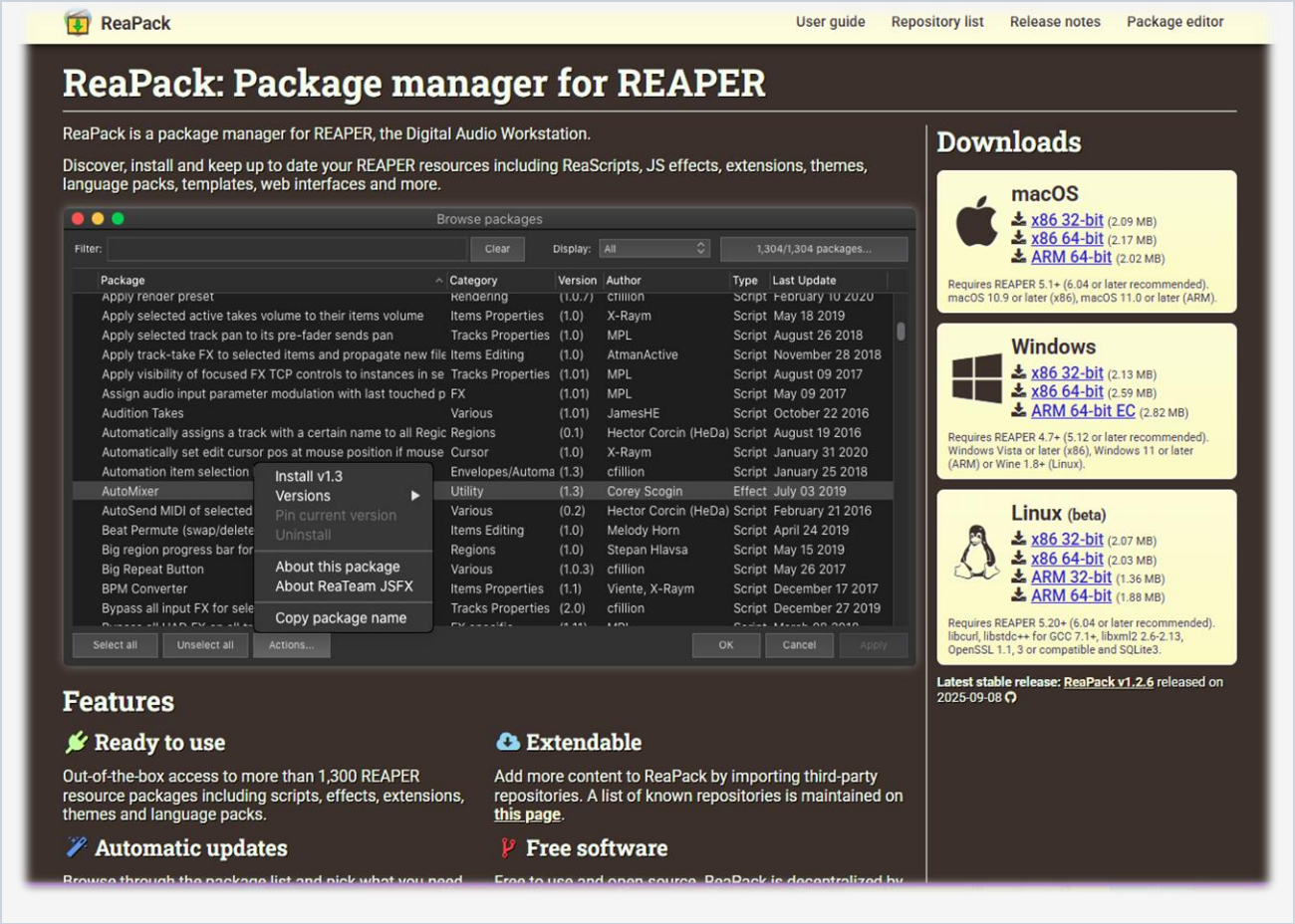
Because this application currently runs as a Lua script in REAPER, users must install the required REAPER scripting tools before launching it.

Step 1: Install ReaPack

1. Download ReaPack from its official site.
2. Follow the installation steps for your operating system.
3. Restart REAPER.

Step 2: Install ReaImGui

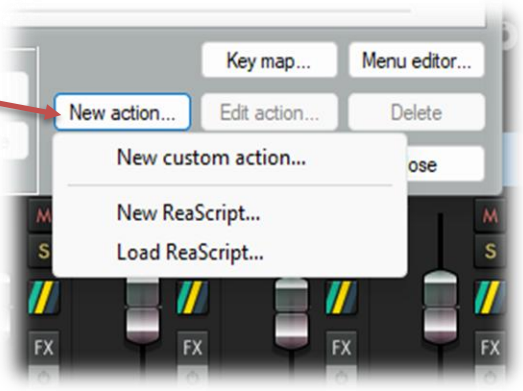
1. Open REAPER.
2. Go to Extensions.
3. Choose ReaPack.
4. Select Browse Packages...
5. Search for ReaImGui.
6. Install “ReaTeam Extensions/API/ReaImGui: ReaScript binding for Dear ImGui.” [ReaPack: Package manager for REAPER]



7. Installing the Lua Script in REAPER

After downloading the repository to C:\Users\[user name]\AppData\Roaming\REAPER\Scripts

1. Open REAPER.
2. Open the Actions list.
3. Select the New Action Button then select Load ReaScript.
4. Browse to DAAP_InSession_Manifest_UI_v0.2.0.lua.
5. Load the script.
6. Optionally assign it to a toolbar button, a keyboard shortcut, or a custom action. Once loaded, it can be launched from the Actions list like any other REAPER script.



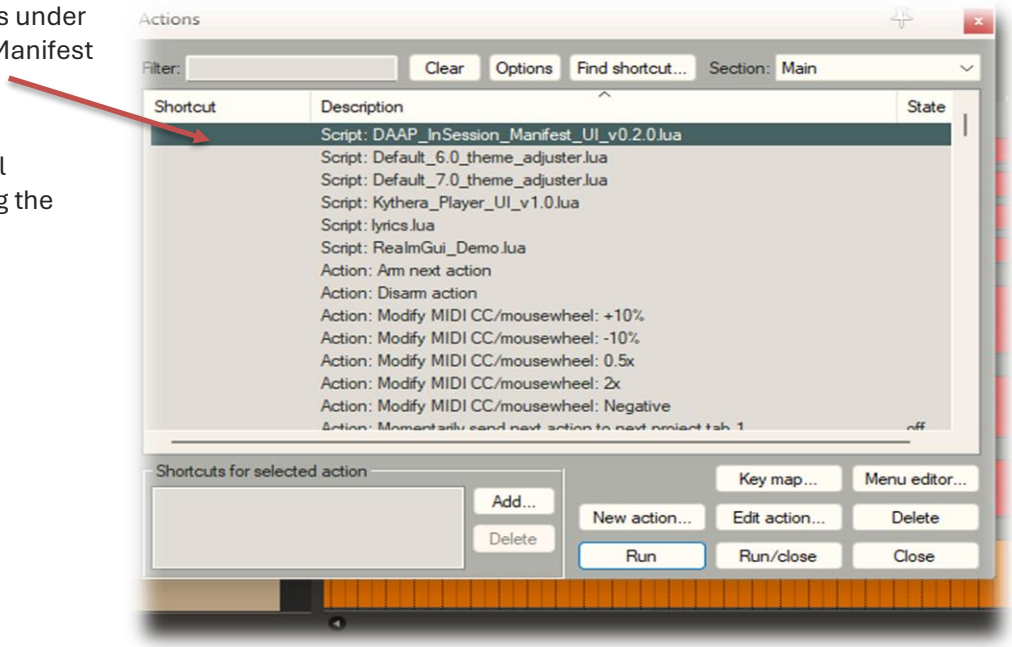
8. Launching the Application

> **WARNING.** If the session changes after launch, the screen will not update automatically. Use Refresh Snapshot whenever you want a new capture.

To open the utility:

1. Open the REAPER project you want to inspect.
2. Run DAAP_InSession_Manifest_UI_v0.2.0.lua.
3. The application window opens under the title “DAAP | In-Session Manifest Utility v0.2.0.”

When the script launches, it automatically performs an initial snapshot refresh before entering the main UI loop.



9. If the Application Does Not Open

If the script is launched and ReaImGui is not installed, REAPER shows a message box stating that ReaImGui is not available and instructing the user to install it via ReaPack.

Message you may see

“ReaImGui is not available.”

What it means

The script cannot create its interface because the ReaImGui API is missing.

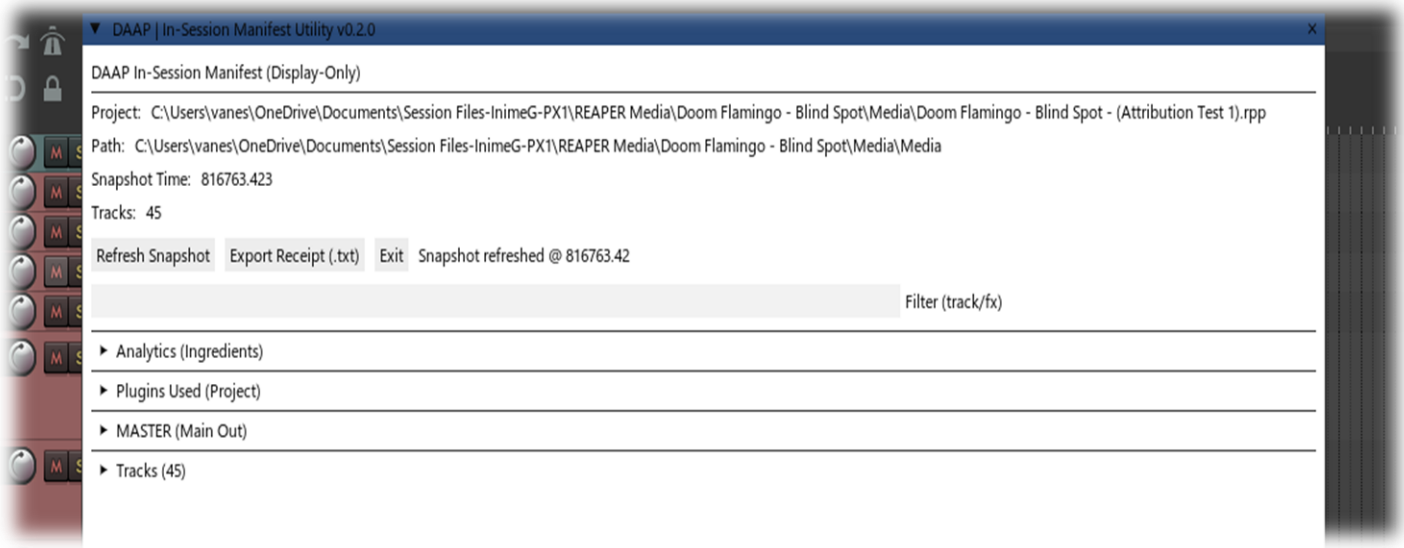
What to do

Install ReaImGui through ReaPack, then restart REAPER and launch the script again.

