A Mac Workflow for Tracking Daily Research Progress

Ronald G. Thomas

2025-04-12

Figure 1: A desk calendar open to today's date, a pen resting across it, dog-eared pages beneath: a running daily log

Structured workflows turn scattered research notes into a searchable knowledge base.

Introduction

I did not really know how to maintain a consistent research log until I started losing track of what I had accomplished across ten concurrent projects. The problem was not motivation; it was friction. Opening a text editor, remembering the right file, formatting the entry, and committing the changes felt like too many steps for something that should take sixty seconds.

The first version of this idea was sketched out but never actually built. It was a three-script pipeline: dictate into ChatGPT's web interface, copy the summary to the clipboard, and run a script to append it to a single tagged log file. What I actually ended up building and using every

day is different in almost every respect. It is one script, `tn`, and it talks to Google's Gemini API directly through `curl`, with no browser and no clipboard hop in the middle.

This post documents `tn` as it exists on disk today: what it stores, how the pieces fit together, and where its rough edges are. Some of those rough edges are worth calling out explicitly. They are the kind of thing that is easy to miss when a script has grown past its original design in small increments.

More formally, this documents the file-system layer of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist. The convention that all research projects live under `~/prj/NN-name/` and are synced continuously by Dropbox is the substrate `tn` depends on. Its notes directory, `~/prj/research_update`, is itself a Dropbox-synced path, even though (as covered below) it is not a Git repository.

Motivations

- I was managing ten research projects simultaneously and could not remember which analyses I had run two weeks earlier on any given project.
- A single log file tagged by project name, which is what the original design called for, does not support editing or deleting an individual entry without hand-editing the file. I wanted commands for that.
- Handwritten lab notebooks did not support full-text search, which made retrieval slow and unreliable.
- Round-tripping through a browser tab (paste prompt, dictate, copy summary, switch back to the terminal) was more friction than dictating directly and letting a script call the summarization API itself.
- I needed the ability to revise a summary Gemini produced, not just accept or reject it outright.
- Voice dictation felt like the lowest-friction input method, but raw dictation output is too messy to store directly.

Objectives

1. Store research notes per project, one text file per project name, inside a single central directory.
2. Send dictated or typed notes to the Gemini API for summarization, from inside the same script that captures them, with no manual copy-paste step.
3. Support an accept, reject, edit, or revise loop on every generated summary before it is saved.
4. Provide commands to review, edit, and delete past entries by index or date, not just append new ones.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Prerequisites and Setup

This workflow assumes a macOS environment with the following tools available:

- **Terminal**: Any terminal emulator (iTerm2, Kitty, or the built-in Terminal.app)
- **Bash**: The script is written for `bash`, not a POSIX-portable shell
- **`jq`**: Builds the JSON request bodies sent to the Gemini API and parses the responses
- **`curl`**: Makes the HTTPS calls to the Gemini API
- **A Gemini API key**: Set `GEMINI_API_KEY` in `~/.env`; the script sources that file with `set -a` so plain `NAME=value` lines are exported automatically
- **`ripgrep`** (`rg`): Used by the `-q` / `--quick` flag, described below, though that flag currently has no matching data source
- **macOS System Events**: Only needed for `-i` / `--input`, which pops an `osascript` dialog so dictation (double-tap Fn) can be used instead of typing into the terminal

No R packages are required. The entire tool runs through the shell:

```
brew install jq ripgrep
```

What Is `tn`?

`tn` is a symlink in `~/bin` pointing at the actual script, `ai-notes-gemini`:

```
$ ls -la ~/bin/tn
lrwxr-xr-x 1 zenn staff 15 Jun 27 10:55 tn -> ai-notes-gemini
```

It is a single ~720-line bash script, not a pipeline of smaller scripts. Every note-taking, reviewing, editing, and deleting operation is a mode of the same executable, dispatched from a `case` statement over its command-line flags.

