

CombiningMinds

MIGRATION GUIDE

Tana to Obsidian

A practical guide to moving your notes

Dario da Silva

combiningminds.org

Contents

Read this first . . . 4
- How this guide was made . . . 4
- Quick start . . . 4

Start here . . . 5

1. Decide what you want to bring across . . . 6
- What you need . . . 6

2. Export and preserve the source . . . 7
- Start with Markdown . . . 7
- Keep JSON for a different purpose . . . 7
- Treat attachments as a separate job . . . 7
- Keep the working areas distinct . . . 7

3. Read the export before designing the converter . . . 8
- Build a representative sample . . . 8
- Account for things you do not intend to import . . . 9

4. Choose a small destination structure . . . 10
- Decide when to aggregate . . . 10

5. Preserve identity before rewriting links . . . 11
- Link syntax is a choice; correct targets are not . . . 11

6. Translate fields into a modest property scheme . . . 12
- Protect dates and status meanings . . . 12

7. Handle journals, tasks and related material . . . 13
- Daily notes should retain their context . . . 13
- Route tasks by evidence . . . 13
- Folder nodes and related notes need review . . . 14

8. Specify a conversion you can inspect . . . 15
- Run each test into fresh output . . . 15

9. Review the result in an empty vault . . . 16
- Mechanical checks . . . 16
- Read the notes as a person . . . 16
- Decide whether to continue . . . 16

10. Bring reviewed content into your everyday vault . . . 17
- Rebuild conveniences afterwards . . . 17

11. Your completion checklist . . . 18

Conclusion . . . 19

Appendix Optional mapping examples . . . 20
- Property names can be mapped explicitly . . . 20
- A legacy reading status needs an explicit flat mapping . . . 20
- Filename changes need a complete map . . . 20

Common mistakes to avoid 20

Read this first

How this guide was made

I wrote this guide with AI, from my own migration. I gave it my notes and my scripts, then read and revised the drafts many times. The experience in it is mine. Much of the wording is the AI's.

That shows in the length. AI is thorough, so some sections go further than most people need. The checklists and acceptance criteria are an example. Use them if they help. They are not a test you have to pass.

Everything here is a guideline, not a rule. This guide does not try to copy my workflow, or anyone else's. Take what fits your notes and leave the rest.

The scripts are the same. They come from my own migration, for you or an AI assistant to read and adapt. They are not a converter, and they will not match your workspace without changes.

Quick start

If you read one page, make it this one.

1. Export your Tana workspace as Markdown, and keep the ZIP untouched.
2. Extract a working copy of the export. Do every test on the copy.
3. Create an empty Obsidian vault for testing.
4. Convert a small sample: a book, a project, a meeting, a day of notes, and the odd cases you worry about.
5. Open the sample in Obsidian and use it the way you use your notes every day.
6. Fix what is wrong, then convert the sample again into a fresh folder.
7. When the sample looks right, convert the rest, check it, and move your daily work across.

Use the chapters when you hit something specific, such as fields, tasks, links or attachments. The contents page takes you there.

Start here

This guide grew out of a real migration: exporting a Tana workspace, working out where its different kinds of content belonged, then using a custom conversion process to prepare an Obsidian vault. The useful part was not a perfect folder structure. It was the cycle of inspecting, making a small set of decisions, checking the result and adjusting those decisions.

The approach here keeps that cycle. You will export and preserve your source, design a small test vault, specify the conversion, review the output and only then consider bringing it into your everyday vault.

This guide takes you from an untouched export to an Obsidian vault you can use with confidence. Work through the chapters in order, prove each transformation on a small sample, then expand to the full export. A small migration can be done manually; the same decisions and checks apply.

Treat historical scripts as unverified reference patterns for an AI assistant or developer to inspect and adapt, not tools to run blindly. Check their assumptions, paths and transformation rules against your own source. Keep independent backups and untouched originals, and test any adapted code on disposable copies before using it for a migration.

1. Decide what you want to bring across

You are moving information, not reproducing Tana inside another application. A useful outcome is a vault in which you can find your notes, follow their relationships, open their attachments and understand the work they record.