10. Main Window Overview

The main window is laid out from top to bottom in a simple document-like format. The tool uses a white-background paper-manifest style rather than a stylized plugin look.

The main visible areas are Project identity, Control buttons, Filter field, Analytics, Plugins Used (Project), Master, and Tracks. This order matters because the tool is meant to move from project-wide context down into individual tracks.



11. Project Identity Area

At the top of the window, the application shows:

- Project
- Path
- Snapshot Time
- Tracks

At the top of the window, the application shows Project, Path, Snapshot Time, and Tracks. These values are read from the current project state and the snapshot cache.

What this section means

This tells you exactly which project the utility inspected and when the current snapshot was generated.

Why it matters

If you make changes in the session after opening the tool, this area helps confirm whether what you are looking at is current or whether you need to refresh the snapshot.

12. Control Buttons

NOTE. A successful refresh produces a time-stamped status note indicating when the new snapshot was generated.

WARNING. If export fails, check write permissions and confirm that the project or fallback REAPER resource location is available.

The application displays three main buttons at the top:

- Refresh Snapshot
- Export Receipt (.txt)
- Exit

These are placed together in the main control row of the interface.

Refresh Snapshot

Press this button whenever you want the tool to rescan the current REAPER session. The code rebuilds the internal snapshot and updates the session view. After refresh, the application displays a note such as "Snapshot refreshed @ [time]."

Use Refresh Snapshot when

- you added or removed tracks
- you changed plugin chains
- you edited routing
- you changed mute, solo, or phase states
- you want an updated manifest view

Export Receipt (.txt)

Press this button to write a plain-text manifest receipt to disk. The file includes project information, analytics, plugins used, track details, FX chain details, parameter captures, and routing summaries.

When successful, the application shows a confirmation message box that says the manifest receipt was saved and gives the full path to the exported file.

Exit

Closes the application window.

13. Filter Field

Below the control row is a field labeled “Filter (track/fx).”

This field lets the user narrow the visible track list by searching against:

• track names

• FX names

How it works

If you type text into this field, the Tracks section only shows tracks whose names match the text or whose FX chain contains a plugin name matching the text.

What it is good for

• isolating vocals

• finding all drum buses

• locating a plugin used across several tracks

• reducing clutter in large projects

What it does not do

It does not change the data. It only changes what is shown.

14. Analytics Section

> **NOTE.** Plugin classification is heuristic in this version. The analytics section is meant to aid inspection, not serve as a final plugin identity database.

The first collapsible analysis block is “Analytics (Ingredients).” When expanded, it shows:

• total plugin instances

• unique plugins

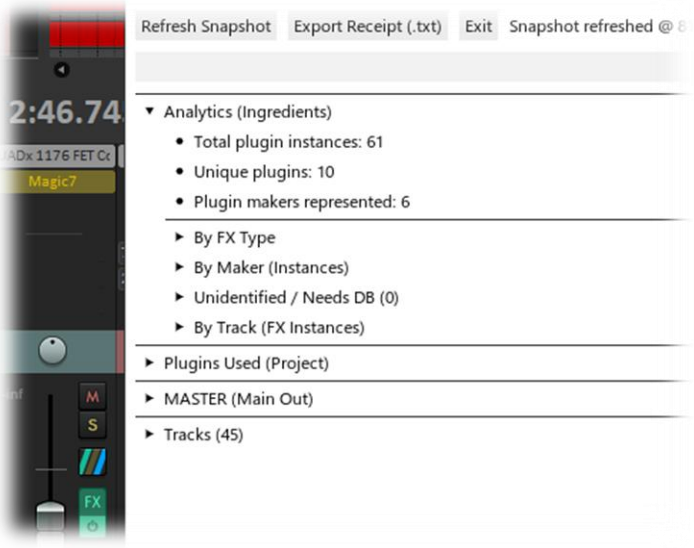
• plugin makers represented

• by FX type

• by maker

• unidentified / needs DB

• by track (FX instances)



What each subsection means

Total plugin instances

The total number of plugin instances found across all scanned tracks and the master track.

Unique plugins

The number of distinct plugin names found after basic normalization.

Plugin makers represented

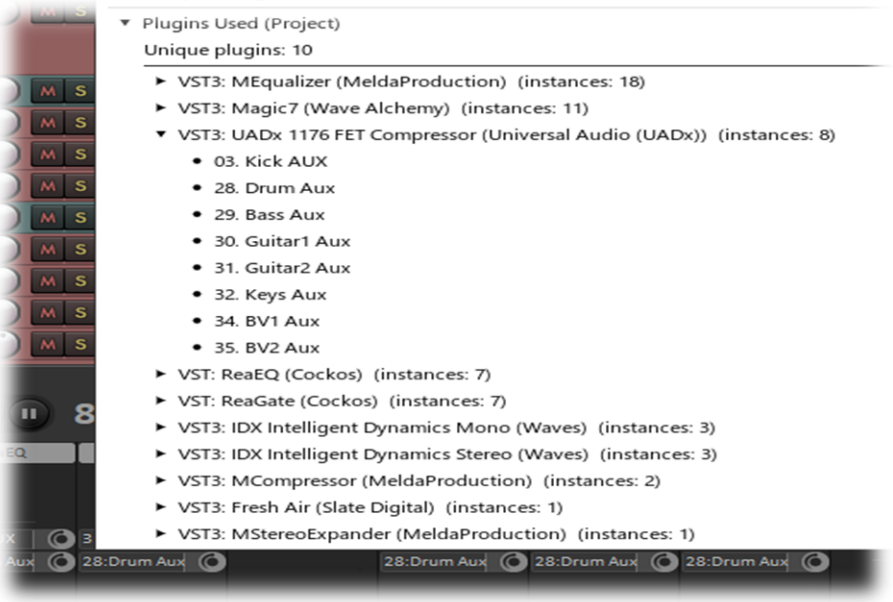
A count of the detected makers or vendors found from plugin naming patterns.

By FX Type
Groups plugin instances by classification such as EQ, Compressor, Reverb, Delay, Limiter, and others.

By Maker (Instances)
Shows how many plugin instances are associated with each detected maker.

Unidentified / Needs DB
Shows plugins that the current heuristics could not confidently classify by maker and type.

By Track (FX Instances)
Shows how many FX instances appear on each track.



15. Plugins Used (Project) Section

The next collapsible block is “Plugins Used (Project).” This is the project-wide plugin inventory.

What it shows
- the plugin name
- number of instances
- the tracks where it appears

How to use it
Expand a plugin name to see which tracks contain that plugin. This is useful when you want to answer questions such as:

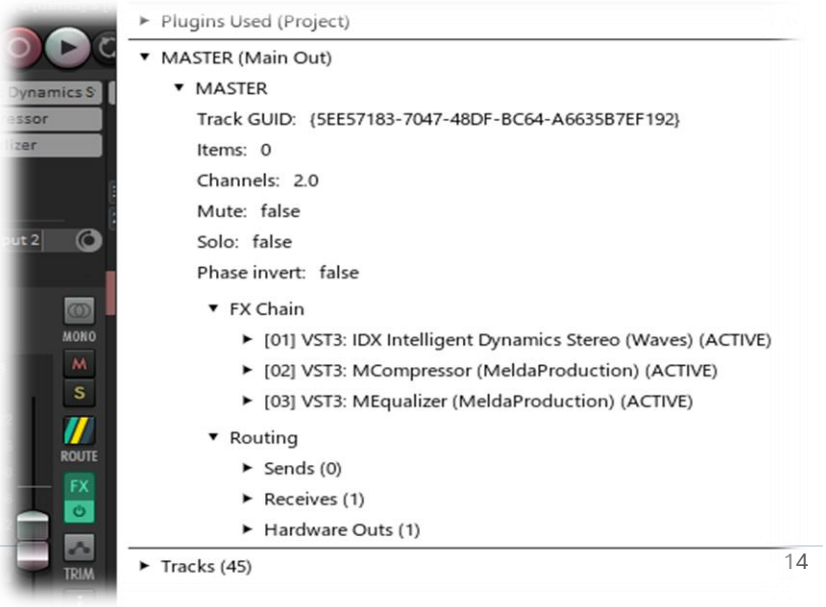
- Where am I using this plugin?
- How many instances of this plugin are in the session?
- Which tracks share the same processor?

If no plugins are present, the section displays “(none).”

16. Master Section

If master inclusion is enabled, the utility shows “MASTER (Main Out).” This is treated like a special track manifest for the main output bus.

What you can inspect here
- master GUID
- item count
- channel count
- mute, solo, phase



- FX chain
- routing

Why it matters

Within the manifest, the master section records the state of the final output path through which the session is being represented for approval, print, or delivery. This is not merely another track view. It is the point at which session structure converges into the distributable signal path. Any processing present here materially shapes the authoritative output of the session, which makes it relevant to asset identity, auditability, and downstream interpretation. For that reason, the main output path belongs in the manifest as part of the session's captured structural truth.

17. Tracks Section

The main body of the utility is "Tracks (N)," where N is the number of scanned tracks. Each track is shown as a collapsible entry.

What each track header represents

Each entry corresponds to one scanned REAPER track. The display name is built from the track number and track name when available, such as "01. Vocal Lead" or "02. Snare Top."

What appears when a track is expanded

- Track GUID
- Items
- Channels
- Mute
- Solo
- Phase invert
- FX Chain
- Routing

18. FX Chain Expansion

> **WARNING.** UNAVAILABLE usually indicates a missing, unresolved, or unreadable effect. Review the session carefully if this appears on an important track.

> **NOTE.** If a plugin has more than the display cap, the utility shows a message noting how many additional parameters were not displayed.

Inside each expanded track is an FX Chain pull-down.

This section lists each plugin in order and labels it with a status. The code displays statuses as:

- ACTIVE
- BYPASSED
- OFFLINE
- UNAVAILABLE

What these statuses mean

ACTIVE
The plugin is enabled and available.

BYPASSED
The plugin exists in the chain but is not currently active.

OFFLINE
The plugin is offline.

UNAVAILABLE
The script could not resolve a real plugin identity or name, or REAPER indicates the effect is missing or unavailable.

What each FX entry may show

- FX GUID
- preset name
- parameter count captured versus total
- parameter list
- message indicating additional parameters are not displayed if the parameter cap is reached

The parameter capture is limited to a configured maximum of 64 parameters per FX for display safety.

If you see

"... (X more params not displayed)" it means the FX has more parameters than the current UI display limit.

19. Routing Expansion

> **NOTE.** Routing is shown in simplified manifest form so the user can understand signal relationships quickly during inspection.

Inside each expanded track is also a Routing pull-down. When expanded, it displays three additional pull-down areas:

- Sends
- Receives
- Hardware Outs

Sends

Sends show where the track sends audio.

Each entry includes:

- destination track name
- volume
- pan
- source channel
- destination channel
- mute state

Receives

Receives show what is feeding audio into the track.

Each entry includes:

- source track name
- volume
- pan
- source channel
- destination channel
- mute state

Hardware Outs

Hardware Outs show hardware output routing.

Each entry includes:

- volume
- pan
- source channel
- destination channel
- mute state

If you see

"(none)" means no entries were found in that routing category.