Each project gets its own file, `~/prj/research_update/<project>_notes.txt`, and each entry in that file follows a fixed, delimited format:

```
===== ENTRY: 2025-04-09 14:32:07 =====

--- RAW ---
<verbatim dictated or typed notes>

--- SUMMARY ---
<Gemini-generated summary>
```

That `===== ENTRY:` delimiter is what every review, edit, and delete operation locates entries by. The RAW section preserves exactly what was captured, and the SUMMARY section holds whatever the researcher accepted, edited, or asked Gemini to revise. Keeping both means the summary is never the only surviving copy of a note.

Getting Started: Notes Directory Layout

The central notes directory is fixed inside the script:

```
notes_dir="$HOME/prj/research_update"
```

Because `~/prj` is a symlink into Dropbox, this directory is continuously synced off-machine, the same as any other project directory in the Workflow Construct. It is, however, **not a Git repository** on this machine, so there is no commit history, no diff view, and no way to recover a specific past state of a note file beyond whatever Dropbox's own version history retains. The original design for this workflow assumed Git would provide an audit trail; the script that actually got built does not touch Git at all.

Running `ls ~/prj/research_update` shows one text file per project the tool has been used on:

```
11-blockchain-curriculum_notes.txt
fisherexacttestrx2_notes.txt
prj_notes.txt
zzcollab_notes.txt
zzlongplot_notes.txt
```

No setup step creates these files ahead of time; `tn` creates `${proj}_notes.txt` the first time notes are appended for a project whose name has not been seen before, after asking for confirmation (see below).

Figure 2: A single handwritten index card filed behind a tabbed divider in an open wooden card box: one day's entry in a running log

Deeper Analysis: How `tn` Works

Capturing and Summarizing Notes (the default mode)

Running `tn` with no flags other than an optional project name captures notes and sends them to Gemini for summarization. The project name defaults to the current directory's basename:

```
proj="${proj:-$(basename "$PWD")}"
```

If the project's notes file does not exist yet, the script confirms the inferred name before proceeding rather than silently creating a file under the wrong name:

```
if [ -z "$review_date" ] && [ -z "$last_n" ] && [ ! -f "$notes_file" ]; then
  read -r -p "First use for project '$proj'. Is this correct? [Y/n] " confirm
  ...
fi
```

Input comes from one of two places. By default, the script reads from standard input until `Ctrl-D`. With `-i` / `--input`, it instead opens a native macOS dialog via `osascript`, which supports dictation (double-tap the Fn key) without needing a browser tab:

```
raw_notes_content=$(osascript <<EOF
  tell application "System Events"
    activate
    set dialogResult to display dialog "Enter notes for: $proj" & return & return & "(Dictation: press Fn
      default answer "" ¬
      buttons {"Cancel", "OK"} default button "OK" ¬
      with title "tn - Research Notes"
    return text returned of dialogResult
  end tell
EOF
) || { echo "Cancelled."; exit 0; }
```

The captured text is sent to Gemini through `summarize_notes`, which builds the request with `jq` and posts it with `curl`:

```
summarize_notes() {
  local notes_content="$1"
  local prompt
  prompt=$(cat <<EOF
You are Gemini, assisting an academic biostatistician.
Summarize the following research notes concisely.
- Set the line length to 74 characters.
- The tone should be direct and clear.
EOF
)
```

```
  local json_payload
  json_payload=$(jq -n --arg prompt "$prompt" --arg notes "$notes_content" \
    '{ "contents": [ { "parts": [ { "text": $prompt }, { "text": "\n\n" }, { "text": $notes } ] } ] }')

  curl -s "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash:generateContent?key=$
    -H 'Content-Type: application/json' -X POST -d "$json_payload" \
    | jq -r '.candidates[0].content.parts[0].text'
}
```

The returned summary is shown, and the script drops into an accept, reject, edit, or revise loop before writing anything:

```
read -r -p "Accept summary? [Y/n/e(dit)/r(evise)] " choice
```

- `Y` (default) appends the entry as-is.
- `n` discards the summary entirely; nothing is written.
- `e` opens the summary in `$EDITOR` (falling back to `vim`) and appends the edited text.
- `r` prompts for free-text revision instructions, sends the current summary and those instructions back to Gemini via `revise_summary`, and loops so the new result can itself be accepted, edited, or revised again.

