

CombiningMinds

MIGRATION GUIDE

Logseq to Obsidian

A practical guide to moving your notes

Dario da Silva

combiningminds.org

Contents

Read this first . . . 3
- How this guide was made . . . 3
- Quick start . . . 3

Start here . . . 4

1. Decide what you are moving for . . . 5
- What to expect . . . 5

2. Preserve the graph before improving it . . . 6
- Make a useful inventory . . . 6

3. Start with a small Obsidian vault . . . 7
- Configure only what you need . . . 7

4. Make a conversion plan before writing code . . . 8
- A brief for assisted conversion . . . 8

5. Handle the patterns that need decisions . . . 9
- Page properties: keep meaning, not every old key . . . 9
- Structured journal blocks: extract selectively . . . 10
- Block references: map the target before replacing the syntax . . . 10
- Namespaces and filenames: decide once, then update links . . . 11
- Dates: standardise the known ones . . . 11
- Tasks and queries: preserve intent before rebuilding behaviour . . . 11
- Outlining and app-specific syntax . . . 12

6. Treat attachments as a separate migration . . . 13
- PDFs and annotations . . . 13

7. Review the output before trusting it . . . 14
- Reconcile transformations, not just totals . . . 14
- Search for leftovers . . . 14
- Acceptance checklist . . . 14

8. Switch over without losing the last week . . . 16
- Work with your notes, not against them . . . 16
- How I use Obsidian now . . . 17

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 graph without changes.

Quick start

If you read one page, make it this one.

1. Back up your complete Logseq graph, and keep that backup untouched.
2. Make a working copy of the graph. Do every test on the copy.
3. Create an empty Obsidian vault for testing.
4. Convert a small sample: some ordinary pages, a few journals, 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 block references, journals, properties or attachments. The contents page takes you there.

Start here

I migrated roughly 1,400 Logseq pages and five years of daily journals to Obsidian. I had been putting it off for ages. Logseq was central to how I worked and thought, and I knew moving would mean more than copying some Markdown files.

It took about two weeks of focused work, with most of that going into journal extraction and the awkward cases. I wrote conversion scripts with Claude's help, reran them on copies, and kept finding things I had missed. The result was worth it for me, even with the things I had to go back and fix. The scripts were written around my own graph, so they need inspecting and adapting before anyone else uses them.

This guide brings together what I learnt along the way: what to preserve, where you need to make a decision, and what to check before relying on the new vault. I can't promise that everything will move perfectly. My hope is that sharing the awkward parts will save you some of the work of discovering them yourself.

Scope: this guide covers Markdown file-based Logseq graphs. It is not a database-graph migration guide. If your source is a database version or a different export format, inspect a representative export first; do not assume its structure or identifiers match these examples. Any automation must be inspected and adapted to your own graph, then tested on a disposable copy.

You will need access to your complete graph, space for backups and working copies, Obsidian, and time to check the results. Coding is optional for a small graph. For a large or heavily structured graph, someone needs to inspect and test any conversion code you use.

If you use older migration scripts, treat them as unverified examples for an AI assistant or developer to inspect and adapt. Don't run them blindly. Check their assumptions, paths and conversion rules against your own graph, keep independent backups and untouched originals, and test adapted code on disposable copies first.

Read the guide once before changing anything. Then work through it in stages:

1. Preserve the source and inventory what matters.
2. Choose a small destination structure.
3. Convert a representative sample.
4. Resolve the difficult patterns.
5. Convert and verify the full copy.
6. Switch your daily work and retain a recovery route.

1. Decide what you are moving for

My reason for moving was partly the writing experience and partly the way I wanted to work with my notes from outside the app. I wanted a vault whose organisation I understood and could use with file-based tools, including AI-assisted workflows.

Logseq's file-based graphs already work with files that other tools can access. For me, the move was about the organisation inside those files: translating it into a shape that suited Obsidian and the way I wanted to work.

The biggest adjustment for me was moving from "everything is a block" towards "this belongs in a page, or under a heading in a page". You can keep writing bullets in Obsidian. You do not have to turn every daily note into an essay. But recreating every Logseq interaction is a poor first migration goal.