"(no routing data)" means the tool did not build a routing block for that track.

20. Manifest Receipt Export

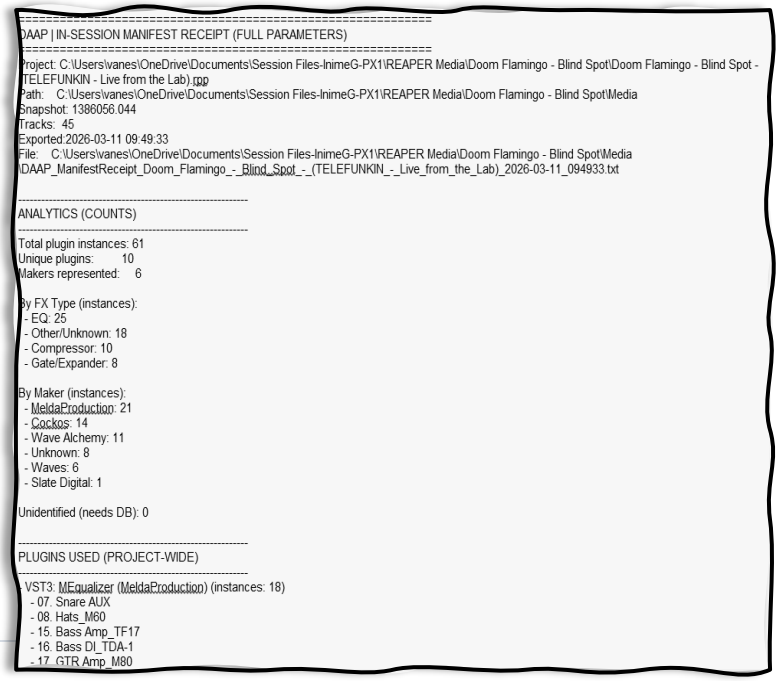
The Export Receipt (.txt) button writes a text report named in this format:

The script saves it to: **DAAP_ManifestReceipt_[ProjectName]_[Timestamp].txt**

- the current project directory when possible
- otherwise the REAPER resource path as a fallback

What the receipt contains

- project name
- project path
- snapshot timestamp
- track count
- export timestamp
- analytics counts
- by FX type
- by maker
- unidentified plugin list



```
================================================================
DAAP | IN-SESSION MANIFEST RECEIPT (FULL PARAMETERS)
================================================================
Project: C:\Users\vanes\OneDrive\Documents\Session Files-InimeG-PX1\REAPER Media\Doom Flamingo - Blind Spot\Doom Flamingo - Blind Spot - (TELEFUNKIN - Live from the Lab).rpp
Path:    C:\Users\vanes\OneDrive\Documents\Session Files-InimeG-PX1\REAPER Media\Doom Flamingo - Blind Spot\Media
Snapshot: 1386056.044
Tracks:  45
Exported:2026-03-11 09:49:33
File:    C:\Users\vanes\OneDrive\Documents\Session Files-InimeG-PX1\REAPER Media\Doom Flamingo - Blind Spot\Media
\DAAP_ManifestReceipt_Doom_Flamingo_-_Blind_Spot_-_(TELEFUNKIN_-_Live_from_the_Lab)_2026-03-11_094933.txt

----------------------------------------------------------------
ANALYTICS (COUNTS)
----------------------------------------------------------------
Total plugin instances: 61
Unique plugins:         10
Makers represented:     6

By FX Type (instances):
 - EQ: 25
 - Other/Unknown: 18
 - Compressor: 10
 - Gate/Expander: 8

By Maker (instances):
 - MeldaProduction: 21
 - Cockos: 14
 - Wave Alchemy: 11
 - Unknown: 8
 - Waves: 6
 - Slate Digital: 1

Unidentified (needs DB): 0

----------------------------------------------------------------
PLUGINS USED (PROJECT-WIDE)
----------------------------------------------------------------
- VST3: MEqualizer (MeldaProduction) (instances: 18)
    - 07. Snare AUX
    - 08. Hats_M60
    - 15. Bass Amp_TF17
    - 16. Bass DI_TDA-1
    - 17. GTR Amp_M80
```

- project-wide plugin inventory
- master track details
- all track details
- FX chain details
- parameter captures
- routing summary

What the receipt is for

It is a readable session-state printout. It is not the final DAAP package.

21. On-Screen Messages and What They Mean

Message	Meaning
ReaImGui is not available.	ReaImGui is missing. Install it with ReaPack and relaunch the script.
Snapshot refreshed @ ...	The session was rescanned and the display now reflects the newest captured state.
Export failed: ...	The utility could not write the receipt file, usually because of a path or write-permission problem.
Manifest receipt saved to:	The receipt export completed successfully and the message box shows the saved file location.
(analytics not available - refresh snapshot)	Analytics were not available in the current cache. Press Refresh Snapshot.
(none detected)	No FX were detected in that chain display.
(none)	No entries were present for that section.

22. Current Limitations

> **WARNING.** This manual should not be used to describe the downstream packaging tool. That application belongs to a separate workflow and separate documentation path.

This version is intentionally narrow.

Important current limits

- display-only design
- no automatic packaging
- no external file reconciliation
- no standards binding yet
- parameter capture capped for readability
- track rendering capped for UI safety
- send capture capped for UI safety
- vendor and type identification are heuristic, not final database-backed classifications

23. Recommended Workflow

For REAPER users, the best current workflow is:

1. Open and finalize the session you want to inspect.
2. Launch the in-session manifest utility.
3. Review the project identity block.
4. Press Refresh Snapshot if you made any recent session changes.
5. Expand Analytics for a project-wide overview.
6. Expand Plugins Used (Project) to inspect processor usage.
7. Expand MASTER (Main Out) if you want to inspect final bus processing.
8. Expand Tracks and inspect any track of interest.
9. Use the Filter (track/fx) field to narrow the visible list.
10. Export a receipt if you want a text record of the current session state.

24. Support Notes for Future Releases

This application is expected to move into a more formal plugin implementation later, but the current Lua version already establishes the intended session-capture behavior inside REAPER.

The future packaging-side workflow is separate from this manual and belongs to the external utility path, not this in-session REAPER tool.

Section II: DAAP Package Prep Utility Module

User Manual for the Drag-and-Drop Packaging Preparation Utility

Application Entry Point
DAAP_Package_Prep_v1.py

Current framework: Python + PySide6
Primary operator environment: desktop Python application
Repository access: controlled / invite-only technical distribution
Intended operators: advanced technical users working with DAW project folders and DAAP packaging flows

Prepared for advanced operators who need to validate a finished DAW project folder, build a filesystem-authoritative Tier-1 manifest, review session-to-asset discrepancies, and then proceed into authority package preparation.

This manual describes the packaging-prep utility only. It is the external drag-and-drop layer that follows the in-session Lua capture utility and is intended for technical professionals working inside controlled repository workflows.

Application File Set

Role	Codebase file name(s)
Primary application file	DAAP_Package_Prep_v1.py
Main UI controller	ui/Package_main_window.py (flat export: Package_main_window.py)
UI support widgets	ui/Package_analysis_progress.py; ui/Package_confirm_project.py; ui/Package_manifest_preview.py; ui/Package_resolution_preview.py (flat exports of the same names)
Validation and project detection	core/Package_validator.py; core/Package_project_detect.py (flat exports: Package_validator.py; Package_project_detect.py)
Tier-1 filesystem manifest builder	core/Package_manifest_model.py (flat export: Package_manifest_model.py)
Tier-2 session mapping modules	core/tier2/reaper/tier2_controller.py; core/tier2/reaper/rpp_reader.py (documented in the uploaded coding notes; Tier2Controller is also imported by Package_main_window.py)
Stage-3 handoff called by Package Prep	DAAP_Package_Authority_Integrator_v1.py

GitHub Quick Start

This utility is not positioned as a consumer-facing application. It is intended for advanced operators working from the repository with a configured Python environment and the expected package layout.

NOTE The uploaded files provided the screen flow and core logic, but the entrypoint and window imports assume a repository package structure such as ui/, core/, and tier2 modules. Follow the repository layout and README exactly when setting up the application.

WARNING The current code depends on additional modules not included in this upload, including the Tier-2 controller path and the authority integrator. The application should be installed from the full repository, not reconstructed from only the visible flat files.

Quick Start Steps

1. Clone the invite-only repository or download the approved project bundle from the controlled access source: https://github.com/Nquist-LLC/Digital-Audio-Assets-Protocol-DAAP---Packaging-Preparation-Utility
2. Create and activate a Python virtual environment.
3. Install the Python dependencies required by the repo, including PySide6.
4. Confirm that the repository package layout matches the import paths used by the application entry point.
5. Launch the utility with the repository entry script and drag a finished project folder into the application window.
6. Review the manifest and resolution screens before proceeding into authority packaging.

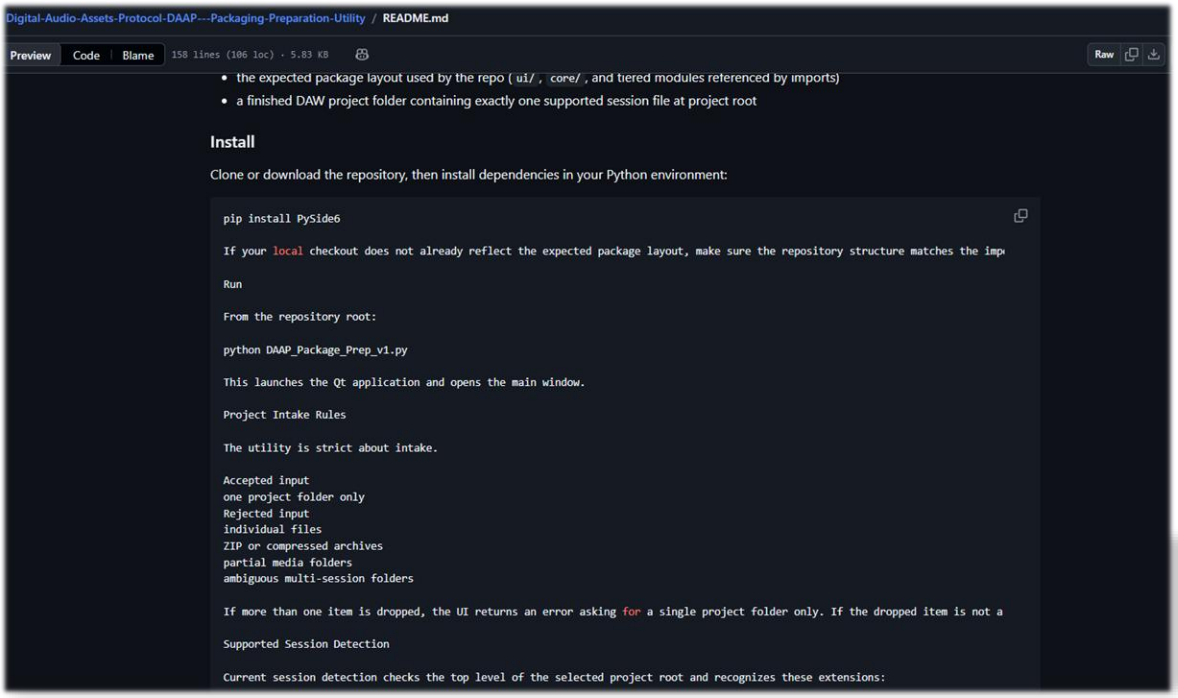


Figure 1. illustration for the controlled repository quick start.