Start with three questions:

- Which content do you still need to use: active projects, reference notes, daily writing, people or tasks?
- Which context makes that content useful: dates, parent projects, status, source links or discussion history?
- Which workflows will need rebuilding rather than importing: saved searches, dashboards, automations or specialist views?

A book can become a note with properties. A project can become a note with related material and a task list. Those are design choices, not automatic equivalents of Tana supertags and fields. Storing a supertag name as an Obsidian property preserves a label; it does not recreate the behaviour associated with that supertag.

Do not redesign your whole knowledge system before you begin. Choose enough structure to test a book, a project, a meeting, a day of notes and a handful of awkward cases. The awkward cases tell you more than a folder full of straightforward notes.

What you need

Prepare access to the Tana workspace, an export location, a separate working area and an empty Obsidian test vault. If you use an AI coding assistant, give it only the folders and information it needs. The original workflow used Claude Cowork; the method does not depend on a particular assistant.

Check your permission to copy shared or organisational content before exporting or handing it to another service. A folder called `Work` is an organisational choice, not an access-control boundary. If content must remain separate, settle that boundary before conversion.

Keep a short migration record with the export date, included workspaces, chosen destinations, unresolved questions and each test run. This will be more useful than trying to remember why a particular rule exists.

2. Export and preserve the source

Start with Markdown

In the workspace home node, open the ellipsis menu and choose **Export workspace as Markdown**. The command line also offers **Export workspace in markdown/JSON**. Let the export finish before closing or reloading Tana. The result is a ZIP containing Markdown files; calendar structure and links between exported notes are represented in that output. These instructions were checked on 20 October 2026. See Tana's workspace export instructions.

Keep the downloaded ZIP untouched. Extract a working copy into a separate folder and inspect it before changing anything. If you have several workspaces, label their exports clearly; do not mix their contents into one directory and hope matching filenames refer to the same thing.

Keep JSON for a different purpose

Tana describes Markdown as the export for browsing or moving to other tools, and JSON as the technical export containing fuller metadata. Keeping a JSON export alongside Markdown can support later investigation, but it is not interchangeable with a Markdown conversion. See Tana's export formats.

Tana also says users cannot directly import that raw JSON back into Tana. Do not treat downloading it as proof of a tested restore route. See Tana's JSON export guidance.

Treat attachments as a separate job

The Markdown ZIP does **not** include media files. Images and videos generally have links, while some other attachments may have no Markdown link and need saving from Tana. Tana's current help page also offers a helper for downloading linked Tana-hosted media. It edits extracted Markdown, does not cover every external host, and leaves failed downloads unchanged. Review it before use, work on a disposable copy and inspect its results. See Tana's attachment export guidance.

Make an attachment checklist rather than assuming that a visible image means a local copy exists. Record the source note, original reference, intended local filename and recovery status. Check PDFs and other documents as well as pictures. Two attachments called `notes.pdf` need distinct destinations if they are different files.

Once you have local assets, test them from the test vault with the network disconnected where practical. A note that still loads a remote image has not demonstrated that its image was preserved locally. Keep unresolved items visible in your migration record; do not delete the source workspace while recovery is incomplete.

Keep the working areas distinct

A simple arrangement is enough:

```
Migration/
  Originals/        Untouched exports
  Working/          Extracted copies
  Test-vault/       Disposable converted output
  Reports/          Mapping and review records
```

These are example names, not required application folders. Put the originals somewhere the converter will not write. Keep a separate backup of anything you cannot afford to lose. Synchronisation is not a substitute for backup, as Obsidian's backup guidance explains.

3. Read the export before designing the converter

The source migration included daily notes arranged by year, week and day, a library of tagged content, an inbox, schema material and saved searches. Some items were single files; others were folders with a `README.md` and child files. Your export may differ.

Open examples from several parts of your workspace. Check how the export expresses titles, fields, nested content, references and dates. Write down what you actually find rather than making the old export's conventions into universal rules.