Write down three things you need to keep working. Perhaps they are daily capture, finding reading notes, and following references to earlier decisions. Use those as your checks throughout the migration. It's easy to spend time getting the folders looking right and lose sight of the workflow you actually use every morning.

It's worth asking what you're happy to leave behind, too. Do you still use that dashboard? Which queries actually help you during the week? You don't need to rebuild something just because it's there. Keep a clear plan for the things you do rely on.

What to expect

Ordinary prose and lists are the easy part. Block references, structured journal entries and renamed files need more attention. Queries, task automation and plugin-specific behaviour are separate rebuilding jobs.

Source pattern	First migration decision
Ordinary pages and bullets	Preserve content and nesting initially
Page properties	Map meaningful fields to a small schema
Block properties	Keep readable text or extract selected blocks
Block references and embeds	Map each source ID to its final destination
Namespaces	Choose folders, flat names or explicit labels
Journals	Standardise dates without guessing missing years
Images and PDFs	Preserve binaries and verify each kind of link
Queries and plugin features	Record their purpose; rebuild selectively

A small, simple graph may need little more than careful manual work. A large graph with years of structured journals deserves a staged project. The two weeks I spent should give you a sense of the work involved, but your graph may need more or less.

2. Preserve the graph before improving it

Make a dated copy of the complete Logseq graph. Include pages, journals, assets, configuration, hidden files and anything stored outside the usual folders that your notes depend on. The configuration may not belong in Obsidian, but it belongs in the archive.

Keep a second independent backup somewhere appropriate for the sensitivity of your notes. Open files from both copies to check that they are readable. If you know how to create a file inventory with sizes and hashes, keep one alongside the archive; it makes later comparisons much easier.

A synchronised folder is not enough on its own. Synchronisation can carry a mistaken change or deletion to your other devices. Keep a recoverable copy separate from the working migration. See Obsidian's backup guidance.

Use four clearly named locations. These names are illustrative:

```
Migration/
  Source archive/       untouched snapshot
  Working source/       copy used for inspection
  Test vault/           disposable conversion output
  Review records/       mappings, reports, decisions
```

Do not open the only archive as your experiment. Do not send converted output back into the source folders. And do not merge a first conversion into the Obsidian vault you already use.

If you continue writing in Logseq during the migration, record the snapshot date and keep a list of later changes. At cutover, you will need to bring those changes across deliberately. Otherwise you can finish a technically successful migration of an out-of-date graph.

Make a useful inventory

Count pages, journal files and assets, but do not stop at counts. List the patterns that make your graph yours:

- Page and block properties, including unusual or inconsistent keys.
- Block references, block embeds and page embeds.
- Namespaces and links containing encoded characters.
- Daily-note filename formats and date-link formats.
- Queries, renderer macros and org-style blocks.
- Task states, deadlines and recurring tasks.
- PDF annotations, drawings, whiteboards and flashcards.
- Plugin-generated data or files outside the graph.

Write down what you find, where it lives and what you plan to do with it. It's fine to decide to keep something as readable text for now. If you're unsure whether it matters, leave yourself a note to check rather than quietly dropping it.

Choose a sample that includes each important pattern, not just your cleanest five pages. Include an old journal, a recent journal, a deeply nested entry, a page with many references, a namespace page, a strangely named file, and both working and suspect attachments. Include the target files that those sample notes link to.

3. Start with a small Obsidian vault

Create an empty test vault in a short, stable location. Avoid rearranging your whole knowledge system while you are still proving that the content can move.

A starting structure can be this simple:

```
Test vault/
  Pages/
  Journals/
  Attachments/
  Needs review/
```

These are just example folders, rather than Obsidian defaults. Keep your existing asset folder name initially if that reduces path changes. Add specialised folders only when the sample shows a reason for them.

I started with several vaults and later brought them together. That consolidation introduced duplicate content to clean up. It was useful experience, but it is not a reason everyone should use one vault. Keep personal and work material separate where your permissions, privacy needs or practical boundaries require it.