Table of Contents

1. About This Application

The DAAP Package Utility is the external packaging-preparation surface in the DAAP workflow. It is a desktop Python application that boots a Qt application and opens the Package Prep main window, which acts as the first interaction point for declaring a finished project folder for Package preparation.

Unlike the in-session Lua utility, which captures the state of a live REAPER session, Package Prep works at the project-folder level. It is designed to accept a completed DAW project directory, validate it, build a Tier-1 filesystem-authoritative manifest, hand that approved manifest into Tier-2 parsing, and then proceed into authority packaging.

NOTE The code positions this utility as a strict UI/controller layer. Filesystem inspection and manifest generation are delegated to the validation and manifest-model modules rather than performed directly by the window code.

2. Intended Operator and Access Model

This utility is intended for advanced operators, technical delivery specialists, archival engineers, and other professionals who can work from controlled repository instructions. It is not written as a simplified consumer Packageing application. The operational assumption is that qualified operators will install the application from an invite-only repository, configure the environment correctly, and understand how to review discrepancies before proceeding.

In practical terms, that means the application is suitable for knowledgeable technical professionals working inside the music-industry digital distribution and preparation layer, rather than for casual end users.

WARNING Because the application flow depends on repo structure and additional modules, operators should use the repository distribution and official setup instructions rather than copying isolated source files into a local folder and attempting to run them ad hoc.

3. What the Utility Does

• Accepts a finished project folder by drag and drop into the application window.

• Validates that the dropped item is a single project folder and that exactly one supported DAW session file exists at the project root.

• Builds a Tier-1 manifest directly from media found on disk rather than from DAW parsing.

• Displays a read-only manifest preview showing detected assets, media filenames, and manifest state.

• Runs Tier-2 session parsing after manifest approval in order to compare session references against the project manifest.

• Displays a resolution screen summarizing unresolved session references and unmapped assets.

• Requires explicit acknowledgment before continuing into Package preparation.

• Calls the authority packaging step and reports the resulting bundle root, manifest store path, and receipt path when packaging completes successfully.

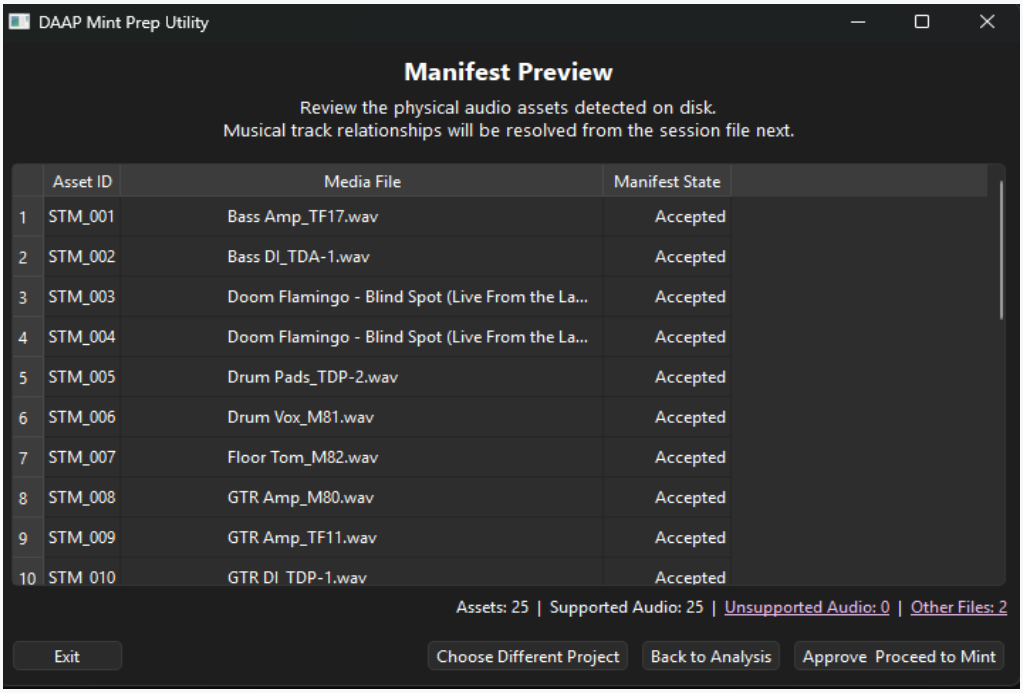


Figure 2. Recommended overall flow illustration: Drop -> Confirm -> Analyze -> Manifest Preview -> Resolution Preview -> Package Preparation.

4. What the Utility Does Not Do

This utility is not a DAW, not a session editor, and not a casual one-click release uploader. It does not mutate project files during the confirmation or analysis stages, and the analysis progress screen explicitly states that no files are being modified during inspection. The uploaded code also shows read-only preview screens for manifest and resolution review rather than editable views.

It is also important to distinguish Package Prep from the later authority layer. The utility prepares and assembles application state so the authority package can be built, but it is not presented as an open-ended editing environment for ad hoc manipulation of the manifest.

5. System Requirements

At minimum, operators should expect the following requirements:

- Python 3.11 or newer recommended
- PySide6
- A full repository checkout that includes the expected ui/ and core/ packages
- Tier-2 controller modules
- The authority integrator module used at the Package-preparation step
- Desktop environment capable of running a Qt application

> **NOTE** The entry script imports PackagePrepMainWindow from ui.Package_main_window, and the main window imports core modules and the authority integrator. That means the installed project must preserve the package layout expected by the repository.

6. Python Installation and Environment Setup

The utility is written in Python and uses PySide6 for the user interface. A clean virtual environment is recommended for all installations.

Suggested installation sequence

1. Install a supported version of Python on the target machine.
2. Clone the controlled repository to a local working directory.
3. Create a virtual environment in the repository root.
4. Activate the virtual environment.
5. Install the repository requirements, including PySide6 and any Tier-2 or authority-packaging dependencies listed by the repo.
6. Verify that the repo includes the expected package directories and __init__.py files so Python can resolve the imports used by the application.
7. Run the application from the repository root or from the startup command documented by the repo.

Recommended replacement

Run the application from the repository root.
Create and activate a Python virtual environment, install the required dependencies, then start the application entry file.

A standard setup would be:

macOS / Linux

```
python3 -m venv .venv
source .venv/bin/activate
pip install PySide6
python DAAP_Package_Prep_v1.py
```

Windows PowerShell

```
py -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install PySide6
python DAAP_Package_Prep_v1.py
```

Windows Command Prompt

```
py -m venv .venv
.\.venv\Scripts\activate.bat
pip install PySide6
python DAAP_Package_Prep_v1.py
```

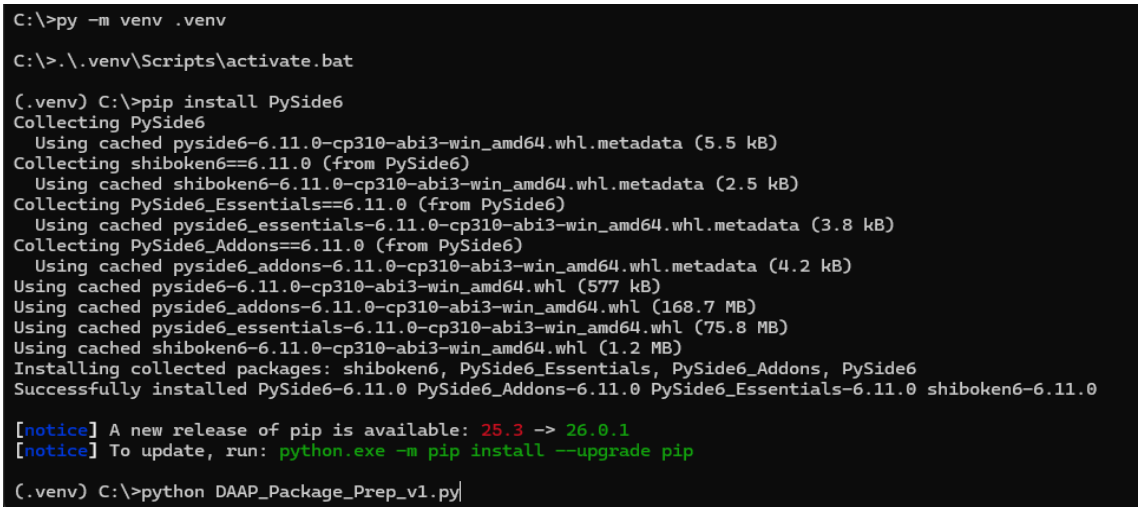


Figure 3. Python environment setup screenshot.

7. Repository Layout and Launching the Application

The application bootstrap file is DAAP_Package_Prep_v1.py. Its purpose is intentionally minimal: it instantiates QApplication, creates the main window, shows it, and enters the Qt event loop. The file is described as a UI bootstrapper with no validation or business logic of its own.

Because the entry point imports from ui.Package_main_window, operators should run the application from an environment in which the repository package layout is intact. A typical launch command is the direct execution of the entry script from the project root.

```
python DAAP_Package_Prep_v1.py
```

8. Main Workflow Overview

The code flow is sequential and strongly gated. Each screen exists to establish one layer of trustworthy state before the next layer proceeds.

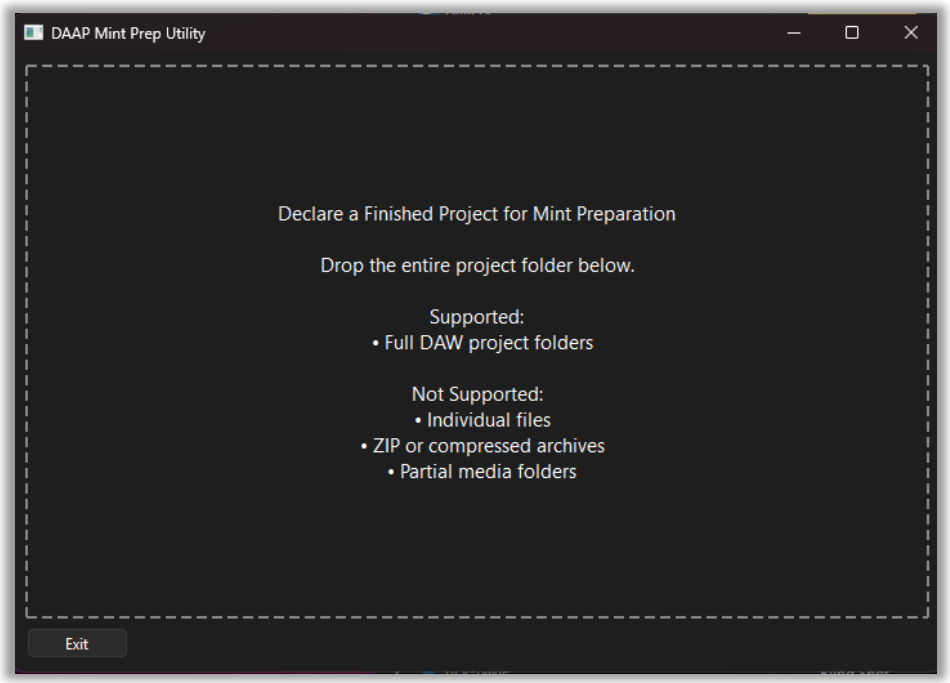
- Project Declaration: the operator drops a single project folder into the application.
- Validation: the app verifies folder-only input and checks for exactly one supported DAW session file.
- Confirm Project: the operator reviews project name, session file, and path before analysis begins.
- Analysis Progress: the application shows a read-only progress view and log output while analysis runs.
- Manifest Preview: the Tier-1 filesystem manifest is shown for review.
- Resolution Preview: Tier-2 discrepancy summaries are shown and must be acknowledged.
- Package Preparation: authority packaging is attempted using the approved application state.
- Reset: after success or certain cancel paths, the application returns to the drop screen for the next run.

