SumizAI .com Your Second Brain, Powered by Intelligence.
https://sumizai.com/blog/how-a-chat-answer-becomes-a-note.html

UNDER THE HOOD

How does a chat answer become a note?

The interesting part of SumizAI is a short pipeline that runs after a conversation turn finishes. **Four stages, each with one job**, and a few decisions inside them that determine whether your library is worth having in six months.

Updated: 2026-08-14 **8** questions 4 min read

Q. What happens while the answer is still arriving?

Your question is written down before the model is called. That ordering is deliberate: if the provider is down, rate-limits you or returns an error, the question is still there and the failed turn is marked so you can retry it in place rather than typing it again.

The answer then streams token by token over a server-sent event channel, so you read it as it is produced rather than waiting for a wall of text.

One consequence worth knowing: if you close the tab mid-answer, the server keeps going. The response has already been paid for at the provider, so it is finished and stored along with its note rather than thrown away.

Q. Stage one — what does the draft generator produce?

A structured draft rather than a blob. The model is asked to return a **title**, a **summary** and an **expansion**, and those become the three visible parts of the note.

The summary is written to be read on its own, in a list, when you are scanning for the right note. The expansion is the part that has to survive being read a year later without the conversation around it.

Alongside the draft, the full original question and the full original answer are stored as the note's source material. That record is append-only for the life of the note — no later operation edits or removes it.

Q. Stage two — how is the chapter chosen?

By selection from a numbered list, not by naming. The model is shown the vault's existing chapters, numbered, and asked to answer with a number — or to ask for a sub-chapter beneath one of them.

It reads a compressed view of the tree rather than the whole thing: a digest capped at about 4 000 characters, so the decision stays affordable in a vault with thousands of notes.

A new branch is created only when nothing fits, and the vault's own name is never used as a chapter — that would just add a root above the root. Chapters whose names differ only by accents are merged when the tree is read, so a stray *Warzywa* and *warzywa* do not become two shelves.

Q. Stage three — what does the duplicate check compare?

First, candidates. Full-text search and trigram similarity over the vault produce a shortlist of notes that might be about the same thing. This is cheap and runs in milliseconds even over thousands of notes.

Then, judgement. A second model — the auxiliary one, not the one you are talking to — reads the candidate and the new material and returns **a score between 0 and 1 with a written justification**. Cheap retrieval narrows the field; an expensive reader makes the call.

What happens next is the vault's setting: ask you, merge automatically, or always create. In ask mode the decision can wait — it does not block you from carrying on with the conversation.

Q. What exactly does a merge do to the existing note?

It adds to it. The new material joins the note, and the new question and answer are appended to the source material alongside the ones already there.

What a merge never does is edit or delete the existing source. This is enforced in the code rather than by convention: the service and the repository simply have no update or delete path for it.

The reason is evidential. A merged note is a claim assembled from several conversations, and the only thing that makes such a claim checkable later is an unedited record of what each conversation actually said.

Q. Stage four — what gets written to disk?

A Markdown file, named after the slug of the title, in the vault's `notes/` directory. If that name is taken the app appends `(1)`, `(2)` and so on — and it asks the storage what exists rather than trusting the index.

The file carries front matter and the note body, and it is complete on its own: everything the index knows can be rebuilt from the files. That is what the reindex operation does, and in any disagreement the file is treated as correct.

The vault's `base.md` is regenerated too, so the table of contents on disk keeps matching the one in the app, with relative links that work in any Markdown tool.

Q. Where do the links inside the note come from?

From the same stage that writes the body. The model is given a map of up to eight candidate notes — title to filename — drawn from the search results, and asked to work a link into a sentence where it genuinely belongs.

Every `.md` target that is not in that map is stripped on the way in. The model does not get to invent a filename that looks right, because a link to a file that does not exist is worse than no link at all. External `http` links are left alone.

There is also a vault-wide *rebuild links* action for going back over everything at once, which is the one to use after importing a batch of material.

Q. Can I turn the automatic notes off?

Yes, per conversation. Some conversations are thinking out loud and should not leave anything behind, and the switch exists for exactly that.

The conversation itself is still stored — questions and answers, in order — so nothing is lost. What does not happen is the pipeline: no draft, no filing, no duplicate check, no file.

In practice most people leave it on and rely on the merge to keep the noise down, but the switch is there when you know in advance that a session is scratch work.