Configure only what you need

For the sample, choose a location for new notes, a location for attachments and a daily-note filename convention. I use `YYYY-MM-DD`; the important part is that filenames and converted date links agree.

Obsidian accepts both Markdown links and wikilinks. Wikilinks can feel familiar coming from Logseq, but you do not need to convert every working Markdown link just to use Obsidian. Folder-qualified links help distinguish identically named files. Official internal-link documentation.

Keep the initial setup modest. Spend some time editing, searching, following links and capturing daily notes before adding optional plugins. If you miss moving bullets around, try an outlining plugin on the sample. Add a query replacement once you know what you want it to show and have a stable set of properties. Templates can wait until you've worked out what new notes should look like.

My own setup uses outlining tools and custom hotkeys, but you do not need my plugin list. Choose shortcuts for opening today’s note, moving list items and switching panes in the settings available on your system. Treat any particular key combination as a personal choice, not a universal default.

Leave sync out of the disposable conversion loop. Once the result is worth keeping, test your chosen sync method with a small vault and an attachment before relying on it across devices. Local availability on your computer does not establish availability on your phone.

4. Make a conversion plan before writing code

Start with a short document that maps out the changes. For each pattern in your sample, write down the source form, the intended result and the test that will prove it worked.

For example:

Pattern	Decision	Check
Reading-note page	Keep as a page; retain author and status	Text and fields agree with source
Journal meeting block	Extract only if it has useful structure	Journal links to the complete note
Namespace page	Flatten with a unique name	All old links reach that destination
Unknown property	Preserve for review	No silent deletion
Missing attachment	Report the original reference	Distinguish absent file from bad path

Make manual conversions for a few sample files before asking a tool to repeat them. That gives you an expected result to compare against. It also exposes disagreements in your plan while they are still cheap to fix.

A brief for assisted conversion

You can use an AI assistant or developer to help inspect patterns and build a converter. Give it a bounded working copy, not unrestricted access to your whole notes collection. Check whether the service and the account you use are appropriate for the information involved.

A useful brief is:

> Inspect this working copy without changing it. Inventory the file types, properties, reference formats and naming patterns. Propose a conversion plan for the supplied sample and show the expected output before writing conversion code. Preserve unknown content, record source-to-destination mappings, report decoding failures and collisions, and write only to a new output folder. Do not delete source files, overwrite existing notes or merge into my live vault. Stop for review when a rule is ambiguous.

Ask for fixtures covering nested bullets, duplicate filenames, colliding block-ID prefixes, year-less dates, moved blocks and missing assets. Ask it to show you the errors and unresolved items. You need to see what still needs attention, even if the overall conversion appears to have worked.

My original scripts assumed things about my own folders and setup. If you adapt them or use other automation, check the paths, dependencies and routing rules, then compare the result with the sample you converted by hand.

For each iteration, write into a fresh empty output folder. Do not assume rerunning a script over yesterday’s results is safe. It may duplicate extracted notes, append another index or overwrite manual corrections.

5. Handle the patterns that need decisions

Page properties: keep meaning, not every old key

Page-level properties are a manageable starting point. Decide which fields you need and what each field means before normalising names.

Here is a fictional legacy source page. The slash-based status is shown only as an input to replace through an explicit mapping:

```
status:: to/read
producer:: [[Example Author]]
rating:: 4

- A useful article about keeping a reading practice.
```

For this example, the destination uses the flat status `unread`:

```
---
status: unread
producer:
  - "[[Example Author]]"
rating: 4
---
```

This example maps legacy `to/read` to the flat status `unread`. Define your own consistent set of flat status values before conversion, and keep the legacy source unchanged.

Follow it with the original body text. YAML properties belong at the top of the note. Keys must be unique; quote internal links. Use a number for a rating and a list where multiple values are meaningful. Obsidian's property format.

Do not blindly treat every `key:: value` line anywhere in a file as page metadata. A field attached to one journal bullet may describe only that bullet. Moving it to the top of the journal changes its meaning.