9. Project Declaration Screen

The first visible screen is the Project Declaration view. The central label instructs the user to declare a finished project for Package preparation by dropping the entire project folder into the application window. The screen explicitly states what is supported and what is not supported.

Supported input is described as full DAW project folders. Unsupported input includes individual files, ZIP or compressed archives, and partial media folders. The application window also includes a global Exit action and an Exit button in the bottom action row.

Figure 5. Project Declaration screen.



10. Validation Rules and Error Messages

Once a drop occurs, the main window enforces a single-item drop rule. If more than one item is dropped, the application raises an Invalid Selection error telling the user to drop a single project folder only.

The validation module then enforces folder-only input, detects DAW session files at the project root, and rejects ambiguous multi-session folders. The detection module currently looks only at top-level files and recognizes .rpp, .ptx, and .logicx as supported session-file extensions.

- Invalid Selection - Package Prep requires a project folder, not individual files.
- No DAW Session Detected - No supported DAW session file was found in this folder.
- Multiple Sessions Found - Only one session file is allowed per project folder.

NOTE Current session detection is top-level only. Recursive discovery or DAW-specific logic is not yet part of the uploaded detector module.

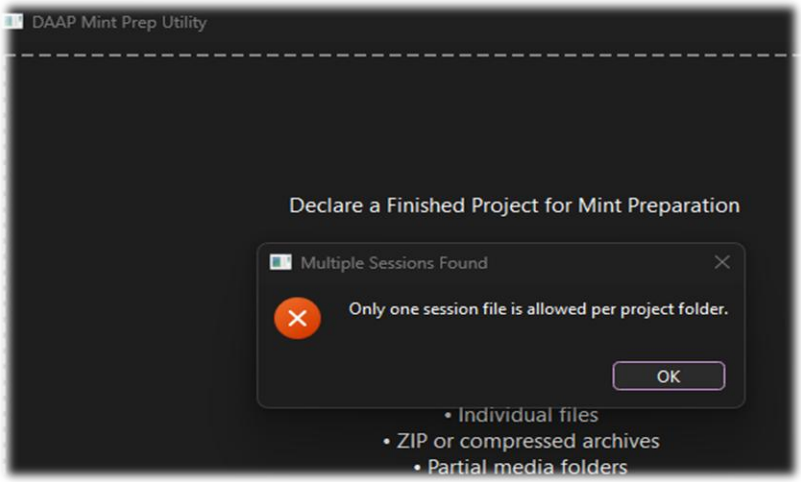


Figure 6. Validation error examples.

11. Confirm Project Screen

If validation succeeds, the application stores project_name, session_file, and project_path in project_context and transitions to the Confirm Project screen. This screen is a read-only scope-lock checkpoint before analysis begins.

The screen shows the project name, session file, and full project path. It also states that confirming will lock the project scope for analysis and that no files will be modified. The two action buttons are Confirm & Continue and Cancel / Choose Different Project.

Choosing Cancel returns the application to the initial drop screen. Choosing Confirm & Continue moves the operator into the analysis phase.

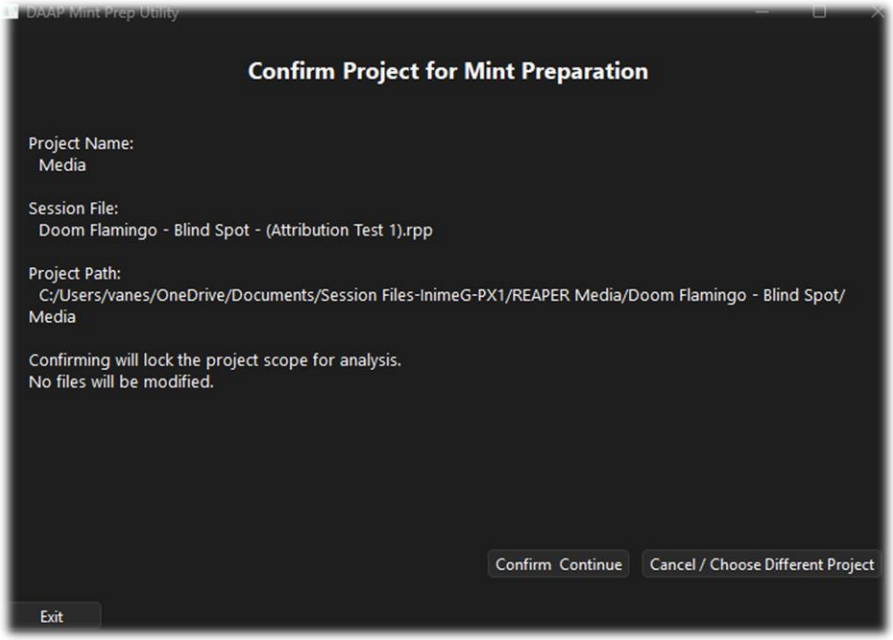


Figure 7. Confirm Project screen.

12. Analysis Progress Screen

The Analysis Progress screen is the active execution boundary between confirmation and manifest preview. It displays a header reading Analyzing Project, a subheader stating that the system is inspecting session structure and validating media, and an indeterminate progress bar.

A read-only log window beneath the progress bar shows textual progress. In the current uploaded code, the analysis logic is still a staged stub. The log sequence includes Starting analysis..., the project name, the session file, Scanning project structure..., Enumerating tracks..., Validating media references..., Preparing manifest structure..., and finally Analysis complete.

A Cancel button is available through the surrounding action row. Cancelling marks the analysis as cancelled and returns the operator to the drop screen.

WARNING Because the analysis widget is currently stubbed in the uploaded code, the manual should describe the existing screen honestly: it presents explicit progress and logging, but the real analysis logic is intended to be injected incrementally.

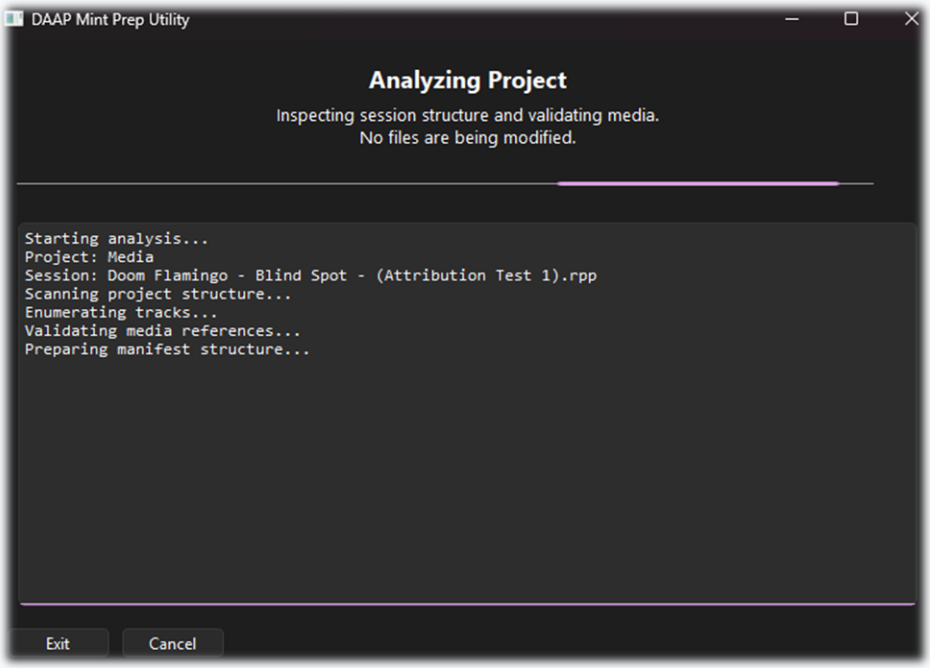


Figure 8. Analysis Progress screen

13. Manifest Preview Screen

After analysis completes, the window builds a filesystem manifest and opens the Manifest Preview screen. This is the final verification checkpoint for the Tier-1 view of assets on disk.

The main table contains three columns: Asset ID, Media File, and Manifest State. Media filenames are shown with tooltips, and the Manifest State column is currently set to Accepted for each listed supported asset. The column tooltip clarifies that this state reflects filesystem inspection only and that session usage is evaluated later.

The header text beneath the title explains the workflow clearly: the operator is reviewing the physical audio assets detected on disk, and musical track relationships will be resolved from the session file next. If the manifest summary contains a blocking_issue, that message is shown prominently and the Approve & Proceed to Package button is disabled.

The screen also presents summary counters for total assets, supported audio, unsupported audio, and other files. Unsupported Audio and Other Files are clickable labels that open informational dialogs.

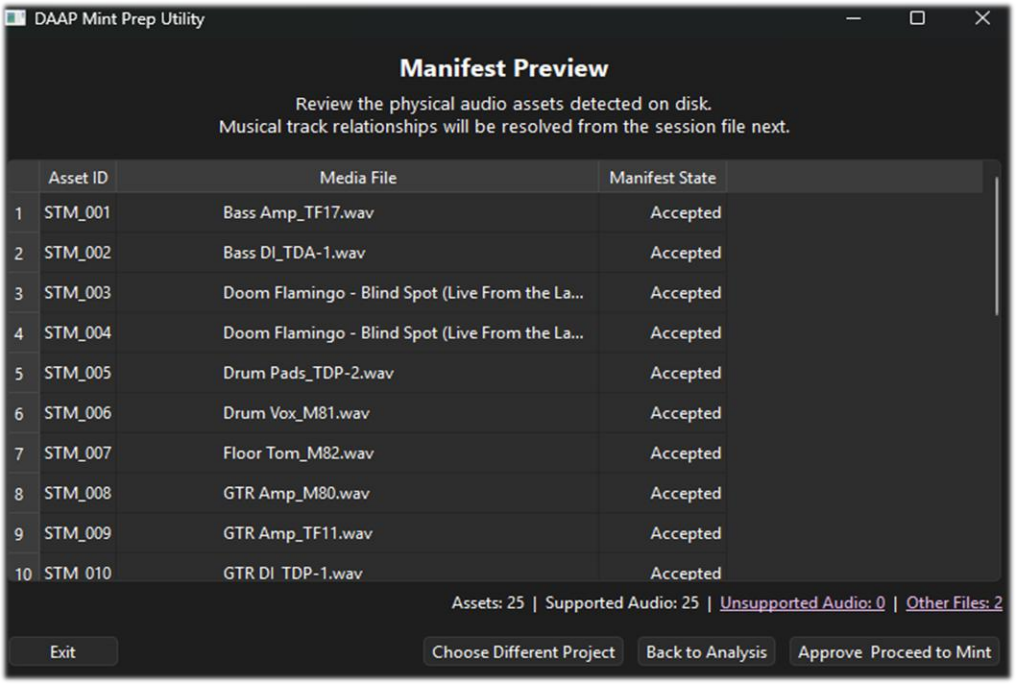


Figure 9. Manifest Preview screen.