Only an accepted (or edited) summary is written, via `append_entry`, which appends the full RAW-plus-SUMMARY block to the project's notes file with a single `>>` redirect.

Reviewing Past Entries: `-r`, `-n`, `-R`

Both review modes route through a shared `awk` state-machine, `filter_entries`, that numbers entries as it prints them:

```
awk -v show_raw="$show_raw" -v date="$date_filter" -v num="$start_num" '
  /^===== ENTRY:/ {
    in_entry = (date == "" || $0 ~ date)
    in_summary = 0
    if (in_entry) {
      sub(/^===== ENTRY:/, "===== [" num++ "] ENTRY:")
      print
    }
    next
  }
  /^--- RAW ---/ { in_summary = 0; if (in_entry && show_raw == "true") print; next }
  /^--- SUMMARY ---/ { in_summary = 1; if (in_entry && show_raw == "true") print; next }
  in_entry && (show_raw == "true" || in_summary) { print }
'
```

`tn -r myproject` lists every entry for `myproject`; `tn -r 2025-04-09 myproject` filters to entries whose header line matches that date substring. `tn -n 3 myproject` shows only the three most recent entries, located by counting delimiter lines from the end of the file. In both cases, the SUMMARY section is shown by default; add `-R` / `--raw` to include the RAW section as well.

Editing and Deleting Entries: `-e` and `-d`

Both commands locate an entry by 1-based index, with negative indices counting from the end (`-1` is the newest entry), and both work by finding the line range that entry occupies:

```
[ "$idx" -lt 0 ] && idx=$((total + idx + 1))
...
start_line=$(grep -n "^$DELIM" "$file" | sed -n "${idx}p" | cut -d: -f1)
```

`tn -d -1 myproject` previews the entry (up to 20 lines), asks for `[y/N]` confirmation, and then deletes the line range with:

```
sed -i '' -e "${start_line},${end_line}d" -e '/./,$!d' "$file"
```

`tn -e -1 myproject` opens just that entry's block in `$EDITOR`, optionally re-summarizes the edited RAW text through Gemini (running the same accept/edit/revise loop as note-taking), and then rebuilds the notes file as before-block, edited-block, after-block, writing to a temp file and `mv`-ing it over the original rather than editing in place:

```
rebuilt=$(mktemp)
[ "$start_line" -gt 1 ] && sed -n "1,$((start_line - 1))p" "$file" > "$rebuilt"
cat "$block" >> "$rebuilt"
[ "$end_line" -lt "$last_line" ] && sed -n "$((end_line + 1)),${last_line}p" "$file" >> "$rebuilt"
mv "$rebuilt" "$file"
```

That difference between the two commands, in-place `sed -i` for delete versus tempfile-and-`mv` for edit, is covered in the gotchas below.

Listing Projects and the Quick-Lookup Gap: `-l` and `-q`

`tn -l` globs `*_notes.txt` in the notes directory and prints each project name with its entry count:

```
for f in "$notes_dir"/*_notes.txt; do
  [ -f "$f" ] || continue
  proj=$(basename "$f" _notes.txt)
  count=$(grep -c "^$DELIM" "$f")
  echo "  $proj ($count entries)"
done
```

`tn -q` / `--quick` is meant to show notes for the current project without specifying a name, but it searches a different file than everything else in the script:

```
current_dir=$(basename "$PWD")
daily_log="$notes_dir/daily_log.md"
if [ -f "$daily_log" ]; then
  rg "$current_dir" "$daily_log" | cut -c1-
else
  echo "No daily_log.md found in $notes_dir"
fi
```

Nothing in `tn` writes to `daily_log.md`. It is a holdover from the earlier tagged-single-log design that never got wired up to the per-project notes files this script actually maintains. On this machine, no such file exists in `~/prj/research_update`, so `-q` currently always falls into the "No daily_log.md found" branch. `tn -l` followed by `tn -r <project>` is the working substitute.