Likewise, do not discard `title` merely because it looks administrative. It may preserve a display name that differs from the filename. Keep it until you have established whether it matters. Fields such as `collapsed` or query-display options can often be removed from the converted copy after you have classified them, but unknown fields should survive for review.

I found redundant classifications after my own conversion: one rule preserved a source field while another added a new field carrying the same idea. Check both duplicate YAML keys and different keys that unintentionally duplicate meaning.

Links are not tags

`[[Reading]]` is a link to a note. It is not automatically the same as `#reading` or a value in the YAML `tags` field. If your Logseq properties use page links as classifications, decide whether to keep those links, turn them into tags, or preserve both for distinct reasons.

For example, a linked `category` field can point to a topic note with its own content. A tag can provide a lightweight label. Renaming `tags` to some other property does not preserve tag behaviour by itself. Write down the mapping and test the searches you actually use.

Structured journal blocks: extract selectively

This was the most complicated part of my migration and the part I found most useful. My journals contained more than daily reflections: they held reading notes, meetings and other structured entries that worked better as separate pages.

You don't need to extract all of yours. Start with entries that have a clear identity and useful properties, and leave ordinary capture where it is. You can come back to the rest when you have a reason to.

For a fictional journal entry titled “Reading practice”, a cautious extraction would:

1. Copy the entire relevant subtree, including children and attached properties.
2. Create a uniquely named destination note.
3. Add only the page-level fields that describe that entry.
4. Leave a link at its original journal position.
5. Record the source date, source path and any block IDs in the mapping report.
6. Check incoming references before removing the copied subtree from the converted journal.

Do not route by title alone. Two journal entries can share a title and be different things. If no rule fits, put the entry in `Needs review/` and report it. An unrouted note you can inspect is far better than an omitted one.

Preserve a way back to the journal context. The surrounding day can explain why you captured something, even when the extracted note is clearer on its own.

Block references: map the target before replacing the syntax

Logseq references such as `((uuid))` need more than a character substitution. You first need to know which block the identifier belongs to and where that content will finish up.

Build a map from each complete source UUID to its destination file and block identifier. Keep the complete source UUID in that map even if the visible destination ID is shorter. Decide extraction and routing before resolving incoming references.

Do not assume an eight-character prefix is unique. I hit collisions in my own migration. Use a deterministic identifier scheme, check for duplicates in each destination note and fail visibly when the mapping is ambiguous. Retaining longer identifiers is usually less work than repairing wrong references later.

For a simple paragraph, Obsidian places the identifier after a space at the end:

```
Keep the capture process small enough to repeat. ^capture-note
```

A reference can then link or embed:

```
[[Pages/Reading practice#^capture-note]]
![[Pages/Reading practice#^capture-note]]
```

The exclamation mark embeds the target rather than merely linking to it. Structured blocks have different placement rules; a whole list or quotation uses an identifier separated by blank lines, while a particular list item can carry its identifier on the bullet. Check the intended target in Obsidian. Block-link rules.

My early conversion put some identifiers on the wrong content. The lesson is not “always use the previous line” or “always use the next line”. Identify the source block, then attach its destination ID to that block’s converted representation.

Where a block has moved from a journal to a separate note, update the destination map and all incoming references. Do not leave two competing copies without deciding which is authoritative. A reference that opens successfully can still be wrong if it displays unrelated text.

For missing targets, preserve the original UUID in a report. Visible markers such as `MISSING-REF` can help, but only if your converter deliberately creates them. They are not built-in Obsidian diagnostics. Some targets may already have been deleted in Logseq; distinguish those from conversion failures.

Namespaces and filenames: decide once, then update links

For a source name such as `Gardening/Soil/Compost`, choose whether the hierarchy should become folders or a flat filename. I used a mixture, based on whether I would actually browse a topic as a folder.

A simple flat option might be `Gardening - Soil - Compost.md`. A folder option might be `Gardening/Soil/Compost.md`. Neither is inherently more faithful; what matters is a complete mapping from the original name to the chosen destination.

