AI CODING EXPERIMENTS SYNTHESIS

Coding texts with LLMs: some transferable learnings from a recent experiment series

Causal Map is software for a particular kind of qualitative research: you feed it interviews or documents, and it helps you map out the causal claims people make, that is, statements that one thing affected another ("the training improved my confidence", "the funding cuts meant we lost staff"). Its central AI task is therefore a large coding job: give a language model a long, messy transcript and ask it to return a list of structured items, each one a causal claim with a cause, an effect and a verbatim quote to back it up. There is no single right answer, the items involve judgement, and quality can only really be checked by a person reading each item against the source. In mid July 2026 we spent a week running about 34 numbered experiments on this task: 22 prompt versions, 9 models, 4 chunk sizes, several multi-model pipelines and three ways of judging output quality. The full log is 1,900 lines (`causal-map-extension/docs/plans/ai-coding-prompt-experiments...md`). This note is the distillation, written for readers with no special interest in causal mapping, because almost none of what we learned is specific to it. If your task is "get an LLM to pull many judgement-laden items out of long text", whether that is entity extraction, qualitative coding, screening studies for a systematic review or annotation at scale, most of this should transfer.

A few words that recur below. **Recall** means how much of the real content the model found; **precision** means how much of what it returned was right. A **chunk** is a slice of the source text sent to the model in one request, because long texts are usually processed in pieces. A **link** is our unit of output, one extracted claim of the form "this led to that". Because the effect of one claim is often the cause of another ("cuts led to stress" and "stress led to sickness" share "stress"), the links join up into a network, which is the map the user actually wants; a link that connects to nothing else is an **island**. None of the lessons below depend on this network aspect except where it is spelled out.

Key learnings

Measurement

1. **An LLM judge is a relative signal only. Human and LLM judges need to work together.** Reading hundreds of items by hand is slow, so like many teams we used a second LLM as a judge, asking it to rule on each extracted item. When we finally audited the judge against a human, it ran about 20 points optimistic and agreed with the human on "is this item usable?" just 63% of the time. Yet it also beat the human in places, catching an inverted cause-and-effect the human missed. So neither is a gold standard: an LLM judge can rank prompt A against prompt B, but its absolute

numbers mean little. Hand-check a real sample before shipping any conclusion; a 60-item audit overturned our headline result.

2. **We rejected a gold standard altogether.** The classic way to evaluate extraction is a gold standard: a hand-made list of the correct answers, and you score the model against it. For judgement-laden tasks this collapses, because two competent human coders produce different but equally valid extractions. Only the wrong items are certain, and they cannot be listed in advance. The workable instrument turned out to be a **verdict bank**: a stratified sample of real model outputs, each ruled OK or NOT by a human, against which any automated judge can then be scored.

3. **Run repetitions, even at temperature 0.** Temperature 0 is the setting that is supposed to make a model's output near-deterministic. It does not make it repeatable enough to trust one run: identical prompts on identical text returned counts of 4, 1, 0, 0 across four runs for the feature we were tuning. That spread covered the entire range our prompt variants differed by, so a week of single-run comparisons had measured noise. Treat any single-run gap under about 10 points as nothing, and re-run before believing anything.

Silent data loss

1. **Most "this model is bad" results were pipeline bugs.** Between your prompt and the model there is plumbing: API calls, retries, parsers, output files. Ours failed silently in seven distinct ways, and every one made a model look worse than it was: reading only the first part of a multi-part response; rate-limit errors ("429", the API saying "too busy, try again") dropped without retry; responses cut off mid-answer but flagged as complete; correct answers wrapped in markdown formatting and rejected by a validity check; runs overwriting each other's output files; an expired security token silently reused. The trap is that a run which lost half its data reads exactly like a cautious model. Before believing any run, confirm that every chunk actually came back.

2. **A fix to the test rig is only half a fix.** We fixed this missing-retry bug in our experimental harness, and the identical bug stayed live in the product for two more days, until a real user's transcript came back with 2 items instead of 40 and they concluded the AI was useless. Every time you fix the measuring tool, ask whether the product fails the same way.

Prompts

1. **Shorter prompts performed better, down to a floor.** Our prompt had grown the way prompts do: rules, exceptions, worked examples, warnings. Cutting it from 10,500 characters to 7,000, keeping every rule but trimming the prose and the examples, improved recall and precision together. Cutting further to 4,000 went too far: recall dropped 20% and the rules compressed to a phrase stopped protecting. Best guess at the mechanism: a shorter prompt keeps the model's attention on the text rather than on rule recitation, provided every rule still appears once.