14. Tier-1 Filesystem Manifest Logic

The Tier-1 manifest model is intentionally filesystem-authoritative. It walks the project root, excludes peaks and .reapeaks directories, and enumerates files found on disk. Supported audio formats are .wav, .aif, .aiff, and .flac. Unsupported audio formats are .mp3, .ogg, and .m4a.

Each supported audio file is assigned a sequential track_id in the form STM_001, STM_002, and so on. The manifest summary stores counts for supported audio, unsupported audio, non-audio files, and total warnings. If no supported audio is detected, a blocking_issue is added stating that no supported audio files were detected in the selected directory.

NOTE Tier-1 does not rely on DAW session parsing. That is a deliberate trust boundary in the uploaded code: every audio file on disk is represented first, and session-aware enrichment happens later in Tier-2.

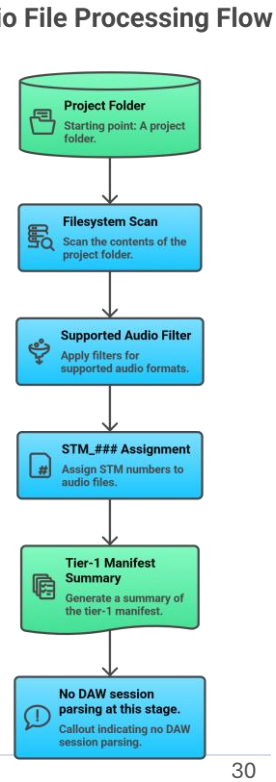


Figure 10. Tier-1 manifest logic diagram or sample table.

15. Unsupported Audio and Other Files Dialogs

If the operator clicks the Unsupported Audio summary link, the application opens an Unsupported Audio Files dialog. That dialog lists each detected unsupported audio file and also lists the accepted formats used for Packaging. If the operator clicks the Other Files link, the app opens a dialog listing non-audio files that will not be included, along with a recommendation to consider moving them before Packaging.

These dialogs are informational. They do not mutate the manifest or remove files automatically.

16. Resolution Preview Screen

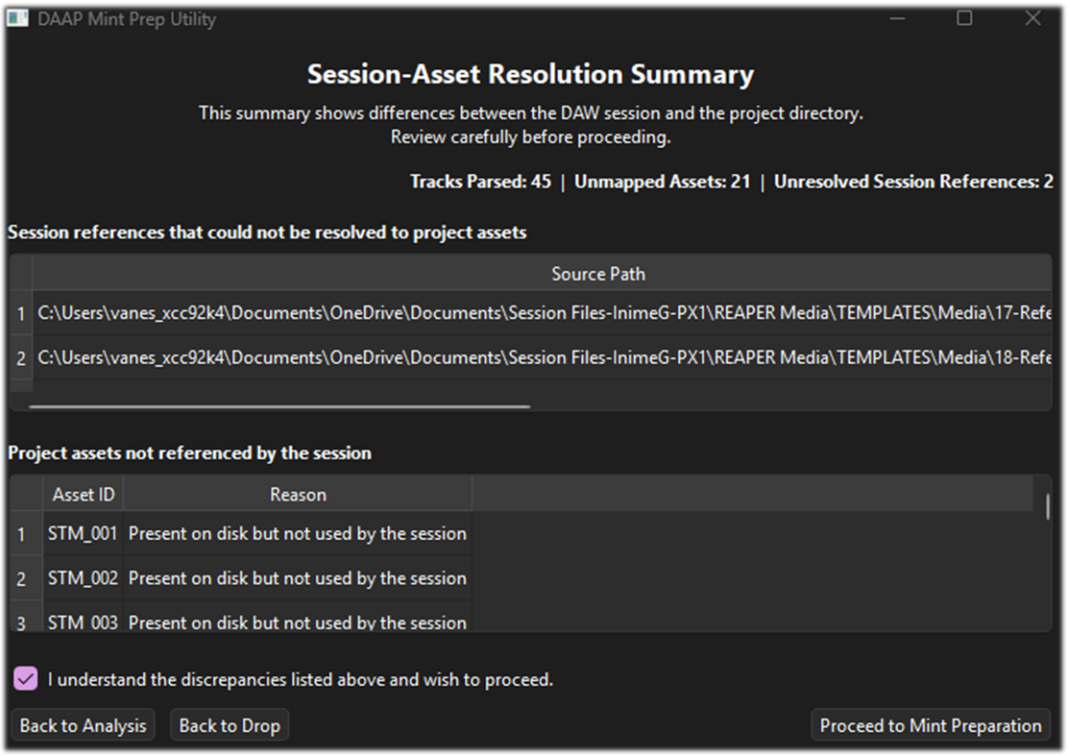
When the operator approves the manifest, the application runs Tier-2 processing. The main window calls the Tier-2 controller using the approved manifest and stored project context, executes Phase 2.1 and Phase 2.2, and then opens the Session-Asset Resolution Summary screen.

This screen is a read-only discrepancy review checkpoint. At the top it shows summary counters for Tracks Parsed, Unmapped Assets, and Unresolved Session References. It then displays two read-only tables: one for session references that could not be resolved to project assets, and one for project assets not referenced by the session.

Each table includes a human-readable Reason column. Unresolved reasons may include Referenced by session but not found in project manifest, Absolute path is outside project root, Multiple assets share this

filename, or Path is missing or malformed. Unmapped reasons may include Present on disk but not used by the session or Present on disk but excluded by Tier-1 rules.

At the bottom, the operator must check an acknowledgment box stating: I understand the discrepancies listed above and wish to proceed. The Proceed to Package Preparation button stays disabled until this checkbox is checked. The other actions are Back to Analysis and Back to Drop.



> **WARNING** This acknowledgment screen is an important accountability step. It records that the system surfaced discrepancies and that the operator intentionally chose to proceed.

17. Proceed to Package Preparation / Authority Packaging

The Proceed to Package Preparation action performs defensive state validation before authority packaging begins. The main window checks that project_context, approved_manifest, session_skeleton, and session_map all exist. If any are missing, the application raises a Package Preparation Error explaining which required state is missing and instructs the operator to restart the Package-prep flow.

If the required state is present, the application calls build_authority_package using the already-approved Tier-1 and Tier-2 state. On success, the system displays a Package Preparation Complete dialog showing the bundle root, manifest store path, and receipt path. After the operator clicks OK, the application resets its state and returns to the project declaration screen.

If authority packaging fails, the application raises a Package Preparation Error dialog containing the failure message returned by the exception.

18. Exit, Cancel, and Reset Paths

The application includes both local cancel paths and a global exit action. The Exit action is bound to Ctrl+Q and also appears as a bottom-row button on the declaration and analysis container views. If the operator tries to exit, the application displays an Exit DAAP Package Prep confirmation dialog warning that any current analysis progress will be lost.

Back to Analysis on the resolution screen clears the session_skeleton and session_map and returns to the analysis phase. Back to Drop clears the stored state and returns to the initial project selection screen. Cancel during project confirmation or analysis also resets state and returns to the declaration view.

19. Supported Formats and Current Limits

Based on the uploaded code, the current supported session-file extensions are REAPER (.rpp), Pro Tools (.ptx), and Logic Pro (.logicx), with detection limited to the project root. The current Tier-1 supported audio formats are WAV, AIF, AIFF, and FLAC, while MP3, OGG, and M4A are counted as unsupported audio for Packaging.

The codebase also reveals several present-day limitations that should be stated plainly in the manual:

- The analysis progress screen is still a staged stub in the uploaded code.
- Recursive session-file detection is not yet implemented in the detector module.
- The application depends on repo modules not fully represented in this upload, including Tier-2 internals and the authority integrator.
- Manifest state values such as Missing and Changed are noted as future possibilities rather than active states in the current preview table.

20. Recommended Operator Workflow

1. Prepare a finished project folder so that the project root contains exactly one supported DAW session file.
2. Launch the utility from the repository environment.
3. Drop the full project folder into the Project Declaration screen.
4. Review the validation result and confirm the project scope.
5. Allow the analysis screen to complete.
6. Review the Tier-1 manifest carefully, including unsupported audio and non-audio file dialogs if present.
7. Approve the manifest only after the Tier-1 view is acceptable.
8. Review the resolution screen, paying attention to unresolved references and unmapped assets.
9. Check the acknowledgment box only if the discrepancy state is understood and acceptable.
10. Proceed to Package Preparation and record the returned bundle, manifest store, and receipt paths for downstream use.

> **NOTE** For high-trust workflows, operators should retain the success dialog information and any generated receipt alongside other authority-package records.

Section III: DAAP Package Authority Integrator

User Manual

Manifest-Store Authority Assembly and Bundle Preparation

Application File	DAAP_Package_Authority_Integrator_v1.py
Role in Pipeline	Stage 3 of the DAAP asset-preparation workflow
Environment	Python authority-layer module invoked after package-prep discrepancy review
Current Status	Draft operator-facing implementation
Intended Audience	Advanced technical operators, studio systems personnel, distribution engineers, and protocol implementers

Prepared as a companion manual to the In-Session Manifest Utility and the DAAP Package Prep Utility.
This manual explains how the authority-layer module turns studio-validated project state into a DAAP authority object and bundle.

Application File Set

Role	Codebase file name(s)
Primary application file	DAAP_Package_Authority_Integrator_v1.py
Imported Tier-1 manifest dependency	Package_manifest_model.py (or core/Package_manifest_model.py in package-layout form)
Imported manifest-store / bundle builder dependency	daap_manifest_draft_v1.py (or core/daap_manifest_draft_v1.py in package-layout form)
Additional local code files directly imported by the current uploaded integrator	No other project-local Python modules are directly imported by the current uploaded authority integrator.

GitHub Quick Start

This repository is not a casual end-user utility. It is the advanced authority-layer bridge between studio-validated project state and a DAAP authority object.

The normal path is not to run this file first. The normal path is:
1) review the session inside the In-Session Manifest Utility,
2) validate and reconcile the project in the Package Prep Utility, and then
3) call the authority integrator only after discrepancies have been reviewed and acknowledged.

WARNING — Do not treat this module as a one-click Packageing shortcut. It assumes that earlier truth boundaries have already been reviewed and committed, and it should be used by technically competent operators.