Decode and sanitise filenames deliberately. Two different names can become identical after removing punctuation. Check collisions before writing, including names that differ only by case. Blank names, brackets and parentheses deserve explicit tests, not improvised regular-expression fixes.

Resolve links through the mapping rather than replacing slashes globally. A slash may be a namespace separator, a valid destination path or part of something unrelated. Check links in pages as well as journals, including property values and embedded references.

If you generate a parent index, preserve any existing parent-page content. Add the index in a clearly bounded section or a separate note. Never overwrite a real page because its name happens to match a folder.

Dates: standardise the known ones

Choose a journal date format and convert recognised date links to match. For example, `[[Jun 10th, 2025]]` can map to `[[2025-06-10]]` if that is the destination journal name.

Use an explicit set of accepted source formats. Report the rest. A link such as `[[Nov 1st]]` does not contain a year; do not quietly assign the migration year or the current year. Review its context, then make a deliberate decision.

Check ordinary date links and dates inside properties separately. A calendar date stored as metadata and a link to a daily note are related concepts, but they need not have the same representation.

Tasks and queries: preserve intent before rebuilding behaviour

My Logseq task setup was fairly simple, so this wasn't the hardest part for me. If you rely on it to manage your day, you'll want to spend more time here than I needed to.

A basic conversion can turn a source `TODO` item into an unchecked Markdown checkbox and `DONE` into a checked one. Do not automatically collapse every other state into “open”. A cancelled task, an item waiting on someone and a task in progress carry different information. Preserve that state in text or a chosen property until you have a destination workflow.

Keep deadlines, scheduled dates and recurrence instructions visible. Rebuild automation only after checking the behaviour of the tool you choose. A checkbox that renders correctly does not prove that its scheduling survived.

For queries, save the original definition and write its purpose in plain language. “Show unread articles” is a better migration requirement than “translate this query somehow”. Once the underlying articles have a consistent destination and status field, choose an appropriate Obsidian view or query tool.

Keep a short list of essential queries and compare their results with the source. Retire unused queries deliberately. Do not strip all query blocks before recording what they were doing.

Outlining and app-specific syntax

Preserve indentation on the first pass. Flattening every nested list can destroy context, particularly where a child only makes sense under its parent. Improve the shape of notes later as you use them.

Review these patterns in the converted copy:

- `collapsed::` and other confirmed display-state fields.
- Renderer macros and plugin-generated instructions.
- `#+BEGIN_QUOTE` and matching closing markers.
- Tip, warning or note blocks that need a readable destination.
- Query blocks and other executable-looking source syntax.

Convert a matched quote block to an ordinary Markdown quotation if that suits the content. Flag an unmatched opening or closing marker instead of swallowing everything that follows. Preserve unfamiliar blocks as readable source text pending review.

The fluidity of Logseq’s outliner is one of the things I missed. Optional outlining tools can help, but test your own daily actions rather than expecting a complete replacement. If a plugin behaves oddly, reproduce it in the small test vault before treating it as a migration failure.

6. Treat attachments as a separate migration

Copying the assets folder is necessary, but it does not prove the links work. Relative paths depend on where the referring note lives. A link such as `../assets/diagram.png` can break when you move a note one folder deeper or extract it from a journal.

For every attachment reference you check, distinguish three questions:

1. Does the source binary exist?
2. Was that binary copied intact to the intended destination?
3. Does the converted reference resolve to it from the new note?

If the source file is missing, path rewriting cannot recover it. Record that as a source gap. If the file exists but the reference is wrong, it is a conversion problem. My own asset work included both, which is why file counts alone are not enough.

Avoid renaming all binaries during the first pass. If you need clearer names, maintain a mapping and rewrite references together. Compare file sizes or hashes for copied binaries where practical, and open representative images and PDFs inside the destination vault.

PDFs and annotations

Check the document, the annotation text and the links between them separately. An extracted highlight may have its own identifier and may be referenced from other notes. Preserve those relationships in the same mapping process used for other moved blocks.