These fictional filenames illustrate patterns found in a Markdown export:

```
A useful book(book).md
Follow up with Example Person(task).md
Planning meeting(meeting)/
```

The suffix suggested a type, but parentheses can also be part of an ordinary title. A filename parser should not strip every final parenthesised phrase without evidence. Check names with multiple tags, punctuation, missing titles and unusual characters.

A fictional representation of the legacy export's field pattern is shown below. The slash-based status is legacy input to be replaced through a reviewed mapping, not a recommended destination value:

```
- **Producer**: [Example Author](<../Example Author.md>)
- **Status**: to/read
- **Rating**: 4
```

That pattern is useful input for a conversion rule. It is not permission to turn every bold bullet into metadata. A note might contain a quoted example or prose using exactly the same formatting.

Build a representative sample

Include at least one example of each common type, plus:

- Two different nodes with the same display name.
- A referenced node that appears in more than one place.
- A folder-based node with meaningful child content.
- A daily note containing both plain writing and tagged items.
- Open, completed and cancelled tasks, if you use those states.
- A linked attachment, an unusual filename and a field with several values.
- An item whose type or parent is unclear.

Keep this sample as a repeatable test set. Add a difficult example whenever you discover a new failure. If a transformation only works on the simplest book note, it is not ready for the whole workspace.

Account for things you do not intend to import

Schema, option lists and saved searches may not belong in your everyday vault, but inspect them before excluding them. They can explain what a field means or preserve the definition of a workflow you want to rebuild.

Do not discard all numbered types, status nodes or deleted material by rule alone. Decide what you want retained, archived separately or excluded, and record the reason. An exclusion is an explicit decision, not the same as an unrecognised item disappearing.

4. Choose a small destination structure

The original migration used separate destinations for books, videos, projects, people, journals and short lists. That distinction is useful; the full hierarchy is optional.

Start with something like this:

```
Test-vault/
  Content/
  Projects/
  People/
  Meetings/
  Journals/
  Lists/
  Attachments/
  Unsorted/
```

Add `Content/Books` or other subdivisions only when they help you navigate. Do not create dozens of empty categories just because the source had dozens of supertags.

Write a routing plan in plain language. For example:

- Books become individual notes in `Content/Books`.
- Projects become individual notes in `Projects`.
- Meetings keep their date, attendees and project relationship.
- People retain discussion points that have a verified relationship to them.
- Short ideas can become entries in `Lists/Ideas`.
- Anything without a reliable rule stays in `Unsorted` for review.

An `Unsorted` folder is a safety mechanism, not a failure. It gives ambiguous content somewhere visible to wait while you decide.

Decide when to aggregate

A one-line idea may work well in a list. A long idea with its own references, attachments or discussion may deserve a separate note. Set a rule based on the content, not only its tag.

Aggregation changes the address of an item. If another note points to an idea that becomes a list entry, the conversion needs an appropriate destination within that list, or a retained standalone note. Do not quietly send every old link to the top of a large list and call the relationships preserved.

Apply the same caution when combining a review with a book or a project chunk with its parent. Merge only after establishing the relationship. Similar wording is a review clue, not proof that two records belong together.

5. Preserve identity before rewriting links

Display names are for people; they are not dependable unique identifiers. Two projects can both be called “Website”, and two people can share a name. A converter that matches solely by title can produce links that open successfully but lead to the wrong note.

Build a source-to-destination map before rewriting references. Prefer stable source IDs where they are available and understood. Otherwise retain an unambiguous source path within the preserved export, together with the workspace/export identity. A path-based key identifies that snapshot; it does not prove continuity across later exports.

For each source object, record its source key, original title, destination and disposition: standalone note, aggregated entry, archived, excluded by decision or unresolved. If the export contains reference comments, preserve them until you know whether they are needed for this mapping.

When names collide, choose distinct destinations deliberately. For example, two fictional projects might become `Projects/Website redesign` and `Projects/Website archive`. If a suffix derived from a source key is needed, check that it is unique. Neither a short suffix nor a truncated title guarantees uniqueness by itself.