Recommended first-run sequence: https://github.com/Nquist-LLC/Digital-Audio-Assets-Protocol-DAAP---Packaging-Authority-Integrator

- Create or activate a Python virtual environment.
- Install the required Python packages, including PySide6 and the optional `jcs` package if standards-compliant canonicalization testing is desired.
- Place `DAAP_Package_Authority_Integrator_v1.py` so it can import both the Package Prep components and `daap_manifest_draft_v1.py`.
- Run the Package Prep flow first; let the UI call the authority integrator after discrepancy acknowledgment.
- Use the CLI demo only for controlled testing after editing the placeholder `project_context`.

```
python -m venv .venv
# Windows PowerShell
.venv\Scripts\Activate.ps1
# macOS / Linux
source .venv/bin/activate
pip install PySide6 jcs
```

Figure 0. Repository entry points and recommended operator order

Table of Contents

1. About This Module

The DAAP Package Authority Integrator is the authority-layer Python module that blends the existing Package Prep package-prep flow with the newer DAAP manifest-store authority model. In the file header, it defines itself as the bridge between two worlds: the Package Prep side, which already builds a filesystem-authoritative view of the project and resolves DAW session structure against physical assets, and the protocol-side draft builder, which adds canonical manifest authority, payload content bindings, manifest hash generation, disclosure and integrity blocks, and DAAP bundle packaging.

In practical terms, this module is where studio-approved, externally verified project state becomes a transportable DAAP authority object. It is the point where the system stops being merely an inspection or reconciliation workflow and becomes a delivery-ready digital asset model.

Figure 2. Authority integrator position between package prep and bundle output

2. Why This Is a Separate Module and Repository

This layer is intentionally separate because it is not doing the same job as either of the first two utilities. The In-Session Manifest Utility is concerned with what the session is while it is still open inside the DAW. The Package Prep Utility is concerned with whether the project folder, session file, and physical media on disk line up cleanly enough to proceed. The authority integrator is concerned with something else entirely: turning approved upstream truth into a bound authority object that can leave the studio working environment and move into broader digital delivery infrastructure.

Keeping this module separate protects the system in several ways. It isolates the most protocol-sensitive logic from the session-inspection surface. It prevents packaging and manifest-store authority mechanics from being conflated with basic UI review tasks. It also reflects the reality that this stage is more specialized, more standards-sensitive, and more consequential than earlier stages. This is the layer where binding rules, manifest-store structure, disclosure tiering, content hashes, and future signing policy begin to matter.

NOTE — Separating this repo does not mean the other stages are unimportant. It means this stage carries a different kind of responsibility: it creates the authority object that will represent the asset outside the studio.

There is also a governance reason for separation. Earlier tools can be used for review, preparation, and internal validation. This layer sits closer to the formal asset-delivery pipeline and therefore benefits from tighter operator control, narrower audience expectations, and independent versioning while standards, trust models, and distribution-side acceptance patterns continue to evolve.

3. Three-Stage Digital Audio Assets Protocol Pipeline

The authority integrator only makes sense when it is understood as the third stage in a controlled sequence. The first stage captures truth inside the studio session. The second stage verifies what is actually present on disk and how the session file relates to those assets. The third stage, described in this manual, binds those verified results into an authority object that can leave the studio environment and travel into external distribution systems.

Stage	Module	Environment	Truth Boundary	Primary Output
1	In-Session Manifest Utility	Open DAW session	Live structural session review	Reviewed in-session snapshot and receipt
2	Package Prep Utility	Project folder + session file	Filesystem truth + session reconciliation	Approved Tier-1 manifest and Tier-2 session map
3	Authority Integrator	Authority-layer packaging module	Manifest-store authority + payload bindings	DAAP bundle, manifest store, exports, and receipt

Figure 1. Three-stage DAAP flow from in-session capture to authority bundle output

4. What the Authority Integrator Does

At a high level, the module performs seven jobs.

First, it enriches Tier-1 filesystem truth so that payload preparation carries stronger identity surfaces such as stable asset IDs, relative paths, absolute paths, media-type hints, and format hints.

Second, it converts those enriched Tier-1 records into authority-facing payload specifications. This is the point where the earlier disk inventory begins to take the shape required by the DAAP manifest-store model.

Third, it builds a base authority manifest using the draft builder. At this stage the asset receives a base identity block, disclosure tier, payload bindings, and party references.

Fourth, it integrates session-aware information from Tier-2 when available so that the bundle preserves not only raw files on disk but also session-file order, track structure, media-item references, and discrepancy diagnostics.

Fifth, it optionally integrates exported information from the REAPER Lua in-session utility so that higher disclosure tiers and audit views can preserve additional synopsis detail such as FX identities and routing summaries without turning the system into DAW replay.

Sixth, it allocates the integrity surfaces: payload content bindings, manifest hash generation, signature-envelope metadata, timestamp placeholders, and logical manifest-store construction.

Seventh, it writes the final output surfaces: the DAAP bundle itself, readable JSON exports for development and troubleshooting, a human-readable packaging receipt, and an optional sidecar manifest-store copy for bridge testing.

- Collect Tier-1 audio assets from the project folder using stable relative paths.
- Accept Tier-2 session outputs when available.
- Optionally accept a Lua snapshot export for richer synopsis data.
- Build a DAAP authority manifest and manifest store.
- Package the result into the normative DAAP bundle layout.
- Emit readable side artifacts for audit and troubleshooting.

5. What the Authority Integrator Does Not Do

This module deliberately stops short of several things.

It does not mutate the user's DAW session. It does not re-run DSP. It does not attempt DAW playback or state replay. It does not adjudicate rights. And in its current draft state, it does not yet perform real PKI signing, OCSP revocation checking, or TSA-backed timestamp issuance.

Those boundaries matter because this layer is about preserving identity and integrity semantics around a finished asset, not about recreating the studio environment or pretending that the legal and trust stack is already finished.

WARNING — Do not describe this module as if it already performs final certificate issuance, revocation, or trusted timestamp proofing. The current draft allocates those hooks and data shapes, but the full trust infrastructure is intentionally deferred.

6. Intended Operator and Access Model

This module is written for knowledgeable technical operators. In your broader system framing, that means infrastructure-minded studio personnel, advanced packaging operators, archival or distribution engineers, and other technical participants who understand why the authority object matters and how to interpret discrepancy results.

That is also why this layer is a poor fit for a mass-market or casual "upload and Package" model. It does not reward uninformed speed. It rewards disciplined preparation, careful state review, and respect for the fact that once the asset leaves the studio, it must carry enough structure to survive ingest, transformation, and verification across external systems.

7. System Requirements

At minimum, the authority integrator needs a Python environment with access to the Package Prep components and the draft DAAP manifest-store builder.

Practical requirements include:
- Python 3.10 or later recommended
- `PySide6` if the module is being invoked through the Package Prep UI flow
- `jcs` recommended if you want standards-compliant RFC 8785 canonicalization rather than the draft deterministic JSON fallback
- Access to the Package Prep support modules such as `Package_manifest_model.py`
- Access to `daap_manifest_draft_v1.py`, which supplies payload spec types, manifest assembly helpers, manifest-store serialization, and bundle creation
- A validated project folder, with earlier-stage state already approved

NOTE — The file supports both flat-file and package-layout imports. That means it can work beside the draft builder in a simple experimental layout or inside the intended `core/` package structure when the repository is organized more formally.

8. Installation and Repository Layout

The authority integrator is best treated as part of a controlled Python repo rather than a loose script dropped into arbitrary directories. Even so, the file intentionally includes import fallbacks for both a flat-file layout and the intended package layout.

For a clean development environment, create a virtual environment, install the required packages, and make sure the authority integrator can import both the filesystem manifest model and the draft authority builder.

```
python -m venv .venv
# Windows PowerShell
.venv\Scripts\Activate.ps1
# macOS / Linux
source .venv/bin/activate
pip install PySide6 jcs
```

A practical repository arrangement is:
- UI/controller layer in the Package Prep application
- Tier-1 filesystem manifest model in the core package
- Tier-2 session parser/controller in the core package
- `DAAP_Package_Authority_Integrator_v1.py` as the authority bridge layer
- `daap_manifest_draft_v1.py` as the protocol-side draft builder

Figure 3. Recommended repository structure for the authority integrator and its upstream modules

9. Inputs Required by the Integrator

The high-level entrypoint is build_authority_package(...). In normal operation, it is called **after** the Package Prep utility has already completed project declaration, validation, Tier-1 manifest review, Tier-2 session mapping, and discrepancy acknowledgment.

Input	Required	Purpose
project_context	Yes	Project name, session file, and project path used as the root authority context.
tier1_manifest	No	Previously built filesystem-authoritative manifest. If omitted, the integrator can rebuild or enrich it.
session_skeleton	No	Tier-2 abstract session structure used to preserve track order and session identity.
session_map	No	Tier-2 session-to-asset mapping used to bind media-item references to payloads and preserve diagnostics.
lua_snapshot	No	Optional export from the in-session Lua utility for richer synopsis data such as FX identities and routing summaries.

Input	Required	Purpose
output_root	No	Override for where the DAAP bundle and working exports will be written.
asset_version	No	Version label for the authority asset state.
disclosure_tier	No	Current default is `T1_INDUSTRY`.
create_sidecar_copy	No	Whether to emit an additional sidecar manifest-store copy for bridge/profile testing.

Figure 4. Suggested call-site placement after Tier-2 discrepancy acknowledgment

Normal packaging-flow inputs

Input	Purpose
project_context	Required root context for the project name, session file, and project path.
tier1_manifest	Filesystem-authoritative asset set already approved in Package Prep.
session_skeleton	Tier-2 session structure used to preserve session identity and track order.
session_map	Tier-2 item-to-asset mapping used to bind session references to payloads and preserve discrepancy diagnostics.

Optional enrichment input

lua_snapshot Optional export from the in-session Lua utility for additional track-level tool and routing enrichment.

Optional packaging / policy controls

Input	Purpose
output_root	Overrides the default export location for bundle and working files.
asset_version	Sets the authority asset version label.
disclosure_tier	Sets the disclosure tier for the manifest. Default is T1_INDUSTRY.
create_sidecar_copy	Controls whether an additional sidecar manifest-store copy is emitted for bridge/profile testing.

Why Several Inputs are Optional in Code

The integrator is written as a reusable boundary, not only as a single UI call. Some inputs are optional because the module can rebuild Tier-1 internally, while others exist to support enrichment, policy control, or bridge testing. In the normal packaging flow, however, the Package Prep utility already provides the Tier-1 and Tier-2 state before authority packaging is called.

10. Normal Operating Flow

The authority integrator is not meant to be the first thing a user touches. Its best-fit system location is after Tier-2 discrepancy review and before final Package/export.

The intended operational sequence is:
1) The operator reviews the project inside the In-Session Manifest Utility while the DAW session is still open.
2) The operator moves to the Package Prep Utility, drops the finished project folder, validates it, confirms scope, reviews Tier-1 assets, and acknowledges Tier-2 discrepancies.
3) Only after those earlier truth boundaries are accepted does the system call `build_authority_package(...)`.