Use a PDF with several annotations as a sample. Confirm that the correct PDF opens, the annotation text survives, any page information remains useful, and references to annotations reach the intended text. A visible PDF attachment is only one part of the result.

I now use an optional split between binary PDFs and notes about them. That is an organisational choice, not a migration requirement. Keeping them together initially is fine. If you later exclude a binary-library folder from sync, remember that a working desktop link does not mean the PDF is available elsewhere.

Whiteboards, drawings and flashcards need their own checks too. Keep the original data. If you make an image or text fallback, label it as a fallback; it does not preserve interactive editing or review history. Decide whether a specialised conversion is worth doing after the main notes are safe.

7. Review the output before trusting it

Run the sample conversion first, manually or with reviewed automation. Open the result in Obsidian and compare it with the source side by side. Read the content, follow the links, inspect the properties and open the attachments.

When a rule fails, correct the rule and regenerate the sample into a fresh folder. Keep a short regression list so the same repair does not break another pattern. Only expand to the full working copy when the representative sample behaves as intended.

Reconcile transformations, not just totals

A migration that extracts journal blocks can legitimately produce more files than it started with. A migration that combines duplicates can produce fewer. Neither count proves success.

Account for source objects with a report that records whether each was preserved, transformed, combined, intentionally excluded or left unresolved. For extracted content, record both its new destination and the link left behind. Review every exclusion and unresolved item.

Inspect duplicate-looking files before combining them. Matching names are not proof of matching content. Preserve conflicting versions until you understand the difference.

Treat decoding errors as failures to investigate. Do not silently replace invalid bytes and call the result clean. Retain the original bytes, identify the affected file, inspect its encoding and use a deliberate fallback only after checking the recovered text. Search for replacement characters in output, while remembering that corruption may already exist in the source.

Search for leftovers

Search results are review leads, not automatic deletion instructions. Look for:

- `((` and `{{`, which may indicate unconverted references or macros.
- `id::`, `collapsed::` and other source properties.
- `#+BEGIN_` and `#+END_` markers.
- Old namespace forms and natural-language date links.
- Missing-reference markers your converter uses.
- Duplicate destination block IDs within a note.
- Invalid YAML, repeated keys and redundant classifications.
- Empty output notes and items in `Needs review/`.

Some matches will be intentional examples or quoted source material. Check before changing them. Also test wrong-target links: a successful click is not sufficient if it lands on the wrong “Reading notes” page.

Acceptance checklist

Work through this list before you switch over, and add anything your own graph needs.

☐ The source archive and independent backup are readable and untouched.

☐ Every source file or extracted object is accounted for in the report.

☐ Essential pages and journal entries retain their complete text and nesting.

- ☐ Moved blocks have the correct destination and incoming references.
- ☐ Block identifiers are unambiguous and point to the intended content.
- ☐ Namespace and renamed-file links work from pages and journals.
- ☐ Properties are valid and preserve the intended meaning.
- ☐ Dates are consistent; unresolved dates have not been guessed.
- ☐ Open, completed and exceptional task states remain distinguishable.
- ☐ Essential queries have been rebuilt and checked, or a fallback is recorded.
- ☐ Attachments have been checked independently of Markdown file counts.
- ☐ Missing assets, decoding failures and unknown patterns are documented.
- ☐ Duplicate and unrouted notes have explicit decisions or review owners.
- ☐ Daily capture, retrieval and your other essential workflows work in practice.
- ☐ A backup of the accepted destination exists before further restructuring.

You don't have to fix every old problem before moving on. Write down what remains and why you're comfortable leaving it for now. “A handful of missing images already absent from the source” is different from “we have not checked images”.

8. Switch over without losing the last week

Choose a cutover point. Capture the source changes made since the original snapshot, then review those changes into the destination. If a full rerun is necessary, keep it separate from any Obsidian notes you have already edited.

For an existing destination vault, compare paths and content before merging. An identical filename is a conflict to resolve, not permission to overwrite. Back up the destination immediately before the merge and keep a record of what changed.