Link syntax is a choice; correct targets are not

Obsidian supports both Markdown links and wikilinks. Conversion to wikilinks is therefore optional. Paths can identify notes in particular folders, and a wikilink can use a readable display label. See Obsidian’s internal link documentation.

For example, a resolved destination might be written as:

```
[[Projects/Website redesign|Website]]
```

Do not obtain that result by copying the source display text into double brackets. First resolve the original target, find its mapped destination, then write the new link. Keep external web links distinct from internal note references and attachment references.

Changing folder depth can break a previously valid relative link. Renaming files, shortening filenames and merging notes all require the mapping to be updated. Test links after those transformations, not only before them.

Record unresolved and ambiguous links separately. An automated check can find missing targets; human review is still needed to catch a link to the wrong existing note.

6. Translate fields into a modest property scheme

Use properties for information you will actually search, sort or maintain. Leave longer explanations and complex nested material in the note body unless you have a clear reason to transform them.

Obsidian stores properties as YAML at the beginning of a note. Each property key must be unique within that note, and internal links in property values need quoting. Its property types include text, lists, dates and numbers. See Obsidian's properties documentation.

This fictional book note maps the legacy `to/read` value to the flat status `unread`. Define a consistent set of flat status values before converting your own notes:

```
---
type: book
status: unread
producer:
  - "[[People/Example Author]]"
rating: 4
---

# A useful book

## Notes
- A question to revisit after the next chapter.
```

Keep your choices consistent. If `status` is a single value, do not make it a list in half the notes. If a field has several people, preserve them rather than silently choosing the first. Retain the original value when its meaning is unclear and flag it for review.

A property called `type: book`, a link to `[[Books]]` and a tag such as `#book` serve different purposes. Choose deliberately rather than treating them as interchangeable. There is no need to add all three.

Avoid empty placeholder fields everywhere. They make a vault look structured without preserving more information. A manually maintained property called `backlinks` is also not a substitute for checking the links that actually exist in the notes.

Protect dates and status meanings

Use an explicit full date when the source provides one. Do not guess a missing year or invent a completion date from the export date. If you create weekly summaries, document how you assign weeks, especially around the year boundary.

Keep "cancelled" distinct from "completed". If you choose to display cancelled tasks with a checked box, preserve the cancellation label so a later review does not count them as finished work.

Before replacing field bullets with properties, verify that the values survived. Keep any unmapped fields in a clearly labelled body section until they have a destination.

7. Handle journals, tasks and related material

Daily notes should retain their context

The source workflow separated tagged items from ordinary journal writing. A meeting could become its own note while the day retained a link to it. That makes the meeting easier to find without erasing when it happened.

For example, `Journals/2026-10-12.md` might contain:

```
# 2026-10-12

- Reading notes: Read a chapter and captured a question.
- [[Meetings/2026-10-12 Planning meeting]]
- Followed up on the reading list.
```

When you extract an item, leave enough context to understand the day. Do not remove surrounding remarks just because they lack a recognised type. If the same node occurs in a library and a journal, use its identity to distinguish one object with references from two genuinely separate objects.

Route tasks by evidence

The original plan used three destinations: project tasks in the relevant project, completed unlinked tasks in weekly records, and open unlinked tasks in a review list. This remains a useful starting point.

Use explicit project relationships where available. When a parent is uncertain, retain the task in `Lists/Open tasks` with its source context rather than guessing. A fuzzy title match may suggest a candidate to review; it should not silently assign ownership.

An illustrative project could contain:

```
---
type: project
status: active
---

# Reading room

## Next steps
- [ ] Compare two shelving layouts.

## Tasks
- [x] Measure the available wall.
- [ ] Check the delivery dimensions.
```

Preserve task text, status and relevant dates before adding conveniences. A clean-looking list is not useful if parent context or unfinished work has vanished.

Folder nodes and related notes need review

A folder containing a `README.md` and children might become one note with sections, or an index plus several notes. Read the children before choosing. A long meeting transcript should not be flattened into an unreadable bullet under a person merely because there is a relationship.