In the current UI code, this call belongs after the resolution preview acknowledgment, not inside the lower-level widgets and not earlier in the flow. That placement is deliberate: by the time Stage 3 begins, the earlier stages have already done their jobs and their outputs should be consumed immutably.

NOTE — The design intent is explicit: keep `_on_analysis_complete()` as the point where Tier-1 truth is committed, keep `_on_manifest_approved()` as the point where Tier-2 mapping runs, and call the authority integrator only after discrepancy acknowledgment.

Figure 5. Suggested controller handoff from Resolution Preview to `build_authority_package(...)`

11. Internal Systems Breakdown

11.1 Tier-1 Collection Enrichment

The existing Tier-1 builder is intentionally simple and trustworthy, but it only carries filename-level references. The authority integrator enriches that surface by collecting stable asset IDs, relative paths, absolute paths, media-type hints, and format hints.

This is important because the authority layer needs stronger payload identity surfaces than a shallow inventory ledger. At the same time, it preserves the discipline that Tier-1 remains filesystem-authoritative. The code explicitly states that this enrichment does not infer musical role or song order.

11.2 PayloadSpec Conversion

Once the enriched Tier-1 manifest exists, the module converts those records into the payload-spec form used by the authority builder. This is where the earlier notion of a track inventory becomes an authority-facing payload-binding surface.

Each payload spec carries a payload ID, a path, a role, a locator, a media type, a format hint, and references such as relative path and Tier-1 track ID.

11.3 Base Authority Manifest Assembly

The module then builds a base authority manifest. This is the moment where the asset receives a base identity block, disclosure tier, asset type, mappings namespace, extension namespace, and default party references. Until dedicated contributor or rights ingestion exists, the module at least identifies Nquist as originator and packager.

11.4 Tier-2 Session Graph Integration

Tier-2 is where the system stops looking like a directory printer and starts looking like a session-aware asset system. When `session_skeleton` and `session_map` are available, the authority integrator adds session nodes, track nodes, media-item nodes, and payload-binding edges into the synopsis.

This preserves session file identity, actual musical track order from the DAW session, item/source-to-asset mapping, and discrepancy diagnostics. Just as importantly, the code is explicit about what it does not preserve here: executable DSP state, DAW edit replay semantics, clip-lane detail, comp-lane detail, and automation specifics.

11.5 Optional Lua Snapshot Enrichment

The REAPER in-session Lua utility has visibility that the external folder packager does not, especially live FX chain identities, routing summaries, and per-track status at the moment of approved review. The authority integrator treats this as optional enrichment rather than a packaging prerequisite.

That is a crucial design choice. It lets the system enrich the synopsis for higher disclosure tiers and audit surfaces without making the external package-prep flow dependent on REAPER internals.

11.6 Integrity Placeholders and Manifest Store

After payload bindings and synopsis integration are complete, the module allocates the integrity block. It adds a signature-envelope metadata shape, adds a timestamp placeholder, finalizes the manifest hash, and then builds the logical manifest store.

The manifest store deliberately separates claims and integrity while keeping them bound. The active manifest remains the authority object, while content bindings, signatures, and timestamps remain the verification machinery around that authority object.

11.7 Bundle Construction and Sidecar Export

Finally, the integrator writes the outputs. It creates a clean exports root, writes readable JSON exports, writes a long packaging receipt, serializes the manifest store, builds the DAAP bundle layout, and optionally writes a sidecar manifest-store copy for bridge/profile testing.

At this point the system has crossed the boundary from verified studio state to a transportable DAAP authority package.

12. Hash Protections and Binding Logic

This is the section that matters most for understanding why Stage 3 is so important.

The authority integrator is not just copying files into a folder tree. It is binding payloads to an authority object with explicit cryptographic surfaces. The payload-binding helper in the draft builder requires every referenced payload to carry at least one cryptographic binding, with SHA-256 as the minimum baseline. The current draft implementation also emits a parallel SHA-384 digest so the manifest already has the multi-hash shape contemplated by the technical paper.

This matters because once the asset leaves the studio, path strings and filenames are not enough. The file can be copied, mirrored, relocated, or delivered in sidecar form. The authority model therefore treats paths as transport hints only. Identity is determined by digest agreement, not by path trust.

- Payload bytes are hashed before they are treated as authority-bound payloads.

- The payload list is stored in the manifest as content bindings rather than mere file references.
- The manifest itself is hashed after it has been populated and finalized.
- The manifest store carries both the active manifest and the integrity block that binds manifest state and payload state together.

WARNING — The current draft implements real payload digests and manifest hashing, but its signing and timestamp layers are still placeholder surfaces. Do not confuse cryptographic binding of payloads with full trust-chain completion.

Figure 6. Payload binding concept: transport path as hint, digest as authority

13. Canonicalization, Signatures, and Timestamp Status

The authority integrator is aligned to the technical paper's direction on canonicalization and signing, but the current implementation is explicit about what is complete and what is not.

The manifest's integrity block declares canonicalization metadata. Where the `jcs` package is available, the draft builder identifies RFC 8785 JCS as the intended canonicalization anchor. Where `jcs` is not available, it falls back to a deterministic JSON draft mode. That fallback is useful for development, but it is not the final standards-equivalent answer.

Likewise, the module allocates a baseline signature-envelope shape using the `x509_ecdsa_p256_sha256` profile name, but the signature field itself remains unpopulated in this draft layer. The timestamp surface is also reserved: the module can allocate timestamp placeholders and carry them into the manifest store, but TSA integration is not yet implemented, so the default timestamp state remains absent.

- Canonicalization intent: RFC 8785 JCS when the required package is available.
- Fallback mode exists for draft work and development testing.
- Signature envelope metadata is allocated now so downstream key-management decisions can plug into the same authority shape later.
- Timestamp placeholders preserve the structure needed for later RFC 3161-style or similar evidentiary integration.

Figure 7. Integrity block lifecycle: canonicalization, manifest hash, signature envelope, timestamp placeholder

14. Output Files and Bundle Layout

By default, the module writes a dedicated exports root beneath the project path. Inside that root it creates a DAAP bundle and a working exports area.

The current layout is designed so operators have both the normative transport object and a readable development/audit surface during this still-settling stage of the pipeline.

```
DAAP_EXPORTS/
  DAAP_BUNDLE/
    manifest_store/
      daap.manifeststore
    payloads/
      audio/
      aux/
    metadata/
```

```
    working_exports/
      daap_manifest.pretty.json
      tier1_manifest.json
      tier2_session_skeleton.json
      tier2_session_map.json
      <project>.daap.manifeststore    # optional sidecar copy
      daap_packaging_receipt.txt
```

The bundle is the transport object. The working exports folder is the operator and development surface. The readable receipt is particularly important because it summarizes Tier-1 filesystem truth, Tier-2 session mapping, unmapped assets, unresolved references, and the DAAP authority summary in a format that can be reviewed without opening JSON files.

Figure 8. Default output tree for the DAAP authority exports directory

15. Messages, Error Conditions, and Operator Meaning

Because this module is usually called from the Package Prep UI flow, operators will most often experience its outcomes through dialogs and controller-level messages rather than through a standalone window.

The main error conditions are defensive-state checks. If `project_context`, `approved_manifest`, `session_skeleton`, or `session_map` are missing when the operator tries to proceed, the UI correctly stops the flow and reports a Package Preparation Error. This prevents Stage 3 from being invoked without the approved outputs of Stage 1 and Stage 2.

If the authority builder itself cannot import the draft manifest builder, it raises a runtime error telling the operator that `daap_manifest_draft_v1.py` could not be imported and that the file should be placed beside the draft builder or the package path should be fixed.

When the build succeeds, the UI reports Package Preparation Complete and shows the bundle root, manifest-store path, and receipt path.

Message / Condition	Meaning
Project context is missing	Stage 3 was invoked before the project declaration / approval flow completed properly.
Approved manifest is missing	Tier-1 truth was not committed or carried forward into the authority call.
Session skeleton is missing	Tier-2 Phase 2.1 did not complete before Stage 3 was invoked.
Session map is missing	Tier-2 Phase 2.2 did not complete before Stage 3 was invoked.
Draft builder import failure	The authority layer cannot assemble the manifest-store object because the protocol-side builder is not available on the expected path.
Package Preparation Complete	The bundle and support artifacts were created successfully and the operator may now review or hand off the result.

16. Why This Is the Most Important Boundary

This layer is the most important part of the pipeline because it is the point at which the asset actually leaves the studio working environment as a bound, authority-bearing object.

Before this stage, the system still exists primarily as a working session, a verified folder, and a resolved mapping between session references and files. After this stage, those earlier truths are no longer merely internal facts. They become an explicit manifest-store authority model, with payload bindings, a manifest hash, disclosure policy, synopsis structure, and a transportable bundle layout.

That is why Stage 3 deserves its own module, its own manual, and its own operator expectations. It is the point where the pipeline stops being about local confidence and starts being about portable authority.

This is especially important because digital distribution systems are still regulating, negotiating, and settling on standards around provenance, disclosure, manifest stores, trust models, and verification policy. By isolating the authority layer, the system can evolve these protocol details without destabilizing the earlier capture and prep layers that studio users rely on every day.

NOTE — If Stage 1 explains the session and Stage 2 proves the files, Stage 3 binds those truths into the thing that can survive outside the studio.

17. Current Limitations

This module is intentionally strong on structure and intentionally unfinished on the final trust stack.

Current limitations include:
- real PKI signing is not yet applied
- OCSP / CRL revocation checking is not yet implemented in this layer
- TSA integration is not yet implemented
- contributor, label, and rights ingestion are still minimal
- the Lua snapshot is optional enrichment rather than a full formal upstream handshake
- some bundle and manifest-store behavior remains in draft form while standards and policy choices continue to settle

WARNING — The authority integrator already matters operationally because it fixes the structure of the authority object, but it should still be described as a draft operator layer rather than a final globally trusted verifier.

18. Recommended Workflow

For disciplined use, the recommended workflow is:
1) review the live session in the In-Session Manifest Utility;
2) validate the project folder and session/media relationships in the Package Prep Utility;
3) review Tier-1 and Tier-2 output carefully;
4) acknowledge discrepancies only when they are understood;
5) call the authority integrator;
6) review the generated receipt, manifest JSON, and bundle layout before any external handoff.

- Keep Tier-1 outputs immutable once approved.
- Do not bypass the resolution step simply because the bundle can technically be created.
- Prefer passing the Lua snapshot when a richer synopsis or audit view is warranted.

- Treat the generated receipt and working exports as operator evidence, not as the final authority surface.

19. Future Direction

The future direction of this layer is clear even though implementation is still draft.

It is expected to move toward stronger canonicalization guarantees, real certificate-based signing, trust-anchor policy, revocation handling, timestamp evidence, richer contributor and rights references, and cleaner integration with the broader DAAP manifest-store model described in the technical paper.

Even in its current form, though, the authority integrator already fixes the most important architectural move: authority is localized in the manifest store rather than in the waveform container, and payload identity is determined by hashes rather than by path trust.

Figure 9. Long-term evolution from draft authority bridge to formal DAAP delivery pipeline