Once you switch, make it clear which application is the place for new writing. Keeping both active indefinitely creates another reconciliation project. Retain the Logseq archive for reference and recovery; there is no need to delete it to make the move feel complete.

Test the new vault during ordinary work. Capture a thought, find an old note, follow an important reference, add an attachment and complete a task. If you use several devices, repeat the relevant checks there after sync has completed. Keep a small fix list as you go. You don't need to interrupt every writing session to redesign the system.

Work with your notes, not against them

The useful payoff for me was a vault I could understand and operate on. AI-assisted search, summaries and small filing jobs became easier to fit into that structure. They still need boundaries.

Begin with read-only requests over a limited set of notes. Ask for source links with summaries so you can check them. For changes, request a proposed file list and diff before broad edits, keep a backup and exclude material the tool should not access.

Write a short instruction file for whichever tool you use: where new notes belong, which folders are off-limits, what naming rules apply and which actions require review. You do not need to export your entire personal operating system or reorganise the vault before an assistant can be useful.

Leave bulk deduplication and mass restructuring until you trust the migrated content. Those can be worthwhile projects, but combining them with a migration makes it harder to tell where a missing paragraph went.

How I use Obsidian now

Once the migration is settled, there is room to improve how the notes support your thinking. Three habits shape how I approach this now. You don't need to apply them to every old note, or finish setting them up before you start using the vault.

Make the outline say something

I use a Pyramid Principle approach when a note needs to explain a decision or develop an argument: put the main point first, then group the reasons and evidence beneath it. A parent bullet should summarise what its children say. A label such as "Benefits" tells me less than a full sentence explaining the benefit.

Here is a fictional example, with invented observations and source-note names:

- **A weekly reading review would help me use more of what I capture.**
  - **Useful ideas are getting lost between capture and writing.**
    - Two recent writing sessions repeated research already saved in the notes.
    - Source: the fictional "Writing session observations" note.
  - **A short review could connect those ideas to work already underway.**
    - Choose one saved idea and link it to an active draft.
    - Source: the fictional "Reading review experiment" note.
- **The time cost is still uncertain, so I would try this for a month before making it a routine.**

The outline makes the reasoning visible without pretending that the question is settled. Keep tensions, unanswered questions and links to the original material. Splitting every sentence into another bullet doesn't create a useful hierarchy. And an exploratory journal entry doesn't need to become a polished argument just because you've moved it to Obsidian.

Leave a readable place to pick up

When I work with AI, I want the current state of the work to remain in readable notes. A chat or an assistant's memory shouldn't be the only place that knows what we decided.

A short Markdown note is often enough. This fictional example uses ordinary links to other notes in the same folder:

```
# Reading review experiment

Decision: Try a short weekly review for one month.
Sources: [Observations](writing-observations.md)
and [Review plan](reading-review-plan.md).
Next pickup: Review three saved articles and link
one useful idea to the current draft.
Open question: Is the review useful enough to repeat?
```

Keep the decision with the project or note it belongs to, and update it when things change. Link back to the evidence rather than copying an unsupported summary into several places. The aim is to be able to resume the work yourself, or with another tool, without reconstructing a whole conversation.

Keep notes, sources and finished work connected

I distinguish between the notes that explain the work, the source material behind it, and the files produced from it. They don't all have to live in the vault. A project note can link to its source notes and finished output; an output can point back to the project note or include a short source record alongside it.

If a file lives outside the vault, check its backup and device access separately. A working link on one computer doesn't mean the file is backed up or available on another. Test the links from the devices you actually use, and keep enough context in the note to understand what an unavailable file contains. This is a practical boundary to check, not a reason to reorganise everything at once.

Conclusion

For me, moving was worth the work. My notes ended up in a shape that better suited how I wanted to read, write and use them. I also gave up parts of an outliner I liked.

You do not need to settle every folder, replace every query or clean every old journal before you can use Obsidian. Preserve the source, prove the important transformations on a sample, account for what moved and make the remaining gaps visible.

What matters is being able to find what you need and keep working with your notes. Give yourself time to settle into the new system. You'll learn what needs changing by using it.

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.