Things to Watch Out For

1. **`--delete` edits the notes file in place; `--edit` deliberately does not.** `delete_entry` runs `sed -i ''` directly against a file inside `~/prj/research_update`, which is a Dropbox-synced path. `edit_entry`, in the same script, explicitly avoids in-place editing for this reason, writing a fresh tempfile and `mv`-ing it over the original instead. In-place editors that write-then-rename can race with a sync in progress on a cloud-mounted directory; `delete_entry` does not have that protection. Deleting entries during an active Dropbox sync is the highest-risk operation in this script.
2. **`-q` / `--quick` does not read from where notes are written.** It searches `daily_log.md`, a file nothing in `tn` populates and which does not currently exist. Treat `-q` as unimplemented rather than broken, and use `-r` with an explicit project name instead.
3. **No version control on the notes files themselves.** `~/prj/research_update` is not a Git repository. Notes are protected only by Dropbox's file-sync history, not by commit-level diffs or an audit trail.
4. **Every note-taking or edit-with-resummarize call requires network access and a valid `GEMINI_API_KEY`.** There is no retry or offline fallback; a failed `curl` call surfaces whatever Gemini's API returned, unparsed, through the `jq -r` pipeline.
5. **Delete confirmation shows only the first 20 lines** (`head -20`) of the entry being removed. A long entry will not fully preview before the `[y/N]` prompt.
6. **Project directory naming still matters for `-l`.** Because notes files are keyed by `basename "$PWD"`, entering the wrong directory before running `tn` silently starts (or appends to) a differently-named project file.

Figure 3: A stack of identical dated notebooks of increasing height, the newest on top still open: accumulated daily records over time

A Typical Mid-Analysis Session

Everything above describes what `tn` does in isolation. In practice it is one step inside a larger session on a zzc-scaffolded analysis repo, not project initiation and not the final push toward publishing. Session five or session ten on an established project looks the same every time: resume where the last session left off, do the analysis work, then close out cleanly before switching to something else. `tn` sits at both ends of that pattern, not in the middle.

Before: Resuming the Session

1. **Reattach rather than start fresh.** If a session for the project is already running, reattach it; otherwise `cd` into the project directory.
2. **Check for uncommitted work from last time** with `git status`. A session that ended without a commit is a sign step 5 below got skipped previously; resolve it before layering new changes on top.
3. **Recall the last entry** before diving back in: `tn -n 1 <project>`. This surfaces whatever was logged at the end of the previous session, what was tried, what broke, what was next, without reconstructing it from memory or from the code alone.
4. **Enter the container** with `make r` (or `make docker-rstudio`). Any package installed from inside the container during the session is captured automatically in `renv.lock` when the R session exits.

During: The Analysis Work Itself

This part has no fixed shape: iterating on code, rendering intermediate output, checking results against expectations. That is precisely why sessions five and ten look different from session one, there is no setup checklist to work through here, just the analysis.

After: Closing Out the Session

1. **Exit the container** back to the host shell.
2. **Validate dependencies** with `make check-renv`, run from the host, not from inside the container. This catches any package that got used during the session but never made it into `DESCRIPTION` or `renv.lock`.
3. **Run the test suite** if the session changed anything beyond exploratory scratch work, with `make docker-test`. Skip this for a session that was pure data exploration with no code changes worth testing.
4. **Commit the work**, with a message describing what changed, not a placeholder. `zzgit` (or `git add` and `git commit` directly) stages, scans, and commits in one step.
5. **Log the session with `tn`, not with the commit message.** The commit message is a record of what changed in the code; the `tn` entry is a record of what was learned, what did not work, and what the next session should pick up. Running `tn` (or `tn -i` for a dictated version) captures that, and the accept, edit, revise loop means it takes a few seconds even when the day's work does not summarize itself cleanly.
6. **Push, if working across more than one machine**, and do not assume this backs up the `tn` entry too. Notes captured by `tn` are not part of the project's Git history (see the limitations above); pushing the code commit does nothing for the day's `tn` entry, since `~/prj/research_update` lives outside the project repository entirely.