2. **One prompt for all models.** It is tempting to keep a tuned prompt per model, and when two models behave differently on the same prompt it looks like proof that you need to. In our case the

real cause was a prompt that contradicted itself: a newly added section said restatement was acceptable while an older section still banned it, one model obeyed each branch, and we nearly shipped "these two models need different prompts", which was false. A per-model prompt also breaks the day that model retires. When you change a rule, search the whole prompt for the old version.

3. **Wording is a weak lever; step-by-step task framing is a strong one.** We wanted the model to also catch a rare class of item (claims that an influence was attempted and failed, for example "we ran the campaign but nothing changed"). Four rounds of rewording the main prompt, up to and including a full rewrite of the task definition, moved the count not at all. What worked was changing the task: a dedicated second pass over the text asking only for that class found nearly all of them, and stayed quiet on a text that contained none.

4. **Naming a bad pattern can, perversely, teach it.** We added a rule banning one specific bad output form, complete with an example of the form. The frequency of that form went up, from 3 runs in 6 to 5 in 6. Showing the model what not to do put the pattern in front of it.

5. **Models satisfice on sweep tasks.** Asked to "go back over the output and find what you missed", the model returned 1 to 3 items per call, apparently because nothing obliged it to do more than plausibly comply. Restructuring the same request as exhaustive accounting, where every island in the output must either get an answer or be explicitly marked "none", made the same model return 15 in one round. When a sweep returns little, suspect the contract you set before suspecting the model.

6. **The same trick busts satisficing on the main task, which is where it pays.** The same model satisfices on the main extraction too, and this is why big chunks under-code (point 18): over a large context it returns a plausible handful and stops. The fix is the accounting contract from point 10, applied one level down. Divide each chunk into numbered segments (roughly a paragraph each) and require the model to account for every one: at least two claims from each segment where they exist, or an explicit "none". This lifted recall by 20 to 40% at moderate chunk sizes, by three to four times at the whole-document extreme where a plain pass collapses to a handful of items, and roughly halved the wasted effort. The decisive detail is that the escape must be **costly**: an easy "none" gets rubber-stamped without reading, so we require the model to quote the segment's most causal-sounding phrase and say in a few words why it is not codeable. A free skip is abused; a skip that has to be argued for gets the segment read. This softens the hard chunk-size tradeoff of point 18 without removing it: in our runs a plain 16,000-character chunk kept between a seventh and a third of the links a 2,000-character run found, and adding per-segment accounting brought that back to roughly half to three quarters, depending on the text, at an eighth of the requests. The accounting makes big chunks usable; small chunks still won on raw recall in every comparison. Two cautions. It has a ceiling: past about 32,000 characters the accounting itself degenerates into rubber-stamped "none"s, because the segment list grows longer than the model will genuinely read. And on clean prose it can be neutral or slightly negative, because a short careful paragraph really does hold one claim, not two, so the floor pushes at nothing. It earns its keep on messy, claim-dense material and on large chunks.

Models (July 2026, Gemini and Claude on Vertex AI, EU residency required)

1. **Models differ on multiple dimensions.** The pattern was consistent by model rather than by size or cost. gemini-2.5-flash: high recall, 17% wrong. gemini-3.1-pro, the expensive one: only 2% wrong but it missed over half the real content, and its restraint was avoidance of the faint or debatable claims rather than better selection. gemini-3.5-flash: the surprise, cheap and precise but finding a third of the volume, and our eventual default. gemini-3.1-flash-lite: cheapest by far, quality swinging from text to text. claude-sonnet-4-6 and claude-haiku-4-5: two to three times the error rate of the others on this task, and ruled out.

2. ***Over-produce and prune* beats *under-produce and pad*.** Two ways to combine a generous model and a careful one. The design that worked: let a cheap high-recall model produce too much, then have a more careful model prune, which kept ~95% of the good items and removed ~70% of the wrong ones. The reverse design, asking the careful model to code and then padding its output with a "find more" pass, bought recall by lowering its bar and added errors. And the pruner must be a different model: pruning with the same model that coded left twice the errors in place, presumably because it likes its own work.

3. **But reverse the roles and the same pairing fails: a candidate list makes a strong model worse and dearer.** This sounds like a contradiction of 13, and the difference is which model authors the output. In 13 the cheap model writes the finished items and the strong model only deletes; deletion is a set of small yes/no decisions on completed work, and it went well. Here we tried the opposite handoff: the cheap model skims the text and passes a rough list of candidates to the strong model, which authors the final output while "focusing" on the shortlist. The strong model's internal reasoning quadrupled, because it weighed every candidate it was about to reject, and it anchored on the draft instead of reading the text fresh, more severely the more of the text the list covered. A rough draft contaminates the model that has to write over it; finished work merely gets filtered. The one part worth keeping: when the cheap model flagged rival readings of an ambiguous passage as explicit alternatives, the strong model picked the right one, fixing an error it had made when working alone.