For book reviews, discussion points and project chunks, document what gets combined and where. Retain links to separately kept material. Never let a merge rule overwrite an existing section without a reviewable record of the change.

8. Specify a conversion you can inspect

An AI assistant can help explore the export and build a converter, but it needs a specification rather than a broad instruction to “clean everything up”. Ask for an inventory and proposed rules first. Review those before authorising output generation.

A useful brief is:

> Inspect only the working copy. Do not modify originals or my live vault. Report the structures, field patterns, source identities and unresolved cases you find. Propose a source-to-destination map and a small representative test. Preserve unknown content. Do not merge by title alone, silently replace undecodable text or overwrite existing files. Before running anything, show the proposed inputs, output location and expected changes.

For the implementation, ask for separate stages: reading, identity mapping, transformation, link rewriting and reporting. This makes it easier to see whether a missing note was never read, was misrouted or was written under an unexpected name.

The report should distinguish source objects from output files. Ten source ideas deliberately combined into one list are not nine missing files. Conversely, a plausible file count does not establish that every source object survived.

Run each test into fresh output

Use a new empty output folder for each test iteration. Keep human review notes outside generated files so they are not lost when output is regenerated. Do not assume a converter is safe to rerun over existing notes just because it ran once.

Have the script stop and report decoding failures. Replacing unreadable characters silently can destroy names and break matching. Inspect the source and choose any alternative decoding deliberately.

Filename limits also need checking on the systems you use. Blind truncation can make two different titles identical. Keep full titles in the content or mapping, choose safe filenames and test collision handling.

Do not remove reference comments, navigation fragments or apparent boilerplate until you have established that they are no longer carrying information needed by the conversion.

9. Review the result in an empty vault

Review in two passes: mechanical checks, then actual use. Neither replaces the other.

Mechanical checks

Ask for a report covering:

- Source objects accounted for by destination or explicit decision.
- Unrouted objects, parsing failures and unmapped field values.
- Filename collisions and ambiguous relationships.
- Missing or unresolved internal targets.
- Attachments recovered, still remote or missing.
- Duplicate property keys and invalid property formatting.
- Tasks grouped by original and resulting status.

“Zero errors” is only meaningful if you know which errors were checked. Keep the report with the exact input snapshot and rule version that produced it.

Read the notes as a person

Open each sample beside its source. Check nested bullets, significant wording, field values and dates. Follow links from a journal to a meeting, from the meeting to a project and from that project to an attachment.

Inspect both same-name examples. Check that they remain distinct and that their inbound links lead to the right place. Find an aggregated idea and verify that its source context is still recoverable.

Search for a phrase you remember from an awkward note. Review open tasks and cancellations. Open a few recovered attachments directly, not just their filenames in a directory listing.

Finally, use the test vault for a small real activity: locate a reference, review a project or find the notes from a day. This reveals awkward organisation that syntax checks cannot catch.

Decide whether to continue

Expand to a full export only when the sample behaves as intended. After the full conversion, repeat the reports and sample review, adding examples from categories not covered earlier.

Unresolved cases need either a fix or an explicit retention decision. They do not all have to be redesigned now, but none should vanish because the rest of the vault looks tidy.

10. Bring reviewed content into your everyday vault

Treat this as a separate operation from conversion. Preserve the approved staging output and back up the destination vault before adding anything.

Set a cutover point before the final addition. Either pause edits in Tana from the final export onwards, or separately capture and review every post-export change, including deletions and task-status changes. Reconcile those changes with the staged output before relying on it. Record when Obsidian becomes the authoritative place for new edits; avoid continuing to update both copies independently.

First compare the proposed destinations with existing notes and attachments. The same filename can mean either duplicate content or a different item. A content match can also be misleading if dates, properties or relationships differ.

For each conflict, decide whether to keep both, rename the imported note, combine reviewed content or skip a verified duplicate. Record the decision and update links accordingly. Do not run a broad merge command against your live vault on the strength of a successful test conversion.