Point 6 is worth restating on its own: the `tn` entry and the project's Git commit are two separate trails that happen to close out at the same moment in this workflow, not one combined mechanism. Losing sight of that is an easy way to assume notes are backed up by a `git push` that never touched them.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- A single script that calls the summarization API directly removes an entire manual step, and its associated failure mode, compared with a workflow that round-trips through a browser tab and the clipboard.
- Separate per-project files, rather than one tagged central log, make deletion and editing tractable: operating on `${proj}_notes.txt` means every index and line-range calculation only has to reason about one project's entries.
- Keeping both the raw dictation and the generated summary in every entry means an unsatisfactory summary is recoverable without re-dictating the notes.

- A script can accrete real functionality, review, edit, delete, well past its original scope, while still leaving a piece of the original design (`daily_log.md`) unconnected. Reading the code, not just the `--help` text, is the only way to find that kind of gap.

Technical Skills:

- Using `jq -n --arg` to build a JSON request body from shell variables safely, and `jq -r` to extract a nested field from the response, without any manual string escaping.
- Writing an `awk` state machine that tracks which section of a multi-line, delimited record it is currently inside (`in_entry`, `in_summary`) rather than trying to express the whole thing as a single regular expression.
- Converting a negative, "from the end" index to a positive one with `$((total + idx + 1))`, and reusing that conversion identically in both the delete and edit code paths.
- Preferring `mktemp` plus `mv` over `sed -i` when a file lives on a cloud-synced path, and recognizing that this preference has to be applied consistently across every function that rewrites the same file.

Gotchas and Pitfalls:

- `sed -i ''` (the empty string is required on macOS's BSD `sed`) is easy to reach for out of habit even in a script that elsewhere goes out of its way to avoid in-place edits on synced paths.
- A feature described in `--help` text (`-q`, quick lookup) can be present in the flag-parsing logic and the dispatch table while never actually being wired to real data. The flag "working" (producing output, even if that output is a not-found message) is not the same as the flag doing what its help text claims.
- Environment variables sourced from `~/.env` via `set -a` are exported for the whole script, including every subshell `curl` call; a missing or stale `GEMINI_API_KEY` fails at the API call, not at startup, unless explicitly checked first (which the note-taking path does, but the edit-with-resummarize path only checks conditionally).

Limitations

- This workflow is macOS-specific due to its reliance on `osascript` and System Events for the `-i` dictation dialog.
- The system depends on Gemini API availability and a funded or quota-available API key. If the call fails, the raw notes are never lost (they exist in memory until the script exits) but they are also not saved automatically; there is no "save raw notes even if summarization fails" path.
- There is no automated backup beyond Dropbox's file sync. Unlike a Git-backed log, there is no way to inspect what a note file looked like before a given delete or edit.
- `-q` / `--quick` is effectively dead code on this machine: the file it reads does not exist, and nothing in the script would create it.
- `delete_entry`'s in-place `sed -i` is a real risk on a Dropbox-synced notes directory, more so than any other operation in the script.
- The workflow does not handle attachments, images, or structured data. It is text-only by design.