4. **Turning thinking down raised output and cut cost by two thirds.** Reasoning models spend hidden "thinking" tokens before answering, and Gemini bills those at output rates: at our default setting, 77% of what we paid for was thought. Dialling thinking to minimal, against all intuition, made the model find more good items. The new errors it introduced fell in one recoverable class (cause and effect the wrong way round), fixable by a later pass, rather than invented content.

5. **One question beats twelve.** Our main prompt asks the model to decide many things per item at once: the claim, the labels, several metadata flags. A follow-up pass that revisited each finished item with a single question ("does this one flag apply to this item?") scored 100%, 14 of 14, including a case the same model had got wrong when deciding everything at once. For reclassification jobs, a focused second look beats loading every decision into the first pass.

6. **Capacity starvation is per model.** When a cloud model is overloaded, requests fail with "resource exhausted" errors. We assumed a sibling model from the same family on the same route would be failing too, so it could not serve as a fallback. Measured during a real outage: the starved model failed 6 probes out of 6 while two same-family models on the same route served 6 out of 6. Capacity is provisioned per model, and a sibling is a valid fallback.

Pipeline design

1. **Chunk size dwarfed every prompt effect we measured.** How much text you send per request ("chunking") mattered more than anything we did to the prompt. Chunks of 2,000 characters found 50% more good items than 4,000. Fed a whole 89,000-character document in one request, the model returned 6 items where chunked runs returned 114: past a certain input length it stops extracting and summarises to a token budget, whatever the instructions say. Larger chunks are also less repeatable, with run-to-run overlap falling from 0.93 at 2,000 characters to 0.55 at 8,000. Extraction over a big context behaves like drawing a sample.

2. **Chunking loses the connections between chunks, and no amount of renaming buys them back.** The price of small chunks: the model can only report what it sees in one slice, so a cause discussed early in a document and its effect discussed pages later never come out as one claim, and instead of the joined-up network the user wants, the output arrives as scattered fragments. We tested every repair that works on the output alone, mostly passes that rename near-identical wordings so that claims about the same thing meet up ("staff morale" and "morale of the team" becoming one node). None of it reconnected anything, because the missing connections were claims that had never been extracted at all rather than matching claims under different names. What did work was a second pass over the whole text explicitly hunting for the missing connections. But such a pass finds real missed claims and also invents some, and the model does not reliably flag which is which, so every item it adds must be gated by a mechanical check in code (for us: the supporting quote must appear verbatim in the source, checked by string matching rather than by the model).

3. **Prompt caching was blocked three ways and worth little anyway.** Providers discount input tokens they have recently seen, which sounds like the fix when you re-send a long prompt with every chunk. For our Gemini pipeline it failed on three independent counts: our prompt was assembled with the varying part first, so there was no constant prefix to cache; Gemini's automatic caching never fired on the EU route we are required to use; and its explicit caching was not available EU-resident at all. Two of the three are provider-specific: Anthropic's caching works differently (you mark the cacheable prefix explicitly in the request), and when we used Claude for a separate feature it cached fine on the same EU setup. Only the first block, prompt assembled wrong way round, is universal, and it is the one to check first in any codebase. Caching also would not have paid here: while thinking dominated the bill, perfect caching would have saved 21%. Shortening the prompt achieved the same end with no infrastructure.

Process

1. **Reasoning about why a prompt behaves as it does, from a handful of outputs, failed four times in one session.** Each time we read some output, formed a plausible mechanism, changed the prompt accordingly, and the change did nothing. What settled the question was a positive control: an old prompt known to produce the desired behaviour, run under exactly the current conditions. It localised the true cause quickly.

2. **A change that touches two modes must be measured in both.** Our product runs the same coding task in two output modes. We evaluated a new default thoroughly in one mode, shipped it to both, and silently broke the other. The metric that would have caught it already existed in our own tooling and was simply never looked at, because the runs that would have shown it were never made.

3. **External LLM judging needs a budget rule.** Automated judging via API might seem free per call, but routine judge runs cost $200 in two days. The standing rule now: a human (or the coding assistant, in this case Claude Code) judges by reading; a paid model judges only on explicit request.

Where it landed for us

Production default: gemini-3.5-flash, single pass, 2,000-character chunks, minimal-length core prompt composed from independent layers. And lots of findings specific to causal mapping. We are now moving that default towards larger chunks (16,000 characters) with the per-segment accounting of point 11, followed by the island-joining pass of point 19: together these keep the recall of small chunks while restoring the connected map that small chunks destroy.