A copy of the destination vault makes a useful rehearsal space. Test the addition there, review the change list and check existing links as well as imported ones. Only then apply the approved changes to the everyday vault.

If you use sync, check its status and avoid simultaneous edits during this step. Confirm the imported content and attachments on each device you rely on after synchronisation. Local success alone does not establish that another device has everything.

Keep rollback simple: retain the pre-import backup, imported-file list and record of any changed existing files. Avoid making unrelated reorganisations at the same time.

Rebuild conveniences afterwards

Start by reading and editing the migrated notes without adding a large plugin stack. Identify a specific missing workflow before choosing a tool to replace it.

A task overview, book list or journal navigation scheme can be a later improvement. None is evidence that the migration itself is complete. Keep saved-search definitions as reference where useful, then rebuild only the searches you still need.

11. Your completion checklist

You are ready to rely on the migrated material when:

- The untouched exports and an independent backup are retained.
- Source objects have destinations or recorded retention/exclusion decisions.
- Important links reach the intended notes, including same-name cases.
- Fields, dates, task states and journal context have been checked.
- Required attachments are locally available and open correctly.
- Unresolved items are visible, understood and retained safely.
- Post-export changes have been reconciled and the cutover point is recorded.
- The destination addition has been rehearsed and reviewed.
- You can find and use the material in ordinary work.

Keep Tana available while you resolve missing content and establish confidence in the new vault.

Conclusion

A successful export, a successful script run and a successful migration are three different things. The useful result is a collection of notes you can find, understand and keep working with.

Start with an untouched export and a representative sample. Preserve identity and context, make deliberate choices about fields and destinations, and account for unresolved material. Bring the result into your everyday vault only after testing the workflows you rely on.

Keep the source and rollback copies while you settle into Obsidian. You do not need to recreate every dashboard or perfect every folder before you can use your notes. Once the content is safe and usable, improve the organisation through ordinary work.

Appendix Optional mapping examples

- These fictional examples show how to make mapping decisions. Inspect your source, define a mapping and test it on a disposable copy before applying it to the full collection.
- Keep independent backups and unchanged originals. Unknown content and ambiguous values need review, not a convenient default.

Property names can be mapped explicitly

Legacy field	Illustrative destination choice	Review condition
`input`	`type`	Only if it really describes the kind of note
`instance-of`	`output`	Only if this retains the source field's meaning
`date_finished`	`date-finished`	Preserve the actual supplied date; do not invent one
`recommended_by`	`recommended-by`	Keep all intended references and quote YAML links

The source field names and destination names need not be identical, but a rename alone does not prove that meaning survived. Do not copy an installation's property table without checking its intended searches and workflows.

A legacy reading status needs an explicit flat mapping

A legacy source `status:: to/read` can map to the flat destination `status: unread`. Choose a consistent set of flat status values for your notes and update relevant queries to match. Keep the slash-form value only in the preserved source.

Do not turn an empty status into `read`: missing data does not establish completion. Do not treat cancelled as completed. Namespace page paths and attachment links containing `/` are separate from status values.

Filename changes need a complete map

Legacy exports can encode punctuation, for example `%3F` for `?`, `%3A` for `:`, `%22` for a double quote and `%7C` for `|`. Inspect the export rather than assuming every encoded sequence must be decoded blindly.

In a fictional Windows-target example, `Example%3A Topic.md` might become `Example - Topic.md`. That is a reviewed choice, not a universal transformation. Check blank names, case-only duplicates, collisions and every incoming link before writing. Preserve the full original title in the source map where filename changes lose information.

Common mistakes to avoid

- Shortening block IDs does not guarantee uniqueness. Map full source IDs, test collisions and check that links reach the right content.
- A filename does not automatically replace a meaningful `title` property.

- Renaming `tags` to `backlinks` does not preserve tag behaviour or create actual backlinks.
- Do not reuse another installation's routing rules or default execution settings without checking them against your own source.

Use these examples to write your own mapping rules, then verify the results against the original notes.