Opportunities for Improvement

1. **Harden `delete_entry` to match `edit_entry`.** Rewrite it to build the post-delete content in a tempfile and `mv` it into place, the same pattern `edit_entry` already uses, removing the one in-place edit left in the script.
2. **Either wire up `-q` or remove it.** Either have the append path also write a project-tagged line to `daily_log.md`, restoring the original quick-lookup design, or have `-q` fall through to `tn -r <current project>` against the per-project file that already holds the data.
3. **Initialize `~/prj/research_update` as a Git repository**, with an automatic commit after each append, edit, or delete, to restore the audit-trail guarantee the original design assumed Git would provide.
4. **A lockfile around edit and delete.** Two terminals running `tn -e` or `tn -d` against the same project at once could race on the same `mktemp`-plus-`mv` rebuild; a simple `flock` around the notes file would close that gap.
5. **Parameterize the Gemini model and prompt.** The model name (`gemini-2.5-flash`) and the summarization prompt are both hardcoded in `summarize_notes`; pulling them from `~/.env` or a config file would make the summarization style adjustable without editing the script.
6. **Log visualization.** A simple script that walks every `*_notes.txt` file and generates a timeline of entry counts per project over time would add a useful retrospective view, similar to what `-l` already sketches with its entry counts.

Wrapping Up

The script that is actually in daily use is simpler in one respect than the original three-script design (one executable, no clipboard hand-off) and considerably more capable in another (review, edit, delete, and a full revision loop on every generated summary). What I found most valuable was removing the browser from the loop entirely: dictate, or type, directly at the terminal or into a native macOS dialog, and let the script call Gemini itself.

Researchers managing multiple concurrent projects who want per-project note files with real edit and delete support, not just append-only logging, may find this approach worth adapting. Reading through `tn` end to end, rather than trusting its `--help` output, is also a useful reminder that a script's documentation and its actual behavior can drift apart as functionality is added incrementally; the `-q` gap here is a concrete example of exactly that.

In conclusion, four points merit emphasis. First, a single script calling an LLM API directly removes both the browser and the clipboard from the note-taking loop. Second, per-project files rather than one tagged central log make deletion and editing by index tractable. Third, consistency in how a script rewrites its own data files matters: `edit_entry`'s tempfile-and-`mv` pattern should have been applied to `delete_entry` as well, and was not. Fourth, a feature can exist in a script's flag parser and help text without being connected to real data, which is exactly the state of `-q` today.

See Also

Related posts:

- Unix Command-Line Workspace Setup for Data Science Development: Terminal and Zsh setup that complements this workflow
- Multi-Laptop macOS Bootstrap: Managing configuration files across machines
- Refactoring a Personal Toolbox: Scripts versus Shell Functions: The broader `~/bin` cleanup that flagged `tn` alongside other overlapping note-capture scripts
- The 55-Item Preparation Checklist: The one-time setup checklist for a new zzc project; the mid-analysis session workflow above is what happens on every session after that checklist is done

Key resources:

- Gemini API documentation: Reference for the `generateContent` endpoint used by `summarize_notes`
- `jq` manual: Reference for the JSON construction and parsing used throughout
- macOS Dictation guide (Apple Support): Setup instructions for dictation inside the `-i` dialog

Reproducibility

This post describes a shell-based workflow with no R analysis pipeline. To reproduce it, place the script on the `$PATH` and set the required API key:

```
mkdir -p ~/bin

cp ai-notes-gemini ~/bin/ai-notes-gemini
chmod +x ~/bin/ai-notes-gemini
ln -s ai-notes-gemini ~/bin/tn

echo 'GEMINI_API_KEY=your_key_here' >> ~/.env

export PATH="$HOME/bin:$PATH"
```

System requirements:

- macOS, for the `-i` dictation dialog (the rest of the script has no macOS-only dependency beyond that)
- `bash`, `jq`, `curl`
- A Gemini API key with `generateContent` access
- `ripgrep`, only if `-q` is wired up per the improvement suggestions above; unused otherwise

Files in this post:

File	Purpose
`ai-notes-gemini`	The script itself: capture, summarize, review, edit, delete
`tn`	Symlink to `ai-notes-gemini` on `$PATH`

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.

Related posts in this cluster

This post is part of the *Shell Scripting and Git Tooling* series. Recommended reading order:

1. Post 41: Refactoring a Personal Toolbox: Scripts versus Shell Functions
2. **Post 43: A Mac Workflow for Tracking Daily Research Progress** (this post)